ABSTRACT

ZAHAN, NUSRAT. Evaluation of Security Practices Towards the Enhancement of Software Supply Chain Security. (Under the direction of Laurie Williams).

CONTEXT: High-profile supply chain attacks and regulatory mandates like Executive Order 14028 and the EU's Cyber Resilience Act have compelled organizations worldwide to adopt software security practices. Organizations increasingly turn to security frameworks for comprehensive guidelines on securing the software supply chain.

PROBLEM: While different security frameworks offer guidelines and extensive lists of practices to prevent supply chain attacks, verifying the effectiveness of these practices in enhancing product security remains challenging. To make the best use of available resources, practitioners seek to make informed decisions about adopting security practices guided by scientific evidence that demonstrates an impact on project security outcomes.

OBJECTIVE: *The goal of this thesis is to aid practitioners in prioritizing the adoption of security practices to prevent software supply chain attacks through the empirical evaluation of software security practices and security outcomes in the software supply chain.*

APPROACH: This research follows a systematic progression from prevention to measurement to evaluation. We begin by studying proactive practices that help organizations avoid SSC attacks, including identifying metrics to prevent account takeover attacks and proposing effective automated code review techniques to detect malicious dependencies. We then shift focus to understanding the trend and evaluating the adoption of practices in open-source ecosystems. Finally, we quantify the effectiveness of these security practice adoptions through security outcome metrics, establishing empirically grounded links between what developers do and how secure their software becomes. Throughout this work, we hypothesize that maintainers who adhere to good security practices are more likely to achieve measurable improvements in security outcomes. By testing this hypothesis through data-driven evaluation, we aim to understand whether the adoption of software security practices contributes to better security outcomes, offering a practical, evidence-based path toward improved SSC security, especially in resource-constrained environments.

Evaluation of Security Practices Towards the Enhancement of Software Supply Chain Security

by
Nusrat Zahan

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2025

APPROVED BY:

Bradley Reaves	Alexandros Kapravelos
William Enck	Laurie Williams Chair of Advisory Committee

BIOGRAPHY

Nusrat Zahan is a researcher specializing in software engineering and software security. Born and raised in Bangladesh, she earned her undergraduate degree in Electronics and Communication Engineering from Khulna University of Engineering & Technology. Following graduation, Nusrat gained industry experience working for three years at NEC Corporation in Dhaka, Bangladesh.

ACKNOWLEDGEMENTS

This doctoral thesis marks the end of an extraordinary journey, one that has transformed me, challenged me, and shaped the person and researcher I have become. Throughout this challenging yet rewarding path, I have been incredibly fortunate to receive the unwavering support, encouragement, and inspiration of many individuals, to whom I owe my deepest gratitude.

First and foremost, I am profoundly thankful to my advisor, Dr. Laurie Williams, for believing in me from the very beginning and guiding me through every challenge. Her mentorship, encouragement, and steadfast support have been instrumental in shaping my development as a researcher.

I extend my sincere appreciation to my committee members, Dr. William Enck, Dr. Alexandros Kapravelos, and Dr. Bradley Reaves, for their guidance, thoughtful feedback, and constructive critiques that helped strengthen this work and broaden its impact.

I am also deeply grateful to the brilliant colleagues and fellow researchers I had the privilege of working with throughout my doctoral journey. To all the students from WSPR, SCI, and S3C2, thank you for fostering a strong sense of community, support, and shared purpose. I am especially thankful to the past and present members of the Realsearch group, whose advice, collaboration, and mentorship guided me in countless ways throughout this journey.

To my dearest friends, thank you for your constant encouragement and for standing by me through both moments of triumph and times of doubt.

Above all, my deepest gratitude goes to my family, my parents and siblings, for being my unwavering support. To Shohan, the love of my life, whose constant support, patience, and belief in me made this journey possible. And to our little miracle Adeel, who has already brought so much joy and perspective into my life even before arriving, thank you for being with me during this final stretch. I look forward to beginning our next chapter together.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Open-source software has revolutionized the development of modern applications, enabling rapid innovation through reusable components. The growing reliance on third-party components in modern applications expands the attack surface, exposing applications to Common Vulnerabilities and Exposures (CVE) [144] exploits or malicious code injection, facilitating software supply chain (SSC) attacks. For instance, in the 9th State of the Software Supply Chain Report [229], Sonatype reported that CVEs have surged by 463% in the last 10 years, and since 2019, more than 700,000 malicious packages have been proactively identified. While facilitating modular and rapid development, SSC also introduces inherent risks, particularly in the form of SSC attacks. The high degree of automation provided by package managers in ecosystems like npm and PyPI can obscure the underlying dependency chain from end users, who may implicitly trust each component, often without visibility into the full transitive graph or any awareness of the maintainers of those maintaining third-party components. The SSC risk is amplified by the hidden complexity of the interconnected third-party components chain, where a single compromised component can affect thousands of downstream users.

In the software development lifecycle (SDLC), applications often rely on external components or packages known as direct dependencies. Each of these direct dependencies can, in turn, have additional dependencies, referred to as indirect or transitive dependencies. Together, these interconnected dependencies, along with their contributors and maintainers, form the foundation of the open-source software supply chain. Overall, the SSC encompasses all components, libraries, tools, processes, and individuals involved in the development, building, and delivery of a software application [231, 262]. Attackers can exploit dependencies, build infrastructure, and even the humans involved in development to execute an attack targeting thousands of downstream users [262]. The SSC attacks often target dependencies by injecting malicious code or exploiting accidentally introduced existing vulnerabilities, leveraging outdated or insecure components, or compromising developers through social engineering and account takeovers [127, 126]. Additionally, the build process itself can be a point of compromise, with threats emerging from tampered configurations in continuous

integration and deployment (CI/CD) pipelines or from vulnerabilities in development tools such as integrated development environments (IDEs) [127, 262, 126]. While non-reproducible builds do not directly increase the likelihood of attack, they hinder the ability to detect unauthorized changes, making it more difficult to ensure build integrity.

Recent high-profile incidents, such as Log4Shell, SolarWinds, and the XZ Utils backdoor, have underscored the multifaceted nature of SSC security risks. The Log4Shell vulnerability [71, 157] exemplifies the exploitation of a severe, unintentionally introduced vulnerability in a widely used open-source logging library, highlighting the risks of unintentional vulnerabilities in dependencies. In contrast, the SolarWinds attack [48, 223, 73] demonstrated how compromising the build infrastructure of a trusted vendor can allow attackers to inject malicious code into software updates, bypassing traditional security defenses. As a result, the malicious SolarWinds' Orion platform was downloaded by approximately 18,000 customers, including government agencies and providers of critical infrastructure [46]. More recently, the XZ Utils backdoor exemplifies an emerging threat where attackers infiltrate open-source software (OSS) communities by gaining trust and intentionally implanting malicious code into core dependencies, posing a direct challenge to dependency trust and maintainership models [76, 101, 195]. The exploitation of the trusting trust problem [240] of OSS has elevated SSC attacks to one of the most pressing threats in today's cybersecurity landscape, as highlighted by ENISA [66]. The Allianz Risk Barometer [17] and Bank of England Systemic Risk Survey [21] both identify cyberattacks as the top business risk, with the latter also ranking them as the most difficult organizational risk to manage. Given the foundational role OSS plays in the modern digital economy, safeguarding the OSS supply chain has become a matter of public interest and is now recognized as part of national security strategy [238, 237, 116]. Consequently, both the academic community and industry have increasingly turned their attention to this issue, aiming to develop systematic approaches and robust defenses to secure the OSS supply chain. *The goal of this thesis is to aid practitioners in prioritizing the adoption of security practices to prevent software supply chain attacks through the empirical evaluation of software security practices and security outcomes in the software supply chain.*

1.1 Problem Statement

As software ecosystems become increasingly interconnected through third-party dependencies, organizations face escalating risks from compromised maintainers, malicious packages, and insecure development practices within their dependency graphs. This challenge is particularly acute in the context of dependency security, where downstream projects inherit security risks from third-party components over which they have little visibility or control. The urgency of addressing SSC security has been recognized by stakeholders worldwide. Many stakeholders have recognized this urgent concern; most prominently, Section 4 of US Executive Order (EO) 14028 [238] calls for enhancing the security of software products purchased by the US government through the adoption of software security practices. Similarly, the European Union's Cyber Resilience Act (CRA), approved in March 2024, mandates common cybersecurity standards for digital products, including incident reporting and automatic updates [116]. The implications of not complying with the CRA include significant fines, the inability to sell a product in the EU, and reputational damage.

These global government actions have compelled software organizations to implement good security practices to strengthen their supply chain security. As organizations seek to address the escalating risks of SSC attacks and comply with these emerging mandates, they increasingly turn to embrace security frameworks

to guide the adoption and assessment of security practices. Government and industry have developed software supply chain risk management frameworks, compiling what practices software organizations should adopt to reduce security risk. Frameworks such as Building Security In Maturity Model (BSIMM) [31], the U.S. National Institute of Standards and Technology (NIST) Secure Software Development Framework (SSDF) [230], Supply-chain Levels for Software Artifacts (SLSA) [217], OpenSSF SCorecard [206], Open Source Software (OSS) Secure Supply Chain (SSC) Consumption Framework (S2C2F) [140], Proactive Software Supply Chain Risk Management (P-SSCRM) [263], OWASP Software Component Verification Standard (SCVS) [177] and others provide comprehensive guidelines and structured checklists for securing the SSC. However, in practice, organizations often lack the resources, whether budgetary, temporal, or personnel, to implement and evaluate all recommended practices. More interestingly, none of these frameworks provides sufficient empirical evidence to help organizations prioritize which practices will yield the most significant impact in their specific operational contexts. Recognizing this gap, agencies CISA and NIST have emphasized the need for measurable, evidence-based approaches to assess security risk and identify high-impact security practices [39, 41, 40, 109, 108, 47]. Therefore, practitioners stand to benefit from data-driven empirical studies that link security practices to improved security outcomes. Such research can help inform strategic decision-making, enabling organizations to adopt the most effective and contextually relevant practices to improve their SSC security posture. The thesis statement of this work is the following:

The adoption of software security practices, including vulnerability prevention and detection practices, leads to better security outcomes, as measured by vulnerability count and dependency update metrics.

This thesis addresses the identified gap by empirically investigating the relationship between security practice adoption and security outcomes within the open-source ecosystem. The research focuses specifically on dependencies as an attack vector and empirically investigates how the adoption of security practices relates to measurable improvements in security outcomes. The work investigates four key areas:

- Identify potential SSC risks in the supply chain by analyzing package and maintainer metadata, contributing to early prevention of account takeovers or social engineering attacks.
- Detect malicious code in dependencies proactively.
- Assess whether a project's dependencies adhere to secure development practices.
- Evaluate whether the adopted security practices are improving security outcomes.

Security practices are actions, procedures, techniques, or other measures that an organization can adopt to reduce the vulnerability of an information system [160]. Examples of security practices include whether a project is actively maintained, conducting code reviews before merging pull requests, and utilizing static or dynamic analysis tools as detection measures for vulnerabilities. In this paper, we used the term "security practice", also synonymous with security control, tasks, safeguards, and countermeasures [160].

Security outcomes are quantifiable indicators of the effectiveness of security practices prescribed for an information system [237]. Security outcomes measure the degree to which security practices reduce harm to

users and systems. Security outcome metrics can include both industry-level measures, such as reductions in attacks, incident frequency, or financial losses, and internal measures, such as reduced vulnerability exposure (e.g., lower vulnerability counts) and faster mean time to respond, which shortens the window of exposure to threats. Outcome metrics can serve as the feedback loop needed to assess and evolve security strategies in complex, adaptive environments.

This research follows a systematic progression from prevention to measurement to evaluation. We begin by studying proactive practices that help organizations avoid SSC attacks, including identifying metrics to prevent account takeover attacks and proposing effective automated code review techniques to detect malicious dependencies. We then shift focus to understanding the trend and evaluating the adoption of practices in open-source ecosystems. Finally, we investigate the effectiveness of these security practice adoptions through security outcome metrics, establishing empirically grounded links between what developers do and how secure their software becomes.

Throughout this work, we hypothesize that maintainers who adhere to good security practices are more likely to achieve measurable improvements in security outcomes. By testing this hypothesis through data-driven evaluation, we aim to understand whether the adoption of software security practices contributes to better security outcomes, offering a practical, evidence-based path toward improved SSC security, especially in resource-constrained environments.

1.2 Contribution

This dissertation makes the following main contributions:

1. **Weak Link Signals (ICSE SEIP, 2022)** We proposed six weak link signals that are indicative of security weakness in npm SSC: 1) expired email domain; 2) install scripts; 3) unmaintained packages; 4) too many maintainers; 5) too many contributors; and 6) overloaded maintainers. An attacker can exploit these signals to execute an account takeover attack. We conducted the study from a data-driven attacker hat perspective to understand how an attacker can collect and analyze package metadata from the package registry to find and exploit the weakest links (e.g., less secure modules, maintainers) and then execute an attack in the targeted supply chain. We conducted a survey completed by 470 npm package maintainers to obtain the practitioners' feedback on our weak link signals. Developers supported three of our six proposed weak link signals: expired email domains, install scripts, and unmaintained packages. We conducted three case studies to understand the impact of these weak links in the SSC. We found that (1) weak link signals exist in popular packages and present a higher supply chain risk; (2) weak link signals can aid in detecting malicious samples; and (3) how data-driven attackers can combine multiple weak link signals to execute a supply chain attack.

   Impact: This work has been recognized and adopted in the industry, and security specialists are now promoting these signals as part of proactive SSC security [269]. The expired domain was a novel signal proposed in this study and is now widely accepted in the industry. Major ecosystems, including npm and GitHub, enforced two-factor authentication on top of developers' accounts to prevent account takeover attacks. Industry tools (Socket [2], packj [175], SDC-Check [20], guarddog [50], and others) integrated these signals as automated supply chain security checks.

2. **Malicious Dependency Detection (MSR 2024, ICSE 2025)** We proposed a malicious package detection

workflow leveraging large language models to review code for malicious samples. We also constructed a benchmark dataset containing both malware and neutral packages. Our study can be used as a baseline study in the security domain. We enhanced existing prompt techniques by incorporating malware-domain-specific prompts and instructions on the expected outcome. We conducted a baseline comparison of the GPT-3.5 and GPT-4 models with CodeQL, which showed promising results for the GPT models with low false-positive alert rates. Our workflows aid in reducing the burden of manual review on security analysts and offer explainable findings with minimal misclassification. The study also highlighted the current limitations in ChatGPT, especially the mode collapse, difficulties of intraprocedural large file analysis, and prompt injection, areas that necessitate further research for the models' effective utilization.

Impact: Practitioners can utilize our workflow to reduce the code review burden while maintaining a minimal false positive rate. Additionally, researchers and tool developers can use our workflow and prompts to evaluate other LLMs and assess their performance across more diverse datasets.

3. **Security Practice and Security Outcome (IEEE S&P Magazine, ICSE SEIP'23, Submitted)** The work explores evidence on how adopting security practices impacts the security outcome. To measure the adoption of security practices, we evaluate the OpenSSF Scorecard tool [206], an automated tool to monitor the security health of the OSS supply chain that auto-generates a "security score" for OSS projects using a list of security metrics. We used the tool to understand the gaps in the current adoption of security practices and evaluate the Scorecard tool's applicability for npm and PyPI ecosystems. In another study, we explore the vulnerability count of a package as a security outcome metric by evaluating the relationship between Scorecard security practice metrics and security outcomes. Since the scarcity of reported vulnerability data makes the vulnerability count of a package metrics less reliable as security outcome metrics, we further explore additional security outcome metrics, such as utilizing comprehensive vulnerability count data from both package code and complete dependency trees and dependency update metrics (time to respond to shorten the window of exposure). We investigate the relationship between publicly available data on the adoption of security practices using Scorecards and the aforementioned security outcome metrics to understand which security practices have a meaningful impact on security outcomes. We also investigate whether project characteristics have an impact on security outcomes.

   Impact: The OpenSSF Scorecard team appreciated our research and committed to improving the Scorecard tool based on our findings in Scorecard version 5 for better ecosystem security measurement. Supported and funded by organizations such as Google and Cisco, this research advances an open research challenge in software security by deepening the understanding of the relationship between security practice and security outcome measurement, providing actionable insights for both practitioners and policymakers.

1.3 Thesis Organization

In Chapter 2, we provide background on key concepts and related work. Chapter 3 presents our first study on identifying weak link signals in the software supply chain. In Chapter 4, we present our benchmark dataset study constructed to conduct the study in Chapter 5. In Chapter 5, we present our study on automated code

review practice to detect malware. Chapter 6, Chapter 7, and Chapter 8 present our final work, where we investigate the relationship between the security practices and security outcome metrics.

CHAPTER

2

BACKGROUND AND RELATED WORK

2.1 Key Terminology

2.1.1 Software Supply Chain Security

The software supply chain encompasses everything that goes into or affects code from development through the CI/CD pipeline until production deployment in the software development life cycle (SDLC). Software supply chain security is securing the components, activities, and practices involved in creating and deploying software, including third-party and proprietary code, deployment techniques and infrastructure, interfaces and protocols, and developer practices and development tools [231]. Organizations are responsible for performing these security activities and for providing proof of their security efforts to consumers.

2.1.2 Software Supply Chain Attack

A software supply chain attack is a cyber-attack that aims to infect organizations and end-users by targeting less secure components in the supply chain [117]. Supply chain attacks are considered critical because of the increasing reliance on third-party packages as a direct and transitive code dependency. A single package dramatically increases a system's attack surface due to the "nested" nature of code dependencies [285].

2.1.3 Software Security Practice

Security practices are actions, procedures, and technical controls that an organization can adopt with the goal of preventing, detecting, and responding to threats against information systems. In this thesis, the term "security practice" is used interchangeably with security control, safeguard, countermeasure, or task [160]. Security practices encompass both technical and organizational measures that are typically standardized and

carried out by individuals or teams as part of an organization's efforts to protect its information, resources, and operations [121]. Security practices are either mandated or recommended by nationally and internationally recognized standards bodies—such as the National Institute of Standards and Technology (NIST) and the International Organization for Standardization (ISO)—to help prevent future security incidents [5]. Governments and industry organizations publicly provide various security standards, guidelines, and frameworks, which define recommended practices for securing software systems. Examples of security practices include whether a project is actively maintained, conducting code reviews before merging pull requests, and utilizing static or dynamic analysis tools as detection measures for vulnerabilities.

2.1.4 Software Security Outcome

Security outcomes are quantifiable indicators that reflect the effectiveness of security practices implemented within an information system [237]. They measure the extent to which such practices mitigate harm to users, systems, and organizations. Security outcome metrics may encompass both external, industry-level indicators, such as reductions in attack frequency, security incidents, or financial losses, and internal operational measures, such as decreased vulnerability exposure (e.g., lower vulnerability counts) and faster response times, which shorten the window of exposure to threats. These outcome metrics can be used as a feedback loop for evaluating and refining security strategies, particularly in complex and adaptive environments.

While industry-level metrics offer valuable insight into the broader impact of security practices, they are often difficult to obtain due to proprietary restrictions, underreporting & inconsistent data, complex costs, or challenges in attributing outcomes to specific interventions [208, 57, 80]. In contrast, internal-level measures are more readily available, observable, and actionable within development teams and organizations. These internal indicators can be consistently monitored to establish baseline trends and track progress over time. As organizations mature in their use of internal outcome metrics, the next step involves examining whether sustained improvements in these areas align with broader reductions in attacks, incident rates, user harm, or financial impact.

2.1.5 Software Security Metrics

Security metrics are quantifiable measurements used to understand the status of systems and services through the collection, analysis, and reporting of relevant data [83]. Security practices metrics focus on preventive and reactive actions, and security outcomes metrics aid in measuring the positive or negative impact of the use of security practices.

2.2 Guidelines and Frameworks

Software supply chain security has received increasing attention from government bodies, standardization organizations, industry, and open-source players due to the high impact of software supply chain attacks. In the following, we focus on the most prominent activities only.

The US government's concern for the increased security risk posed by the increasing threat due to mounting supply chain attacks is embodied in Section 4 of **Executive Order 14028** [238], which mandates increased use of security practices by suppliers who sell to the US government. The mandate includes the adoption of the software development controls documented in *NIST SP 800-218, Secure Software Development Framework*

(SSDF) [230]. NIST SP 800-161r1, Cybersecurity Supply Chain Risk Management Practices [3], overlaps with the SSDF and includes controls related to vendor/third-party management.

The **Software Component Verification Standard (SCVS)** [177] by OWASP is a framework to develop a common set of activities, controls, and security practices that can help in identifying and reducing risk in a software supply chain. There are 6 control families that contain 87 controls for different aspects of security verification or processes. The SCVS has three verification levels, where higher levels include additional controls.

The **Building Security In Maturity Model (BSIMM)**[31] is the result of a multiyear study of real-world software security initiatives (SSIs). Each year, various firms in different industry verticals use the BSIMM to manage their SSI improvements because the BSIMM report provides a clear picture of actual practices used by organizations across the security landscape [257]. The BSIMM is organized as a set of 122 activities, which consist of four domains distributed into 12 practices.

The **CNCF Technical Advisory Group (TAG)** [75] published a series of recommended best practices, tooling recommendations, and design considerations that can reduce the likelihood and overall impact of a successful supply chain attack. They discuss the 5 stage of software supply chain security—securing code, materials, building pipelines, artifacts, and deployments. The framework proposes 54 practices with associated risk factors to provide a holistic, end-to-end guide for organizations and teams.

In response to Section 4 of the President's Executive Order (EO) on "Improving the Nation's Cybersecurity (14028)" [238], the U.S. National Institute of Standards and Technology (NIST) improvised the **Secure Software Development Framework (SSDF)** [230]. The framework does not introduce new practices or define new terminology. Instead, the framework describes a set of high-level practices based on established standards [31, 30, 176, 75] of secure software development practices.

The **Proactive Software Supply Chain Risk Management (P-SSCRM)** framework [263], released in 2024, results from the study of real-world software supply chain risk management initiatives and the union of the 73 security practices in ten government and industry documents (standards and frameworks).

Industry has also documented mitigating controls, often endorsed by industry groups brought together through the *Linux Foundation's Open Source Security Foundation (OpenSSF) [6]* (is a cross-industry collaboration), such as the *Supply Chain Levels for Software Artifacts (SLSA) [217]*, which provides controls to fortify the build infrastructure attack vector, and the *Software Supply Chain Consumption Framework (S2C2F)*, Scorecard [206] which provides a secure open source software (OSS) ingestion pipeline to protect developers from supply chain threats.

Supply Chain Levels for Software Artifacts (SLSA) [217] is a security framework established by industry consensus, a set of standards and controls to prevent tampering, enhance the integrity, and also a set of security rules that may be adopted incrementally. SLSA has four levels, with SLSA 4 representing the ideal end state. The lower levels represent incremental milestones with corresponding incremental integrity guarantees.

In August 2022, the Cybersecurity and Infrastructure Security Agency (CISA) and the National Security Agency (NSA) released a recommended practice guide on **Securing the Software Supply Chain for Developers** [67]. The report advocated specific frameworks like SLSA and SSDF. The framework is a roadmap to building a healthy software development environment that includes a Software Bill of Materials (SBOM) for describing software components, active monitoring for vulnerabilities and software supply chain attacks, and a secure build environment and development team.

Microsoft released the **Open Source Software (OSS) Secure Supply Chain (SSC) Consumption Frame-**

work (S2C2F) [140], which is a security assurance and risk reduction process focused on securing how developers consume open source software. The framework provides security guidance and tools throughout the developer's inner-loop and outer-loop processes that have played a critical role in defending against and preventing supply chain attacks through the consumption of open-source software across Microsoft. In September 2022, the OpenSSF released the **npm Best Practices Guide**[170] to help JavaScript and TypeScript developers reduce the security risks associated with using open-source dependencies.

OpenSSF launched the **Scorecard** project [206] in November 2020 to provide an automated security tool that gives a "security score" for OSS and reduces the manual effort required to analyze a package's security. These results are made available via a BigQuery public dataset and the Open Source Insight (OSI) site. Additionally, practitioners can execute the Scorecard on a specific GitHub repository to evaluate the security practices of that repository.

Open Source Insights (OSI) or Deps.dev [174] is a Google-developed and hosted tool to aid practitioners in grabbing information about the source code location, package metadata, licenses, releases, and vulnerabilities of open-source products. OSI scans millions of open-source packages from different ecosystems, constructs dependency graphs, and annotates the metadata in a dashboard. Apart from the package's metadata, the OSI dashboard also shows statistics about a package's direct or transitive dependencies. On the OSI website, users can view the vulnerability mapping of a package as well as the vulnerability mapping with associated dependencies. In addition to the package metadata, OSI has also incorporated Scorecard security practices to help understand package security practices.

In-toto: A framework that enforces the integrity of a software supply chain by gathering cryptographically verifiable information about the chain itself [242, 113]. `in-toto` grants the end user the ability to verify the software's supply chain from the project's inception to its deployment. To achieve this, `in-toto` requires a project owner to declare and sign a layout of how the supply chain's steps must be carried out and by whom. The concerned parties will document their actions and produce a cryptographically signed declaration for each step as they are completed. The link metadata recorded from each step can be verified to ensure that each step was completed correctly. Both SLSA [217] and CNCF Technical Advisory Group (TAG) [75] recommend in-toto as a framework to secure the integrity of software supply chains.

2.3 Related Work

2.3.1 Software Supply Chain Attack

Several approaches have been proposed to detect and prevent supply chain attacks that can be classified as rule-based [60, 275, 81, 90], differential analysis [164, 249, 251, 204] and machine learning (ML)[163, 210] approaches. Rule-based approaches often rely on predefined rules about package metadata (e.g., package name), suspicious imports, and method calls. Many studies [60, 275, 81, 90] have focused on detecting static features like libraries that access the network, file system access, features in package metadata, dynamic analysis, or language-independent features like commit logs and repository metadata to flag packages as suspicious using heuristic rules. Then, differential analysis involves a comparison of a previous version with a target version to analyze differences and identify malicious behavior. These tools [164, 249, 251, 204] leverage artifacts, file hashes, and "phantom lines" to show differences between versions of packages and source code repositories. The studies [163, 210] evaluated various supervised ML models to identify effective ML models

for detecting malware. Additionally, multiple research studies [251, 235] exist on detecting typosquat attacks. Despite the existence of such vast research in this domain, Ohm et al. [166] revealed in a Systemization of Knowledge (SOK) study that none of the presented approaches can be considered a complete and isolated solution to detect malicious behavior. Rather, these approaches are an important part of the review process to reduce the workload of reviewers. Ohm et al. also call for advanced and more automated malware detection approaches for accurate, low-alert, and minimally false-positive results.

2.3.2 Security Outcome Metrics

Various metrics can assess the effectiveness of security practices by tracking security performance over time, generally categorized as threat-oriented, asset/impact-oriented, and vulnerability-oriented [155, 156]. While threat- and asset/impact-oriented metrics offer valuable insights, they are difficult to quantify at scale due to reliance on real-time threat intelligence and complex impact assessments. In contrast, vulnerability-oriented metrics are more accessible and relevant to software security. Among them, severity-based metrics (e.g., CVSS) face challenges such as scoring inconsistencies and reliance on exploitability data [180]. Temporal vulnerability metrics, such as vulnerability count, time to detect, time to repair and time to remediate, are widely used to evaluate security [148, 202, 41, 147]. Morrison et al. [147] conducted a systematic mapping study on software security metrics to evaluate the security properties of software and found that the most cited and most used metric is the vulnerability count (vulnerability metric). Davidson [51] studied the time to close bug/vulnerability (a form of mean time to repair) as a security metric and measured its effectiveness. Beres et al. [23] studied the window of exposure (vulnerability metric), which is the length of the interval of time needed by the security team to remediate a vulnerability once it has been disclosed. However, the scarcity of reported vulnerabilities, compounded by silently fixed issues, limits their reliability in assessing security outcomes [15, 61]. To address these limitations, prior work suggests alternative metrics, such as fault prediction models [212]. Additionally, in software reliability, MTTR and MTTU have long been used to measure system resilience and maintenance efficiency [88, 33]. While MTTR is often employed in security contexts [179, 182, 228], its reliance on reported vulnerabilities restricts its applicability. MTTU, by focusing on dependency updates, offers broader coverage but remains constrained by proprietary measurement methods [44].

2.3.3 Large Language Model (LLMs)

Different government reports by CISA and NIST highlighted the leverage of AI to reduce human loads as part of recommended security practices [41, 108]. LLMs are now widely applied in software engineering across various tasks, including code generation, testing, program repair, and summarization [14, 268]. In this section, we explore related work in LLM and its prospects in software security.

Zero-shot is a technique that enables models to tackle unseen tasks without prior examples. Zero-shot learning has shown promising results in various NLP tasks [29, 255], with Chain-of-thought (CoT) or step-by-step reasoning prompts enhancing LLMs' ability to produce intermediate reasoning steps [255]. Studies [267, 122, 123] have identified effective CoT components, such as zero-shot CoT and role-play prompting, that improve reasoning quality. Kojima et al.[122] demonstrated LLMs' proficiency in zero-shot reasoning, while Kong et al. [123] found that role-play prompting yields more effective CoT. Madaan et al. [137] observed that the SELF-REFINE method, applied to tasks ranging from dialog to reasoning using GPT-3.5 and GPT-

4, significantly improves performance, validated by human and automatic metrics. Zelikman et al. [277] confirmed the utility of self-generated rationales for model self-improvement.

White et al. [260, 259] demonstrated that augmenting the model with domain-specific information improves the model's performance since language models find more value in detecting specific identifiers in code structures when generating code summaries. Chen et al. [35] investigated the performance of GPT-3 code generation. They explored the model's ability to understand and generate code, offering valuable insights into the potential of using such models for code-related tasks. Ahmed et al. [14] showed that the GPT model can significantly surpass state-of-the-art models for code summarization, leveraging project-specific training. In another study, Sobania et al. [220] utilized ChatGPT for code bug fixing, further improving the success rate of bug fixing. Purba et al. [185] conducted a study measuring LLMs performance in detecting software vulnerabilities. The authors concluded that LLMs do not perform well in detecting software vulnerabilities, but models are particularly good at finding common patterns that are involved in vulnerable code. The author also observed that ChatGPT 4.0 can explain the "intention" of a given code snippet.

CHAPTER

3

WEAK LINKS IN THE NPM SUPPLY CHAIN

Modern software development frequently uses third-party packages, raising the concern of supply chain security attacks. According to Snyk [107], 96% of applications use third-party packages, and 80% of the code in the software supply chain comes from third-party packages. The scope and scale of the expanding supply chain also come with high-security risks [285, 225]. Large package managers, like npm, maintain a centralized repository where developers can access and add third-party packages to their dependency tree easily [285, 60]. Unfortunately, attackers can also leverage the same features of npm and inject malicious package updates into a software supply chain.

A recent report from Sonatype shows supply chain attacks has increased 650% in 2021 on top of year-over-year growth of 430% in 2020 [225] where attackers injected malicious code into benign packages [283, 111, 247, 211, 233, 65]. An example of a sophisticated supply chain attack is the SolarWinds attack [84]. SolarWinds is a proprietary cybersecurity monitoring solution [107, 46] whose customers include 425 Fortune 500 companies and at least nine U.S. federal agencies [46]. More than 100 companies and federal agencies were exposed to the breach [56]. The attacker gained access to customer networks, systems, and data through a malicious package update [46]. An incident of this magnitude raises significant concerns about the consequences of supply chain attacks.

In supply chain attacks, instead of exploiting latent vulnerabilities in source code, attackers inject malware directly into benign code that is likely to be deployed by users [225, 165]. A common strategy is to target the most commonly used packages in a dependency chain to infect a maximum number of users [225]. In that way, bad actors can execute an attack that will propagate throughout the supply chain. Thus, popular large registries like npm, which hosts 1.8 million JavaScript packages as of 2021, are a highly targeted malware distribution channel for attackers due to heavy growth and dependence on JavaScript packages [117, 165, 225, 72, 90].

One way attackers can develop such supply chain attacks is by following a data-driven attack strategy. For example, an attacker can collect and analyze package metadata from the package registry to find and exploit

the weakest links (e.g., less secure module, maintainers) and then execute an attack in the targeted supply. The dynamic nature of such attacks challenges conventional detection methods [165, 60, 90, 72] because the attacks are new and spread fast without the victim knowing where bad actors planted the malware in the supply chain. Hence, practitioners need proactive approaches to identify warnings events that predict package susceptibility at low cost measures to prevent future supply chain attacks. *The goal of this work is to help software developers and security specialists in measuring npm supply chain weak link signals to prevent future supply chain attacks by empirically studying npm package metadata.*

Our work addresses the following research questions:

- **RQ1** (Quantification): What type of weak link signals exist in npm package supply chain?
- **RQ2** (Agreement): How do practitioners perceive the proposed weak link signals in the npm supply chain?

As we know, the saying- "*A chain is only as strong as its weakest link*". A weak signal is an indicator of an event or change that may become significant in the future [241] and can break the supply chain. In this study, **we define a signal as a weak link if the signal exposes a package to a higher risk of a supply chain attack and an attacker can exploit the signal to execute a supply chain attack**. We focus on the relationship between neutral and suspicious metadata in a supply chain. We propose six weak link signals for npm package dependencies. To that end, we perform an empirical study on the metadata of 1.63 million npm packages to quantify and prioritize weak link signals in the npm supply chain. This paper makes the following contributions:

- Six proposed weak link signals, three of which are confirmed as strong signals from a survey completed by 470 npm package maintainers.
- Eight new weak link signals suggested by survey participants.
- A framework[1] to collect, categorize and analyze package metadata in npm registry to evaluate weak link signals.

3.1 RQ1: Weak link signals

In this section, we describe our weak link signals and present quantification of those signals to answer RQ1. **We define a signal as a weak link if the signal exposes a package to a higher risk of a supply chain attack and an attacker can exploit the signal to execute a supply chain attack**. If a bad actor targets the supply chain, they are going to follow the path of least resistance [248], for example, finding weakest links from publicly available package metadata without penetrating the whole source code. In this study, we followed an approach the attackers might take to find the weakest link in the npm registry. An accurate estimation of the impact on a target application may not be given from a single weak link signal but identifying multiple weak link signals from different metadata aiming at a common target is probably a hints that this target will be impacted in future.

One of the critical challenges in identifying the weak link signals was selecting the suspicious candidate of metadata for further investigations. To overcome this, we took advantage of the prior reported supply chain attack patterns reported in literature [165, 60, 81] and the author's security domain expertise. Additionally, we

[1] Due to reduce malicious use of the scripts, please contact the authors for access to the framework.

discussed our observation with security specialists from GitHub working on npm supply chain security. Based on our discussion, we focused on metadata associated with human involvement (e.g., maintainer, contributor information) since many past attacks involved human actors as a weak link to the package [106, 38, 43, 111, 68]. Then we selected functional metadata (install scripts) [283, 211, 233, 247, 81, 60, 246, 120, 236, 138] that was exploited by attacker mostly. We also measure the attack surface (e.g., dependency, dependents) to understand the underlying reason that might lead to an attack attempt.

3.1.1 Data Process

Data Collection

We captured a snapshot of 1,630,101 package.json files on June 7, 2021 through the npm public API[2]. Each JSON file contains metadata properties of an npm package. We used the package ID (package name and version together form a unique identifier called "ID") as the primary reference while storing corresponding metadata for each package. As a root, package ID maps to all other metadata like dependency information, package last modified time, scripts, versions, license, repository, unpacked package size, a total number of files, maintainers and contributors names and email addresses. We used Python libraries (JSON,Numpy, Pandas, and Pyscopg2) to extract all the relevant metadata from each package.json file which we stored systematically in a PostgreSQL database. Each package.json file contained multiple versions history of package metadata against one unique ID. In this study, we only used the most recent version of the metadata to analyze our weak link indicators. Apart from metadata extraction, we analyzed the maintainer reach and package reach to understand the impact of a package and its maintainers in the npm registry. Below we explain how we queried and measured package and maintainer reach–

Package Reach: We measured the package reach to evaluate a package's popularity in terms of dependents and downloads. Hence, we considered two metrics: (1) the number of packages that depend on a package (dependents), which we computed from a set of all the packages that have a direct dependency on a package; and (2) the number of downloads of a package in the past 12 months which we collected from public npm API[3].

Maintainer Reach: We measured the number of unique dependents that depend on a maintainer's packages to evaluate the maintainer reach in npm. We queried against npm packages to compute maintainer reach, a set of unique packages where atleast one package of the maintainer is listed as a package dependency.

Exclusion Criteria

We removed packages that might introduce noise (for example, wrong ownership, invalid metadata, etc.). In this study, we removed a package if the package has no **dependents and it meets with any one of the following IF** conditions:

First. If a package is tagged as a **security holding package** [58] and is removed from the *npm* registry by the npm security team due to malicious activities. Hence, the JSON file we received for such packages consisted of dummy metadata filled by the npm team. We removed 8,344 packages as security holding packages. The "descriptions" and "dis-tags" property of the package.json file define the security holding status.

[2] `https://replicate.npmjs.com/_all_docs?include_docs=true`
[3] `https://api.npmjs.org/downloads/point/{period}[/{package}]`

Second. If a package is **deprecated** [161], meaning, the package is not actively maintained by the maintainers. The package.json file contains a separate property called "deprecated" to indicate package deprecated status, which is assigned by the npm administrators or the maintainers themselves. Though a deprecated package does not always indicate an unusable package, npm and maintainers recommend using alternative packages or versions. While the deprecated packages are not actively maintained, end-users might still use them directly or transitively. Hence, we only removed deprecated packages in the recent version if no other packages in the npm registry used them in their dependency tree. We removed 37,917 deprecated packages that used by none.

Third. If a package has **no repository and no license**. A repository is essential to track, organize, and validate the source coderights and a license allows the OSS community to reuse the code. A package without a license indicates that the authors retain all source code [162] and no repository is attached to verify otherwise. We have identified packages where both repository and license property was null or filled with invalid values (e.g., "UNLICENSED", "XYZ", "personal use" etc.). Hence, we only removed the packages from further considerations if they meets all three conditions: 1) no dependent **and**; 2) no license **and**; 3) no repository. We found 89,893 such packages and removed them from the database.

In total we removed **9%(135,996)** packages that fits into our proposed exclusion criteria, and our final dataset contained **1,494,105 packages**, which were used for further analysis.

3.1.2 Weak Link Signals

We briefly discuss six weak link signals proposed in this study, their attack models, and specific data analysis in this section.

W1: Expired Maintainer Domain

An attacker can hijack a component if a maintainer's domain is expired and does not have 2FA authentication set up on their account. In general, any domain name can be purchased from a domain registrar allowing the purchaser to connect to an email hosting service to get a personal email address. An attacker can hijack a user's domain to take over an account associated with that email address. Typically, a domain hijacking attack occurs by 1) gaining unauthorized access to the registrar or 2) gaining access to the owner's email address and then resetting the password. Domain hijacking is not a new notion. We have seen many attacks in the past, such as $perl.com$ hijacking [49], where the attacker changed the domain registrar and renewed the domain expiration date until 2029, and then changed the DNS address.

In the npm registry, an attacker can execute a more simplistic approach to hijack an email address. An attacker can track the domain of a maintainer in the domain registrar site. If the domain is expired and available for sale, the attacker can register and alter the DNS "mail exchange" (MX) records to hijack the maintainer's email address. In most cases, maintainer accounts are associated with an email address in the package registry. One could reset a npm account directly by email address unless the maintainer activated 2FA authentication or used different email address in user account.

Analysis of npm maintainers domain: To collect the maintainer email address, we extracted and stored the maintainer name and email address from each package.json file. We then queried and split the domain name from each email address and counted the number of times a domain is used across all npm packages. Out of 93K domains in the 1.63M npm packages, 86% were unique (appeared once), while the rest were from

the public or organizational domain. We hypothesize that all maintainer domains in the npm registry are up-to-date, and none of them are available for sale. To that end, we performed a bulk query of 93K domains in Godaddy a domain registrar site. We found 5,346 domains available for sale. We picked a random sample of 50 domains to determine the true positive rate and checked each domain independently in Godaddy. We found 33 of them are not for sale. Due to the high false-positive rate, we manually verified 5,346 domains in Godaddy.

We found 2,818 maintainers' domains are available for sale and can be purchased. These maintainers own 8,494 packages in the npm registry with average direct dependents of 2.43 packages and average downloads of 53K in the past 12 months.

We reported our findings to the npm security team to validate the email address for npm user accounts. We note that our bulk query may have a high false-negative rate since we found 47% false positives from the 5,346 domains. Hence,npm may have more than 2,818 maintainers associated with expired domains

W2: Installation Script:

An attacker can use installation scripts to run commands that perform malicious acts through the package installation step [234]. Install scripts run automatically either before, during, or after package installation when certain events are triggered. These scripts are used to make the installation process easy since they are automatically run by npm. However, for an attacker, such scripts create opportunities where "sky is the limit". The attacker could steal user-sensitive data or execute a new child process to create backdoor access or gain access to execute a series of commands remotely [283, 211, 233, 247, 81, 60]. Alternatively, the attacker can infiltrate the third-party dependency since that installation script will run automatically by the targeted package and its users during installation. The rc, coa, ua-parser-js [183], CCleaner [120], Solarwinds [46], NotPetya [138], and Adverline [236] data breaches are examples of such attacks where attackers targeted the remote server, third-party vendors to execute a large supply chain attack. Though the presence of installation scripts themselves are not a direct indication of maliciousness, the privilege to run automatically makes installation scripts a weak link signal in the supply chain. As best practices, even npm registry recommends avoiding install script- "*Don't use install...You should almost never have to explicitly set a preinstall or install script.*"[4].While evidence of such an attack through the installations script is not rare [283, 211, 233, 247, 81, 60, 246, 120, 236, 138], similar or perhaps even worse attacks may happen in the future.

Analysis of npm packages: To collect script details, we extracted and stored the script key, which is the script's name (e.g., preinstall), and the script value, which contains the script path/shell commands from the package.json file. Then, we query and separate all the packages that have script keys like "%Install%".

We found 2.2% (33,249) of packages use install scripts, indicating that 97.8% of packages may follow npm recommendation of not using the install script as best security practices.

Additionally, we collected 3,635 malicious packages.json files from npm. They are similar to the security holding package.json file (see Section 3.1.1) with all dummy metadata. The only difference is that these JSON files have actual malicious script key and value pairs, for example, the original directory of malicious code

[4] `https://docs.npmjs.com/cli/v7/using-npm/scripts`

files or a malicious shell script embedded in JSON files with other dummy metadata. To analyze the malicious JSON file, two researchers separately reviewed these scripts and compared results for verification. Since the scope of the project was limited to package.json files only, we were only able to analyze 485 (out of 3,635) packages where malicious shell command was embedded directly in the JSON file and 442 of them were in install scripts. From our analysis, we found four types of attack patterns that attackers use frequently-

- **Transfer Users data** to third party server (e.g.- `hostname, etc/shadow, /etc/passwd, /home/<user>/.ssh`). We found 344 packages that communicate and transfer data with third party server.
- **Download** the malicious tool and run it on the user's machine. For example, download a crypto miner software and run it on a user's machine. We found 115 packages.
- **Reverse Shell** 12 packages opened reverse shell and transferred data to a third-party server.
- **Removed directories** 14 packages removed file/folder from current directories.

We found 93.9% (3,412) of malicious packages had atleast one install scripts, indicating that malicious attackers use install scripts frequently.

W3: Unmaintained Package

Attackers can target packages that are more likely to take over and sneak in malware due to a lack of maintenance. Differentiating between unmaintained and feature-complete packages that require no further releases is complex and ambiguous [245]. Even if the package may not require any maintenance by itself, it may need maintenance due to security issues in its dependencies or use new syntax to improve performance, bug fixing & documentation improvements. In 2020, the average time to remediate security issues was 68 days in open source projects [142] and 66% of security vulnerabilities in npm packages remain unpatched [224]. Hence, the time required to remediate a security issue in unmaintained packages remains unknown. We considered unmaintained packages from two directions: 1) Inactive packages; and 2) Inactive maintainers.

Inactive Packages: We considered a package inactive if the package's last modification time in the package.json file is past two years. Other prior work [245, 224] has defined inactivity as one year gap. However, many npm packages have low complexity with a few lines of code and may not require any recent update. Thus, we consider two years as an inactivity period.

An attacker can exploit vulnerabilities in all applications that directly or transitively depend on vulnerable code as long as the vulnerable dependency or the package itself remains unfixed [285]. The response time to fix such issues is undetermined for inactive packages, and end-users may remain infected due to their unawareness of the security threat. A deprecated package (see Section 3.1.1) exposes an even higher security risk since the maintainers no longer maintain the package to fix security issues. An attacker can inject malware into widely used but unmaintained packages. A prominent example of such an attack is the "Mailparser" attack [45], an old deprecated package. An attacker indirectly reaches the "MailParser" through a relatively new package named "getcookies", which had indirectly been made into the nested dependency chain of MailParser.

Inactive Maintainers: We defined an inactive maintainer if the maintainer had no active packages in the past two years. An attacker can target packages with inactive maintainer(s) because any attack will

remain undetected due to the inactivity of maintainers. Therefore, distinguishing between active and inactive maintainers is necessary to identify the packages that require maintenance. Although inactive maintainer's packages are a subset of inactive packages, we must identify them separately. Because an inactive package may have maintainers who are active in other packages, whereas inactive maintainers indicate the maintainer is inactive in the entire npm registry.

Analysis of npm packages: We extracted and stored the "time" property of the package.json file to measure the number of packages that have been inactive for the past two years. We identified inactive maintainers by evaluating the last modified time properties for all packages corresponding to an individual maintainer. A package where none of the maintainers are active elsewhere in the entire package registry is determined as an inactive maintainer of unmaintained packages. We also considered deprecated packages as unmaintained since they are officially unmaintained by the maintainer. We separated the deprecated package where the last modification time passed our threshold value because the deprecation was declared later.

We found 58.7% of packages and 44.3% of maintainers are inactive in the npm registry. There are 5,532 additional deprecated packages where the deprecation date passed our threshold value.

The package.json does not provide the individual maintainer's activities history. Hence, distinguishing inactive maintainers from active packages was not plausible.

W4: Too many Maintainers

A package with too many maintainers will provide an attacker many targets to exploit account takeover and social engineering attacks. Having too many maintainers requires security assurance of many maintainers. The open nature of open source software and npm not enforcing the two-factor authentication make it harder to ensure the maintainer's secure account. Bad actors may leverage the possible oversight of a large team and hide their identity by compromising one maintainer profile or performing social engineering to access the package. The fewer number of maintainers facilitates better security in terms of better communication.

Our hypothesis is supported by the previous work of Meneely et al. [139] where they empirically showed that projects with more developers have more vulnerabilities. Zimmermann et al. [285] also addressed our concern, where they suggested that the value of over 20 maintainers in a npm package is questionable. A recent attack on inactive rc and coa [183] packages shows that additional or inactive maintainers pose a security threat. In both packages, malicious code was injected via the compromised maintainer's account in November. Since the attack, the current version has fewer maintainers than before. Though we do not know how the attacker compromised the maintainer account, we have observed the change in maintainer metadata after the attack was resolved.

Analysis of npm packages: Though the exact number of maintainers varies from project to project, in the case of npm, the number of maintainers is more likely to be less because npm is building upon reusing small JavaScript libraries. 1.5 million npm packages have an average cost of 1.7 maintainers, which makes sense as small packages may not require many maintainers. Hence, analyzing the whole data set is not practical to identify the unusual package with too many maintainers. We picked the top 1% (14,941) packages in npm ranked by the total number of maintainers. We extracted and stored the list of maintainers corresponding to each package in a SQL database and then ranked them by the total number of maintainers in the package.

We found that our selected 1% packages had an average of 32.4 maintainers per package, which was 19 times more than average package maintainers in the entire registry.

W5: Too many contributors

An attacker can sneak in malicious code, bypassing the maintainer's radar when a maintainer is responsible for many contributors. Many prior studies show that when multiple contributors change a file, the file is more likely to have more failures [25, 139, 150, 24], which may include security issues. These observations motivated our next signal: A maintainer-to-contributors ratio or an increased number of contributors increases the security risks.

Contributors may vary in knowledge, skill, and experience. Package quality will inevitably suffer if the maintainers do not pay enough attention to reviewing the pull request from contributors. Especially where an average of 1.7 maintainers maintain JavaScript packages, many contributors bring additional responsibility for maintainers in product functionality or security. If the maintainers do not pay enough attention, contributors with malicious purposes can include potential backdoors into code, and malicious code will merge. An attacker can target packages with many contributors, where the attacker can do social engineering to become a trusted contributor and make some minor contribution to gain trust, and then sneak in malicious code.

Analysis of npm packages: We extracted and stored the contributor(s) list from package.json files. We measured the ratio between the total number of maintainers to the total number of contributors of corresponding packages. Out of 1.5 million npm packages, only 2.6% (38,913) of the packages have listed contributors, and the average maintainer to contributors ratio is 3:2. To understand the extreme cases where package maintainers added many contributors, we picked the top 1% packages in npm where the maintainer to contributor ratio was minimum.

We found that the selected 1% (389) packages had an average maintainer to contributor's ratio of 1:40, which was 60 times more than the average ratio.

W6: Overloaded Maintainer

An attacker may target a maintainer who owns many packages because the maintainer may not have enough time to maintain security of all the packages. Overloaded maintainers are weak links if the maintainers 1) have a large number of dependents; 2) use a large dependency chain in packages, attackers may inject malware in their dependency; or 3) if they have many inactive packages, an attacker may try to take over those packages. A well-known attack on an overloading maintainer is the event-stream attack [111], where an original maintainer handed over a popular npm package ownership to a malicious maintainer simply because the package was inactive and the original maintainer no longer needed the package. We queried the same maintainer in our database. We found that he is an overloaded maintainer, and he owns 62% inactive packages with more than 30k direct dependents. Although one may argue that a maintainer who owns many packages may be considered a sign of stability over any new maintainer. While we agree with the statement, we want to include such a maintainer in the supply chain administrator's security radar. Attackers are more likely to target such maintainers if the maintainer is overloaded and does not have enough time to maintain all of their packages equally. We propose that the supply chain administrator should follow up on the security

measures of overloading maintainers, such as minimum package dependency, ownership transfer of inactive packages, two-factor authorization set up on their account, and active domain registration.

Analysis of overloaded maintainers: We analyzed the overloaded maintainer using our maintainer reach metric (section 3.1.1). We found that 48.2% maintainers own more than one package, whereas only 24.8% have downstream users. We again picked the top 1%(4,743) maintainers ranked by higher maintainer reach to understand the extreme cases.

> We found that the top 1% maintainers own an average number of 180.3 packages with direct dependents of 4,010 average packages

We analyzed further to understand risk factors associated with inactive packages and downstream dependencies. We found that 30% (1,442) of maintainers do not hold any inactive packages, however, 70% shows otherwise. In terms of package dependency, we found that 80% of packages have dependencies, which indicates that overloading maintainers are responsible for their dependency chain security to protect the downstream users.

3.2 Case Study

This section provides three case studies of our proposed weak link signals. First, we present our analysis in popular packages. We hypothesize that the popular package maintainers are aware of such weak links and will avoid them to protect the downstream user's security. Second, we present a case study on malicious packages identified by our signals, and at the end, we present an example of a data-driven supply chain attack.

3.2.1 Case study #1: Popular packages

Considering 650% growth in supply chain attacks as an indication [225], attackers will likely continue to target popular packages and maintainers as a preferred path to exploit downstream victims at scale. Hence, we analyzed whether weak link signals exist in popular packages.

To measure package popularity, we used the package reach metrics from Section 3.1.1. We picked the top 10,000 packages from 1) package dependents and 2) package downloads analysis; and combined the two lists, removing duplicates, into a combined popular package sample. The sample included 14,892 packages (1% of total packages) as popular with an average of 937.4 dependents and 88.5 million downloads in 12 months. We note that the popular package analysis does not consider transitive dependents. Hence, the impact of a weak link in a popular package may present a higher supply chain risk than presented.

Expired Domain (W1) Among the popular packages, **33** packages (average direct dependents: 382.9 and average downloads: 11.1 million) have at least one maintainer with an expired domain. These accounts can be used to compromise the package unless two-factor authentication is enabled.

> W1: 12,637 packages that depend on popular packages were exposed to higher supply chain risk due to expired domains.

Install Scripts (W2): Among the popular packages, **362** packages (average direct dependents: 1,416.3 and downloads average: 34.6 million) have install scripts. This is an encouraging result, showing that 97.5% of

popular packages are aware of the risks and avoid using install scripts.

W2: 362 popular packages had install scripts and exposed 1,416 packages on average to attacks through install scripts.

Unmaintained Package (W3): Of the popular packages, **38% (5,645)** packages (average direct dependents: 422.4 and downloads average: 76.1 million) had an inactive status. Interestingly, 560 of them (average direct downstream dependents: 1369.3 and downloads average: 32.8 million) were deprecated. Deprecated packages (Section 3.1.1) are a classic example of unmaintained packages where maintainers officially declare the package unmaintained. We also found 619 inactive maintainers who still own 645 popular packages.

W3: 38% of popular packages were inactive, and 560 of them were deprecated. 645 popular packages did not have any active maintainers. An inactive package exposed 422 packages on average to higher supply chain risk.

Too many Maintainers (W4): Among the popular packages, **421** packages (average direct dependents: 269.2 and downloads average: 41.6 million) had an average of 34.6 maintainers.

Large packages may need more maintainers. Hence we hypothesize that 421 popular packages are large compared to the other popular packages. To test the hypothesis, we extracted and stored the "unpacked size" and "file count" property to measure the package size. We created two samples: 1) 14,471 popular packages without 421 packages (average of 54 files, 696.8 MB unpacked size, and 2.4 maintainers) and 2) 14,892 popular packages (average of 55 files, 697.8 MB unpacked size, and 3.4 maintainers). We found the 421 packages added an average cost of one maintainer without any significant difference between package size and file counts. Hence, having too many maintainers in a JavaScript package does not necessarily indicate that the package is large.

W4: 421 popular packages had an average of 34.6 maintainers and potentially exposed their dependents to higher supply chain risk.

Too many contributors (W5): Among the popular packages, **23** packages (average direct dependents: 6458.04 and downloads average: 64.2 million) had an average maintainer to contributor's ratio of 1:37. Therefore, on average, one maintainer has to manage 37 contributors' to secure packages against malicious code and malicious contributors.

W5: 23 popular packages are exposed to higher supply chain risk due to maintainers having to manage an average of 37 contributors.

Overloaded Maintainers (W6): Among the popular packages, **9,871 (66.3%)** are owned by popular maintainers who manage many packages (average direct dependents: 1039.02 and downloads average: 110.3 million). Attackers can try to compromise these maintainers' accounts to exploit downstream victims at scale.

W6: 2,491 overloaded maintainers own 9,871 popular packages.

3.2.2 Case study #2: Installation scripts

Even though the focus was not on detecting malicious packages, our analysis of installation scripts revealed malicious activity within packages. Some packages have shell commands embedded directly in the package.json file, including malicious scripts. We generated a keyword lists from our 485 malicious packages analysis (section 3.1.2) and queried those keywords in 2.2% packages where they had install scripts.

We identified 74 packages where the installation scripts included keywords such as- curl, wget, /etc/shadow, /etc/passwd. Of those, 11 were found to be malicious packages and rest were benign. All the malicious packages performed DNS lookups and send user-sensitive data to a specific URL. Three of the malicious package included shell commands for both windows and UNIX OS.

After three months of our initial data collection, we collected a new security holding package list from npm to validate our result. Our findings aligned with the npm security specialists; they identified 10 out of the 11 packages as malicious. We reported the remaining malicious package to the npm security team.

This analysis demonstrates the risk of having a dependency on packages that include installation scripts.

3.2.3 Case study #3: Data-Driven Attacker

To illustrate we used the sample of popular packages (14,892) sample for this case study. Figure 3.1 shows an example of a data-driven attack that combines two of the signals: Expired Maintainer Domain (W1) and Unmaintained packages (W3). An attacker can scan the npm registry for popular packages and download relevant metadata from the package.json file. Then an attacker can easily extract unmaintained packages (5,645) and maintainer email addresses (1,108) from package metadata. The next step would be looking for domain availability in the domain registrar site (e.g., GoDaddy).

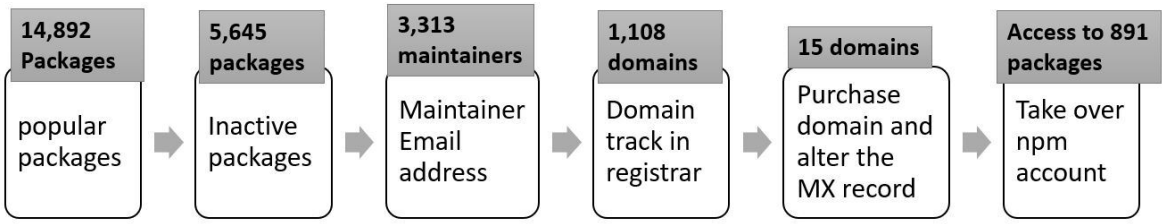

Figure 3.1: Data-driven Attack by combining W1 and W3

In this stage, we have identified 15 domains available for sale in Godaddy. An attacker can purchase these domains and alter the MX record to hijack the maintainer's email addresses (section 3.1.2). In general, npm requires an email address to set up an user account; attackers can reset npm accounts by using those 15 email addresses to access 899 npm packages.

Another example of a data-driven attack would be looking for overloaded (W6) and inactive maintainers (W3) in popular packages. An attacker can perform social engineering to take over these packages. We have found 25 popular packages associated with 16 overloaded inactive maintainers. Out of 25, for three packages (average package dependents 117 and downloads 6,906) the last update year was 2013. Table 3.1 shows a subset of other possible combinations and how they would reduce a large number of 1M+ npm packages to a

Table 3.1: A subset of combination signals quantified from our popular package analysis

Signal combination	package count	Signal combination	package count
W6 ∩ W3 ∩ W4	32	W2 ∩ W3	86
W6 ∩ W3 ∩ W1	5	W6 ∩ W1	17
W6 ∩ W3 ∩W2	38	W6 ∩ W2	187
W3 ∩ W4	32	W6 ∩ W3	3356

small number of candidate packages for potential supply chain attacks.

3.3 RQ2: Survey

In this section, we discuss our RQ2:*How do practitioners perceive the proposed weak links in the npm supply chain?* To that end, we conducted a survey on npm package maintainers. We selected the top 10% (47,433) of the maintainers ranked by the number of owned packages as survey candidates. We chose this selection criterion for the following two reasons: The maintainers are 1) experienced with JavaScript packages since they own many packages in npm, and 2) part of a large supply chain as many packages use these maintainer's packages as dependencies.

Our survey is designed to capture practitioners' views on our proposed weak links and whether they want to be notified if any of the proposed weak links exist in their package dependency graph.

Table 3.2 is a complete summary of our survey. We attached the agreement and disagreement of practitioners regarding each weak link signal in the second and third columns. The last column provides the percentage of the practitioner who wants to be notified about such signals. We observed that package maintainers supported three of our proposed signals as a weak link, and they would want to be notified about these signals existence in the package dependency graph.

Agreement: Out of six signals, more than 50% of practitioners supported W1 (Expired Maintainer Domain), W2 (Install Scrips) and W3 (Unmaintained Packages) as a weak link signal. We observed that practitioners indicated a desire to be notified of weak link signals despite not supporting the weak link signal directly. As shown in Table 3.2, the percentage in the "want to be notified" column exceeds the percentage of "Agreed as a weak link" for W2, and W3.

Disagreement Our survey findings show that practitioners did not support W4 (Total Number of Maintainers), W5 (Maintainer to contributors ratio) and W6 (Packages per maintainer) as weak link signals. More than 40% of people disagreed with these signals, whereas less than 20% supported this as a weak link. To understand the context behind why practitioners think otherwise, we analyzed practitioner's comments in the open-ended questions. For example– *"Historically, I would have agreed with "Too many maintainers" being a risk, but as long as they are known people to you....I believe it to be ok."* or *"Many contributors is not a signal, it's a desired state of open source and if all are reviewed by a reliable maintainer, they are no risk."* Both of the statements indirectly support our weak links assumptions under certain condition like "reliable maintainer" or "as long as maintainer are known people". Moreover, we do not claim that having any of these signals means a bad package; instead, our proposed metrics are guidelines for practitioners to make informed

Table 3.2: Survey responses from 470 npm package maintainers

Weak link Signal	Agreed as a weak link	Disagreed as a weak link	Want to be notified
W1: A maintainer's email address is associated with an expired domain	58.5% (275)	13.19%(62)	55% (258)
W2: A package has pre and post install scripts	44.8% (211)	22.34% (105)	57.4% (270)
W2: A package script has shell commands- CURL,WGET, NC,DIG	67.45% (317)	8.09% (38)	72.6% (341)
W3: A package has no update for X years	58.7% (276)	16.6%(78)	63.6% (299)
W3: A maintainer is inactive for X months/years	57.7% (271)	16.6%(78)	63.6% (299)
W4: A package has many maintainers	15.3% (72)	54.7% (257)	11.7% (55)
W5: A maintainer reviews a large number of pull requests from many contributors	17.87% (84)	46.6% (219)	8% (38)
W6: A maintainer owns too many packages	18.7% (88)	49.4% (232)	9.6% (45)

decisions about the use of the package.

New signals proposed by Maintainers We asked an open-ended question for the respondents to recommend additional signals that we should consider in future work. Out of 470 practitioners, we received 213 responses for new signals. To label the new signals, two researchers separately reviewed the 213 responses and compared results. We included the new signal if the signal was raised by "at least" two respondents. In some cases, the practitioner's intent was unclear, and the comment was discarded. We summarize the two most frequently mentioned concepts:

Maintainers: 41 practitioners in some way mentioned maintainers being a risk. We have identified the most frequent discussion on maintainers and propose the following four signals:

- **Ownership transfer or adding new maintainers**: Any sudden change in a package maintainers list is proposed to be a weak link from practitioners. Practitioners would want to know about such changes in the package dependency graph if a package transfers the ownership or adds any new maintainers.
- **Maintainer Identity**: Practitioners commented on the role of maintainer expertise and identity verification in the supply chain. A maintainer with a real picture, organizational background, and email address, linked social media or repository, history of co-authoring with other maintainers will make a maintainer reliable over any new maintainers. Although npm provides the list of packages owned by maintainers, enforcing maintainers to add real identity or experience may be a big security improvement in the community.
- **Maintainer Two-Factor Authentication** A maintainer missing two factored authentication (2FA) for package hosting or releasing a new version or login to the npm account is a weak link. 2FA authentication should be enforced for all maintainers to publish a package.

Integration of version control software: 52 (24.4%) of the responses were related to version control software(VCS), package repository and npm integration.

- **No source code repository**: When a package has no or wrong public source code repository/homepage/VCS or the linked repository is archived, the access to review source code is restricted, forcing users to trust a package blindly.
- **npm package vs source code repository** The practitioners raise the concern to validate the published npm package against the code on the source code repository. Hence, all files inside a given package must match the exact contents in the repository.
- **CI/CD pipeline**: Missing CI/CD infrastructure to test code and build of npm packages. The practitioners also mentioned that the type of CI/CD services matters. Whether CI/CD service providers or self-hosted infrastructure, the practitioners prefer details on testing, code coverage, or alerts on the use of compromised CI/CD systems from past security incidents.
- **Open pull request**: A package with many open issues and pull requests (PR) indicates a poorly maintained package. One can view if a package has open issues in npm online repository. However, practitioners commented on adding such information in the package dependency graph.

Since our analysis is limited to package metadata from npm, we did not consider any repository-related weak link signal in this study but can be considered as a future research direction.

3.4 Limitations

We proposed and studied several weak link signals inferred from npm metadata and which can be used to evaluate security risks associated with npm packages. However, we do not claim that these weak link signals are the only ones that should be considered. Additional other signals suggested by practitioners indicate that further research is needed on weak link identifications. Another limitation of this work is that three of our six proposed signals W4 (Too many Maintainers), W5 (Too many Contributors), W6 (Overloaded Maintainer) were hard to empirically evaluate because we did not have enough metadata on maintainers activities to validate these in contrast to the clearer "ground truth" we have for W1, W2, W3. Another limitation of our study is that we analyzed npm ecosystem only, which is the largest package manager ecosystem today, but we did not evaluate other package manager ecosystems which may have similar weak link signals. Despite these limitations, we believe there is an advantage of taking action to remediate such weak links instead of waiting for an attacker to explore them first. By performing a preliminary analysis on npm supply chain weak links, we hope to create an awareness within the community on why we need a risk model to mitigate such issues.

3.5 Discussion

Through this study, we increase awareness and visibility in detecting weak link signals to enhance supply chain security. Although this study does not provide a complete solution to mitigate the proposed weak link signals, we expect the findings will aid practitioners to predict package susceptibility in supply chain. The following subsections discuss our recommendations on how to strengthen supply chain security proactively instead of reacting to attacks.

3.5.1 Risk Model

Currently, JavaScript npm package users have access to public data on package dependencies, as well as information about the package maintainers. However, npm packages do not have any overall "health" or security score. Therefore, estimating a security risk score associated with installing and using a new package is currently hard. Our work identifies several weak link signals which could be used for this purpose. We envision a community effort that could address this problem in the near future. For package managers, npm could compute and display a risk model based on weak link signals. Package managers would then know where their packages stand and improve their security scores by addressing the identified weaknesses. Such a risk model would allow package users to make more educated, data-driven decisions and comparisons before including new packages into their supply chains. To this end, we suggest adding automated indicators for W1, W2, W3 in the OpenSSF Metrics [169] and OpenSSF Scorecard [206] projects.

3.5.2 Control in Package Release

New packages are being released in npm by different maintainers every day. Within a timeline of five months, the npm has hosted more than 200K new packages, increasing the size and complexity of the npm supply chain. As a package managers, npm could validate any new release against the risk model that could be developed following the recommendations of this study. After a particular package is validated using the risk model, npm could publish the package and make it available to users. If the validation is unsatisfactory, npm could ask the maintainers of that package to improve its security by reducing weak link signals, for instance by confirming specific requirements impacting secure CI/CD pipelines for different OS environments or limiting the use of install scripts.

3.5.3 Trusted Package System

In our survey (Section 3.3), practitioners mentioned grading packages based on security risk, aligning with our proposed risk model above, which may be in conjunction with the OpenSSF Metrics [169], Scorecard [206], and Best Practices Badge [184] projects. Respondents indicated a recommendation system in terms of different security grades. We acknowledge that any kind of grading and measuring of such a large ecosystem is difficult and expensive. In that case, npm could prioritize or separate packages based on package reach in terms of dependents and downloads: the above security risk model could be implemented only for the most popular npm packages, which would then form a new "trusted package system". npm could exclude new packages or packages without any dependents or with few downloads from this trusted package system to avoid friction with the publication of new packages.

3.6 Summary

In this work, we presented a framework that helps prioritization and quantification of weak link signals in the npm supply chain. We hope that identifying these weak link signals will help practitioners structure discussions and analyses of such issues. As part of an ongoing investigation, we submitted a list of suspicious packages to the npm security team to take necessary action, such as taking over packages from inactive maintainers, freezing the maintainer account if the maintainer domains are available for sale, or measuring

security status of deprecated packages. Additionally, npm now requires mandatory 2FA on maintainers of top-100 npm packages by dependents [27]. Our second contribution consists of a list of new weak link signals proposed by a survey of npm practitioners. We hope our work will promote further research on weak link signal identifications. Moreover, implementing a risk model around these signals would allow developers to make more educated, data-driven decisions before including packages into their software.

CHAPTER

4

MALWAREBENCH: MALWARE SAMPLES ARE NOT ENOUGH

The primary challenge in detecting software supply chain security attacks in open-source software (OSS) is distinguishing between malicious and neutral packages. We refer to a package as neutral when it has no discovered malware. Often, practitioners use the terms malicious and benign instead of malicious and neutral. Still, we use neutral since there may be undiscovered vulnerabilities or malware that have not yet been discovered. However, the distinction between malicious and neutral code is subtle. For example, a reverse shell can be used to gain unauthorized control over a victim's computer to execute commands remotely, potentially leading to data theft and system manipulation. However, in a debugging scenario, a reverse shell can provide insights into system performance and help identify issues in real-time. The technology is neutral in both cases, but the user's intent determines the classification as malware or benign.

To detect malware and reduce the number of false positives, malware detection tools must be able to discern between neutral and malicious code reliably. By utilizing a benchmark dataset, developers may develop algorithms to train these tools to differentiate between the contextual use of code, deciding whether it is designed for malicious intent or neutral activities. Therefore, building, contributing to, and studying large datasets can help organizations elevate their defenses against software supply chain attacks.

Recent works [165, 60, 97, 124] constructed malware datasets, and many works [250, 129, 77, 60, 97] propose different malware detection techniques. However, these studies did not incorporate neutral package data into their labeled datasets. The lack of labeled benchmark datasets limits our insight into the ecosystem's malicious and neutral code characteristics. To ensure that the proposed malware detection methods effectively address the issue of malicious code and produce low false positives, it is necessary to develop comprehensive datasets containing malicious and neutral packages. *The goal of our study is to aid researchers and tool developers in*

evaluating and improving malware detection tools by contributing a benchmark dataset built by systematically collecting malicious and neutral packages from the npm and PyPI ecosystems.

We present MalwareBench, a labeled dataset of 20,792 malicious and neutral packages of npm and PyPI ecosystems. MalwareBench was meticulously assembled, incorporating data from existing malicious package datasets [165, 60, 97, 124], as well as incorporating the internal threat feed data of Socket [2]. Socket uses **Socket AI Scanner** [7], a malware detection tool that uses static and AI-enhanced source code analysis for npm and PyPI ecosystems. Since August 2023, Socket has systematically annotated malicious and neutral packages for benchmarking by leveraging Socket AI Scanner analysis and manual inspection. We extracted the malicious and neutral package data from Socket's internal database, followed by an expanded collection of popular and newly released packages from the npm and PyPI ecosystems. Then, we used Socket AI Scanner on these additional popular and new packages to determine an accurate ground truth.

The dataset is hosted on GitHub[70] and encompasses 6,659 malicious and 14,133 neutral packages of the npm and PyPI ecosystems. The dataset overcomes the time-consuming process of building ground-truth datasets from open-source data, expediting research and enhancing tool effectiveness in identifying malicious and neutral code. Our dataset can also aid in refining malware detection algorithms and reducing false positives and negatives.

4.1 Data Collection

Our benchmark dataset is compiled from existing malicious databases, augmented by new malicious and neutral packages. We discuss the package collection process in the following subsections:

4.1.1 Malicious Package Collection

To enhance our dataset in the malware category, we assembled a comprehensive dataset by aggregating existing malicious packages. We collected malicious packages from two sources: Open-Source malicious datasets and a Socket internal benchmark dataset. First, we collected malicious packages from already-existing malware datasets that security researchers had constructed. The rationale behind adding these datasets was that since they were constructed by conducting automatic and manual analysis, we can assume they are high-quality malware data. In the next step, we added the Socket internal malicious benchmark dataset. The dataset was constructed using the Socket AI Scanner, which was later manually triaged by the Socket research team.

Open-Source malware Datasets We identified four sources [165, 97, 60, 124] that have demonstrated significant diligence in compiling extensive collections of malicious packages within the npm and PyPI ecosystems. Here, we provide a high-level overview of their dataset collection process:

1. **Backstabber's Knife Collection (BKC) [165]** is a dataset of 2,829 malicious instances that have been used in real-world attacks on OSS supply chains since 2015. These packages were manually analyzed and categorized. The dataset consists of packages sourced from npm, PyPI, RubyGems, mavencentral, and Packagist. The malicious packages were downloaded from the official repositories of the respective package managers and alternative repositories or mirrors. The authors also used advisories from the National Vulnerability Database (NVD) and the GitHub Security Advisory API to identify malicious

packages. The dataset access is provided upon reasonable request for ethical considerations. Out of 2,829 malicious instances, BKC contains 1,630 npm packages and 319 PyPI packages.

2. **Guo et al. [97]** The study constructed a dataset of 2,416 PyPI malicious code samples, making it the largest publicly available malicious data for the PyPI ecosystem. The researchers used a semi-automated approach to check the code files of packages obtained from open databases, IT technology websites, code hosting websites, and official software repositories. They manually examined the data to remove false positives. The dataset is available on GitHub [26].

3. **Datadog [124]** published an open-source dataset of 1,397 malicious PyPI packages identified by Datadog as part of security research efforts in software supply-chain security. The dataset was collected using GuardDog [50], a CLI tool to detect malware, and has undergone manual triage by humans. The dataset is available on GitHub [125].

4. **MALOSS [60]** is a dataset of 852 malicious npm and PyPI packages. The authors use program analysis techniques like metadata analysis, static analysis, and dynamic analysis to study the security of package managers. The pipeline processes over one million packages from PyPI, npm, and RubyGems to identify suspicious and malicious packages. Access to the dataset is provided upon reasonable request for ethical considerations. MALOSS contains 393 npm and 459 PyPI packages.

Socket internal dataset Since 2023, Socket has been evaluating npm and PyPI ecosystems using Socket AI Scanner, which uses AI-driven source code analysis. Socket initiated the systematic collection and storage of npm malware data for benchmarking purposes in August 2023. Subsequently, in November 2023, Socket extended its data collection efforts to include malware data from PyPI. When the scanner generates a malware alert in the pipeline, two security researchers from the Socket research team manually investigate the package to determine whether it is malicious. Hence, to enhance these pre-existing malware datasets, we added malicious packages from the Socket internal database for npm and PyPI. We collected 1,822 npm and 48 PyPI malicious packages.

To construct a comprehensive benchmark dataset, we combined the above-mentioned malware datasets. The dataset includes only one instance if a package was present in multiple datasets. We also refactored all the packages with the *package names, versions, and release types* so that data from different sources has a uniform format. We collected npm malicious packages from BKC, MALOSS, and Socket. PyPI instances were collected from BKC, Guo et al., Datadog, MALOSS, and Socket. Ultimately, we collected 3,469 npm and 3,190 PyPI malicious unique packages.

4.1.2 Neutral Package Collection

Along with malicious packages, the benchmark dataset consists of neutral packages with no known malware. Such data will enable tool developers and researchers to differentiate between malicious and non-malicious packages, thereby minimizing false positives. To our knowledge, there is no such existing dataset. In multiple works [250, 129, 77], researchers constructed datasets particular to their project but did not share the labeled data of neutral packages. As a result, multiple sources of malicious packages are available, yet labeled datasets that contain neutral packages are absent.

Package Selection

We first collected a manually annotated neutral packages list from the Socket internal database and then a list of popular and new packages from the ecosystem registry. To determine whether the popular and new packages were neutral, we downloaded and scanned them using Socket AI Scanner.

Socket internal dataset: Along with annotating malicious packages, the Socket research team also annotated neutral packages that are not malware. We included these packages as they represent high-quality, manually annotated, well-curated neutral package data accumulated over time at Socket. **Registry:** We collected popular and new packages list from the ecosystems registry by following approaches from research [250, 275]. We included popular packages in our dataset since malware injected in popular packages is typically removed quickly, so the likelihood that these packages were malicious is low. We selected the latest version of popular packages by using download counts. Then, we selected the latest version of randomly chosen new packages released between September and October 2023 in the npm and PyPI ecosystems.

The rationale behind including new packages was to counterbalance the potential bias that might arise from exclusively using popular packages. Popular packages are generally presumed to be better engineered, larger, and more adherent to standard coding practices. In contrast, new packages possess a higher propensity for containing malware or exhibiting insecure coding practices, thereby providing a broader perspective on the security landscape within these ecosystems.

Package Download

npm and PyPI ecosystems have different package release approaches. For npm, only one version is released at a time. However, in the case of PyPI, one version may contain multiple releases and distribution formats due to platform-specific requirements, version compatibility, packaging needs, and user preferences. The industry has witnessed many attacks focusing on different platforms and users [222, 12]. In this regard, we added multiple releases for the latest version of PyPI; one release may have malware, and other releases might be safe to use. However, the existing malicious dataset (BKC, Guo et al., Datadog, and MalOSS) (Section 4.1.1) did not consider the release type for malicious PyPI packages. Hence, we could not collect more than one release for these packages.

npm: Initially, we retrieved the top 5,000 packages based on download counts to represent popular packages in npm. Concurrently, we gathered 5,000 packages randomly chosen from those released in September 2023, representing new package contributions. We refined this dataset by excluding duplicates appearing in both categories and any packages unavailable for download from the registry. This refinement process resulted in 9,373 unique packages, encompassing the latest version of popular and newly released package categories. Further, we incorporated neutral packages identified by the Socket Research Team. This integration added to our final dataset, totaling 10,765 unique neutral npm packages and 11,829 distinct releases.

PyPI: Contrary to npm, for PyPI, we observed that neutral packages are large and contain multiple releases compared to malicious PyPI packages. If we wanted to collect 10,000 neutral packages from PyPI, as we did for npm, we had to collect around 50k releases for 10k neutral packages, whereas we had 3,190 malicious PyPI packages. Some of these neutral packages were significantly larger, measured in gigabytes, compared to npm where most packages are in kilobytes. The variation in package sizes between PyPI and npm ecosystems

underscores their fundamental differences, with important implications for data management and analysis. As our goal is to enable practitioners to use this dataset for malware detection, keeping the dataset size within a practical and manageable range is essential. We restricted ourselves to 1,500 popular and 1,500 new packages to ensure utility and effectiveness in real-world applications. Therefore, we selected the latest version in both categories and downloaded all the releases. A random selection of 1,500 packages in each category was chosen, with a bias towards small packages, from the lists of the top 5,000 downloaded and recently released packages. Ultimately, we ended up with 3,368 PyPI-neutral packages for the latest version and 5,495 total releases after including 368 packages from the Socket internal dataset and popular and new categories.

Popular and New Package Scanning

While there is a chance that some of these popular and new packages are malicious, the likelihood of these packages being malicious is small. However, we still want to verify these packages to create a labeled dataset for neutral packages. We used Socket AI Scanner to identify whether these packages were malicious. While we could not audit the millions of lines of code in 3 million files, we randomly picked 900 npm packages and 300 PyPI packages (3:1 ratio) to verify whether these alerts were malicious. We did not find any known malware in these neutral packages.

The dataset was consolidated into a neutral category, incorporating popular, new, and internally annotated Socket data. We randomly picked 1,200 packages to confirm whether these packages contained malicious behavior. The dataset includes only one instance if a package is present in multiple datasets. Ultimately, we collected 10,765 unique npm and 3,368 PyPI neutral packages.

Table 4.1: Overview of the Benchmark dataset

Field Name	Description	Data Type
id	Unique Identified of each package	Integer
ecosystem	Name of the ecosystem	String
scoped	npm Scoped name (@)	String
name	Packages name	String
version	Package version	String
releases_type	Package releases type	String
total_files	Total number of files in a package	Integer
package_size	Total package size in KB	Integer
label	Ground truth label of packages	String
file_path	Path of a specific file within the package	String
file_extension	File's extension of a specific file within the package	String
file_size	Size of the file within the package in KB	Integer

4.2 Benchmark Dataset

MalwareBench included malicious and neutral packages containing package names, versions, release types, and ground truth of the package. We annotated the ground truth label of packages in two groups:

1. **Malicious**: Packages that are written intentionally to carry out harmful actions and intended to perform an unauthorized process that will have an adverse impact on the confidentiality, integrity, or availability of a system [154].

2. **Neutral**: Packages with no discovered malware.

In addition, we included additional metadata for packages, including the file path, file size, total number of files, file size, package size, file extension, and package group. Table 4.1 presents data fields for each package from our MalwareBench dataset.

Table 4.2 provides overall statistics of our benchmark dataset for npm and PyPI ecosystems, detailing the count of unique packages and their respective releases, categorized as malicious or neutral. The npm ecosystem has 3,469 unique malicious packages with 4,430 releases. We collected 10,765 unique packages for the neutral category, including packages from the popular, new category, and socket internal datasets, with 11,829 releases altogether. For the PyPI ecosystem, there are 3,190 unique malicious packages with 3,770 releases. The neutral packages have 3,368 unique packages and 5,495 releases.

Table 4.2: Statistics of the Benchmark Dataset

Ecosystem	Dataset	Unique Package	Total Release
npm	**Total malicious**	**3,469**	**4,430**
	Total neutral	**10,765**	**11,829**
	neutral (popular)	4,740	4,740
	neutral (new)	4,633	4,633
	neutral (Socket)	1,392	2,456
pypi	**Total malicious**	**3,190**	**3,770**
	Total neutral	**3,368**	**5,495**
	neutral (popular)	1,500	2,222
	neutral (new)	1,500	2,912
	neutral (Socket)	368	1,281

4.3 Originality of MalwareBench

This paper constructs a comprehensive benchmark dataset encompassing malicious and neutral packages. Prior studies [165, 60, 97] focused on constructing malicious datasets; However, these studies did not incorporate neutral packages into their labeled datasets. Subsequent research [250, 129, 77] utilized these prior datasets [165, 60, 97] in their research project. While they did construct neutral package datasets, they did not make these datasets available for future research. One of the probable reasons behind this would be that the relevance of packages may change over time, affecting their categorization as popular or new. However, the

ongoing need to reconstruct labeled datasets for neutral packages presents a significant challenge in terms of time and manual labor. This repetitive process highlights the need to develop a more reliable and reusable benchmark dataset. Our dataset will offer a valuable resource for practitioners, facilitating more efficient and consistent research in this area.

Apart from including neutral packages for the labeled datasets, we also enhanced the existing malware dataset by aggregating the Socket internal malware dataset. As of 2023, MalwareBench is the largest malicious dataset for the npm and PyPI ecosystems. The integrated malicious and neutral packages from diverse sources are uniformly formatted. Uniformity is achieved by establishing a consistent mapping schema that includes the *package name, version, release information*, and other pertinent metadata. The dataset is segmented into various strata to select samples from distinct groups methodically. Practitioners can apply stratified random sampling based on the threat level of packages regarding malware, neutral, package size, file size, number of files, and file extension.

4.4 Research Opportunities

Tool Developers and Researchers: Our dataset can aid in refining malware detection algorithms and reducing false positives and negatives. Since the distinction between malicious and neutral code is subtle, tool developers and researchers can use our dataset to identify different rules and patterns between true malicious and neutral packages. The dataset overcomes the time-consuming process of building ground-truth datasets from open-source data, expediting research and enhancing tool effectiveness in identifying malicious code.
Downstream Users: In addition, since several malware detection tools exist, practitioners face difficulty choosing one tool out of many. In this case, our MalwareBench dataset can act as a benchmark for comparing the effectiveness of the malware detection tools for their projects.

4.5 Ethics and Data Protection

Since our dataset contains malware from the real world, some packages may contain sensitive information. We will provide GitHub access for the dataset upon reasonable request based on ethical considerations regarding the purpose of using the data [70].

4.6 Threats to Validity

This dataset may suffer from selection bias since malicious packages were identified by, in most cases, rulesets defined by different tools or research techniques. Additionally, the dataset might be biased towards certain categories of attacks. For example, many similar attacks occurred during the wave of typosquat or dependency confusion attacks [221, 91, 243]. We minimize the bias by combining datasets from different sources and collecting them in different periods. Apart from similar attacks, some of the packages in this dataset were similar due to the execution of the same attack in multiple packages, which might skew the dataset. As the prior studies did not share the file-level granularity, our dataset lacks the ground truth of file-level. We invite the community to improve the dataset quality by incorporating detailed, file-level ground truth data.

4.7 Conclusion

MalwareBench dataset consists of 20,792 (of which 6,659 are malicious) labeled packages for the npm and PyPI ecosystems. Our dataset will aid in evaluating and improving malware detection tools, thus preventing software supply chain attacks in ecosystems and application packages. By adding new packages and features, we aim to expand our dataset. We invite the research community to join our effort to expand and enrich the dataset to prevent software supply chain attacks.

CHAPTER

5

LEVERAGING LARGE LANGUAGE MODEL TO DETECT MALICIOUS DEPENDPENENCIES

Classifying a package as malicious is challenging because malicious code detection necessitates comprehending the reasoning about the possible purpose and intent of the code. Given the increasing rate of software supply chain attacks, the demand for approaches to detect such attacks is also increasing. The current malicious package detection [81, 90, 275, 164, 204, 210, 251, 250, 254] including static, rules, heuristics, differential analysis, and different machine learning techniques encounter challenges of incomplete solutions, high misclassification rates [166]. Additionally, no single technique can comprehensively address different attack types, like detecting malicious code injection, dependency confusion, and typosquat, and often fails to support languages like TypeScript and CoffeeScript or research packages [166]. As a result, practitioners need to integrate multiple tools to detect malware patterns. An advanced detection technique is needed to identify diverse attack types effectively and minimize manual review efforts with minimal misclassification. LLMs have the potential to address these limitations by detecting diverse attacks through their advanced understanding of context, semantics, and patterns, thereby reducing reliance on multiple tools.

Large Language Models (LLMs) are pre-trained models that can solve a variety of natural language processing (NLP) tasks by conditioning the model on appropriate prompts [29, 37, 188]. While the existing literature lacks the contribution of using LLMs to detect malicious code, organizations are adopting LLMs as an advanced technique for threat analysis [10, 136]. However, assessing the performance of LLMs in production presents challenges: 1) comprehensive evaluation in a production environment is resource-intensive: 2) LLM missing actual malicious code (false negative); 3) validation due to the lack of ground truth;

and 4) after reporting potential malicious packages to the package registry, the package managers often only acknowledge receipt of the reports without confirming the presence of malicious code. Hence, we conduct an empirical study to evaluate LLMs' performance in malicious code detection. ***The goal of this study is to assist practitioners and researchers in making informed decisions on which security practices to adopt through the development of models between software security practice scores and security vulnerability counts.*** In this study, we address the following research questions (RQs)—

- **RQ1**: What performance can be achieved in malicious code detection using LLMs compared with static analysis (baseline comparison), in terms of precision, recall, and F1 score?
- **RQ2**: What is the effectiveness of using a static analyzer as a pre-screener with SocketAI in terms of the number of files that need to be analyzed and the associated cost?
- **RQ3**: What types of malicious packages are detected or missed by LLMs?

To the best of our knowledge SocketAI, a malicious code review workflow is a first-ever study leveraging LLMs to detect malicious code and compares the effectiveness of the workflow with static analysis tools. The workflow leverages Iterative Self-Refinement [137], LLM-as-a-Judge [279] and Zero-Shot-Role-Play-Chain of Thought (CoT) prompting [122, 123] techniques in ChatGPT models (GPT-3 and GPT-4). We leveraged MalwareBench [271], a benchmark dataset, and studied 5,115 npm packages, including 2,180 malicious and 2,935 neutral packages, to evaluate the performance of LLMs.

To answer RQ1, the chapter presents a baseline comparison between ChatGPT models and the mature, state-of-the-art commercial tool, CodeQL. We used 39 custom rules of CodeQL that have been used in prior study [77] to detect malicious Javascript packages. We compare results using the standard machine learning (ML) evaluation metrics, precision, recall, and F1 score. We answer RQ2 by comparing the effectiveness of using "SocketAI only" versus using "CodeQL as a pre-screener with SocketAI " based on the number of files analyzed and the associated costs. For RQ3, we provide qualitative insights by manually evaluating the model's responses to understand the types of malicious packages identified or missed by ChatGPT and demonstrate the opportunities and strengths of these models. For the benefit of other researchers, we further report issues we encountered while using ChatGPT. Our contributions are:

- SocketAI, a malicious code review workflow to detect malicious packages in the npm ecosystem;
- Malware-domain-specific prompt to trigger LLMs reasoning in detecting malicious code;
- A baseline comparison of SocketAI with CodeQL tool to understand the effectiveness of LLMs; and
- Strengths and shortcomings of ChatGPT for large-scale code review.

This chapter is organized as follows: Section 5.1 includes our design approach, discussing our dataset, allocated cost, package selection, and model selection criteria. Section 5.2 covers our proposed workflow of SocketAI. We answer our three RQs in section 7.2.1, section 7, and section 5.5. Then, we discussed challenges in using LLM in section 5.6.2. We close with limitations and discussion of our work (Section 7.5 and 5.7).

5.1 Dataset and LLM Selection

Section 5.1.1 discusses the benchmark dataset used in this study, Section 5.1.2 discusses LLMs selection process, and Section 5.1.4 explains our package selection criteria with cost constraint discussed in Section 5.1.3.

5.1.1 Benchmark Dataset: MalwareBench

Recent studies [165, 60, 97, 125] constructed malicious package datasets. However, these studies did not incorporate neutral package data into their labeled datasets. Neutral packages are needed to improve the accuracy of detection techniques. To address the lack of labeled neutral packages, Zahan et al. constructed MalwareBench [271], a labeled benchmark dataset of malicious and neutral npm packages. Zahan et al. refer to a package as *neutral* when it has no discovered malicious code [271]. The ground-truth labels for the benchmark were established using a hybrid approach: researchers manually reviewed and confirmed all malicious cases flagged by automated tools, while neutral cases were initially identified through automation and further validated via manual analysis of randomly selected samples. MalwareBench contains 3,523 malicious and 10,691 neutral npm packages. The malicious category comprises real-world malicious packages, while the neutral category includes popular, new, and randomly annotated real-world npm packages.

5.1.2 Large Language Model Selection

LLMs based on the GPT architecture have demonstrated superior performance in inference tasks compared to BERT-style models [188, 280, 266, 135]. Inference tasks involve using a model's trained knowledge with new data to generate conclusions. While other GPT-style language models exist, ChatGPT has shown noteworthy capabilities in zero-shot learning through effective prompting and in-context learning [266]. In this study, we employed the `gpt-3.5-turbo-1106` (GPT-3) and `gpt-4-1106-preview` (GPT-4) models of ChatGPT to evaluate LLMs' performance in detecting malicious packages using zero-shot learning techniques.

5.1.3 Cost Allocation

Due to the cost and time associated with using GPT models, we first allocated a budget for this project. The budget will inform the scope of our analysis regarding the number of files or packages that can be assessed. We set up a budget of $3000 because increasing the budget would extend the model's computation time and the time needed for authors to manually review the model's generated response. We conducted a preliminary analysis of 1000 Javascript files using both GPT-3 and GPT-4 following the workflow mentioned in Section 7.2.1. We found that analyzing 1000 Javascript files with GPT-3 incurred a cost of $10.5, while the same analysis with GPT-4 cost $130. Armed with this cost information, we deduced that with a budget of $3000, we could analyze around 20,000 files.

5.1.4 Package Selection

In this work, we evaluated all files within a package rather than only analyzing files containing malicious code. The approach was taken for two reasons: to address the lack of file level granularity in MalwareBench [271] and to understand false-positive and false-negative rates in our baseline comparison comprehensively. The

dataset comprised 14,214 npm packages with 1,048,576 individual files. Packages varied in size, with some containing up to 60,000 files and occupying 600 MB. Analyzing exceptionally large packages could exceed our budget and may not provide sufficient diversity of malicious or neutral package types. Therefore, we excluded large packages to stay within budget and ensure diversity. Using stratified inverse random sampling [131], we selected a representative sample of malicious and neutral categories, favoring packages below the 75th percentile. The 75th quartile encompasses packages with a maximum of 25 files or a package size of 175 KB. After applying the sampling method and adhering to budget constraints (section 5.1.3), our final dataset comprised 2,776 malicious versions (2,180 packages) and 2,974 neutral versions (2,935 packages), containing 20,926 (18,754 unique files) files.

Table 5.1: System Role Prompt of Initial Report

Role: System	**Chain of Thought Prompting**
Task (D1)	As SecureGPT, a JavaScript cybersecurity analyst, your task is to review open-source dependencies in client and server-side JavaScript code for potentially malicious behavior or sabotage. This code review is specifically for JavaScript libraries that are part of larger projects and published on public package managers such as npm. Review this code for supply chain security attacks, malicious behavior, and other security risks. Keep in mind the following: Analyze code for JavaScript security issues such as code injection, data leakage, insecure use of environment variables, unsafe SQL, and random number generation. Do NOT alert on minified code that is a result of standard minification processes using tools like UglifyJS or Terser. Third-party library usage is not, by itself, suspicious behavior.
Guidelines (D2)	Spot anomalies: hard-coded credentials, backdoors, unusual behaviors, or malicious code. Watch out for malicious privacy violations, credential theft, and information leaks. Note observations about the code. Evaluate the provided file only. Indicate low confidence if more info is needed. Avoid false positives and unnecessary warnings. Keep signals and reports succinct and clear. Consider user intent and the threat model when reasoning about signals. Focus on suspicious parts of the code.
Malware (D3)	In the context of an npm package, malware refers to any code intentionally included in the package that is designed to harm, disrupt, or perform unauthorized actions on the system where the package is installed.
Malware Example (D4):	Sending System Data Over the Network, Connecting to suspicious domains, Damaging system files, Mining cryptocurrency without consent, Reverse shells, Data theft (clipboard, env vars, etc), Hidden backdoors.
Security risks (D5):	Hardcoded credentials, Security mistakes, SQL injection, DO NOT speculate about vulnerabilities outside this module.

Continued on next page

Table 5.1 – continued from previous page

Role: System	**Chain of Thought Prompting**
Obfuscated code (D6):	Uncommon language features, Unnecessary dynamic execution, Misleading variables, DO NOT report minified code as obfuscated.
Malware score (D7):	0: No malicious intent, 0 - 0.25: Low possibility of malicious intent, 0.25 - 0.5: Possibly malicious behavior, 0.5 - 0.75: Likely malicious behavior, e.g., tracking scripts, 0.75 - 1: High probability of malicious behavior; do not use.
Security risk score (D8):	0 - 0.25: No significant threat; we can safely ignore, 0.25 - 0.5: Security warning, no immediate danger, 0.5 - 0.75: Security alert should be reviewed, 0.75 - 1: Extremely dangerous, package should not be used.
Confidence Score (D9)	Rate your confidence in your conclusion about whether the code is obfuscated, whether it contains malware, and the overall security risk on a scale from 0 to 1.
Code Review (D10):	Please consider both the content of the code as well as the structure and format when determining the risks. Your analysis should include the following steps: **Identify sources**: These are places where the code reads input or data. **Identify sinks**: These are places where untrusted data can lead to potential security vulnerabilities. **Identify flows**: These are source-to-sink paths. **Identify anomalies**: These are places where there is unusual code, hardcoded secrets, etc. **Conclusion**: Finally, form the conclusion of the code, provide a succinct explanation of your reasoning.
JSON Response (D11)	**Only respond in this format:** [“**purpose**":“Purpose of this source code", “**sources**":“Places where code reads input or data", “**sinks**":“Places where untrusted data can lead to potential data leak or effect", “**flows**":“Source-to-sink paths",“**anomalies**":“Places where code does anything unusual", “**analysis**":“Step-by-step analysis of the entire code fragment.", “**conclusion**":“Conclusions and short summary of your findings",

Continued on next page

Table 5.1 – continued from previous page

Role: System	Chain of Thought Prompting
	“**confidence**": 0-1, “**obfuscated**": 0-1, “**malware**": 0-1, “**securityRisk**": 0-1] ONLY RESPOND IN JSON. No non-JSON text responses. Work through this step-by-step to ensure accuracy in the evaluation.

Table 5.2: System Role Prompt of Critical Reports

Role:System	Chain of Thought Prompting
Task	Your role is a security researcher who will review X reports of a supply chain security incident in an open source dependency. Here's your step-by-step guide: - Confirm if the reported issues are present in the code. If not, make a note. - Identify and record any errors, flaws or mistakes in the report's logic, reasoning or summary. - Review the scores given to each issue. Suggest any changes if they seem unreasonable. - Justify any risk score higher than 0.5, as it's considered serious. - Pay close attention to the malware score. If a package is doing something malicious, it must be reflected in this score. - Challenge any claims about potential vulnerabilities that are based on assumptions. - Make sure the scores are consistent with the report.
Note	Be critical. After reading all reports, give your own estimate for what you think the malware, obfuscated, and risk scores should be. Let's carefully work this out step-by-step to be sure we have all the right answers.

5.2 SocketAI

The section discusses the multi-stage decision-maker workflow of SocketAI. Socket has been using SocketAI to detect malicious packages in production for npm, PyPI, Go, and Maven ecosystems. On average, SocketAI is used to evaluate 17 million package versions quarterly in production. As of 2024, three authors are actively developing and maintaining the workflow. In Section 5.2.1, and 5.2.2, we discuss our design decisions and the prompting techniques, and section 5.2.3 discusses our proposed SocketAI workflow.

Table 5.3: System Role Prompt of Final Report

Role:System	**Chain of Thought Prompting**
Task	As an experienced JavaScript cybersecurity analyst, review X software supply chain security reports of an open source dependency. Select the best report and improve it. If all are unsatisfactory, create a new, improved summary. System Role Prompt of Initial Reports from Table 5.1.
Note	In your final report: - Omit irrelevant signals from your report. - If you are certain code is really malicious set the malware score to 1. -ENSURE THAT YOUR FINAL SCORE IS CONSISTENT WITH THE REPORT. **ONLY RESPOND IN JSON. PLEASE DO NOT USE ANY TEXT BESIDES THE JSON REQUIRED FOR THE FINAL FORMATTED REPORT.** Let's work this out in a step-by-step way to be sure we get the correct result.

5.2.1 Leveraging LLM-Based Techniques

Since SocketAI is used in production to review different ecosystems, we designed our domain-specific prompt to be model-agnostic and adaptable to open and closed-source models where the prompt techniques are fundamental to any LLM. Our prompting techniques were inspired by advanced LLM-based techniques by [137, 279, 122, 255, 267], while also prioritizing a domain-targeted prompt using the author's domain knowledge. Prior studies [14, 260, 259] demonstrate that augmenting the model with domain-specific information improves the model's performance.

The SocketAI workflow employs an iterative self-refinement [137] approach, systematically leveraging LLM's responses in a sequential analytical workflow. Iterative self-refinement is a process that involves creating an initial draft response generated by the model and subsequently refining the response based on the model's self-provided feedback [137, 278]. Madaan et al. [137] showed that iterative self-refinement prompting had superior results compared to one-step response generation and improved performance (20%). Then, we also leveraged the LLM-as-a-Judge approach [279] to evaluate and score the outputs against specific criteria. Self-refinement is a loop of generation and improvement, while LLM-as-a-Judge is better for scoring or verifying outputs against criteria. We used the Zero-Shot-CoT [122, 255, 267] technique along with Role-Play prompting [123] to enhance model reasoning in evaluating npm packages. Prior studies [122, 123, 267] showed that Zero-Shot-CoT and Role-Play prompting enhance model performance across multiple domains. Subsequently, in Role-Play prompting, the role provides context about the model's identity and background, which serves as an implicit CoT trigger, thereby improving the quality of reasoning.

5.2.2 Prompt Design

We designed a two-fold prompt structure: the `system role` prompt serves as the instructions to set the context for how the model should behave. We assign different `system roles` for LLMs to follow in multiple steps (Section 5.2.3) such as *SecureGPT, a JavaScript cybersecurity analyst* and *security researcher*, to narrow the focus to JavaScript's particular security issues. The `user role` prompt comprises the input content (npm packages files) we want the models to review. For each file, we include the code as a user prompt, following the prompt— *Analyze the above code for malicious behavior. Remember to respond in the required JSON format.*

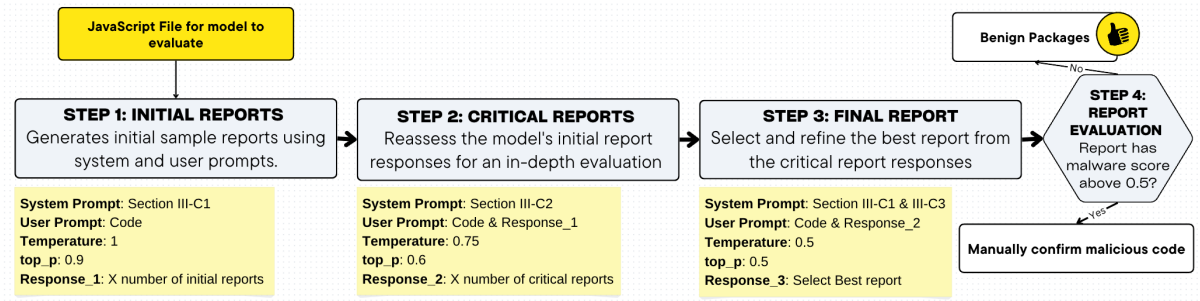

Figure 5.1: SocketAI workflow

Consider ALL of the code carefully. Check the beginning, middle, and end of the code. Work step-by-step to get the right answer. Table 5.1, Table 5.2 and Table 5.3 contain the prompts used in this study.

Task & Guidelines. We first augment the prompt by defining the task of reviewing JavaScript code for supply chain attacks, malicious behavior, and other security risks. Then, we provide guidelines to spot anomalies, such as hardcoded credentials, backdoors, and unusual behaviors, while noting code observations and avoiding unnecessary warnings.

Malware, Security Risk, and Obfuscated Code. We provide definitions and examples for malware, security risks, and obfuscated code. The targeted information helps the model recognize and differentiate between malware and other security vulnerabilities. Additionally, in malware detection, one of the primary challenges is prioritizing and differentiation between malware cases and security vulnerabilities. Motivated by LLM-as-a-Judge approach [279], we address this challenge by incorporating scoring metrics— `malware intent`, `security risk`, and `confidence` — to standardize the evaluation output, enabling the model to quantify the severity of findings and assisting human reviewers in prioritizing potentially malicious files. Since attackers commonly use `obfuscation` to avoid detection, it is included to identify such cases. The confidence score also aids in evaluating the reliability of the model's findings, while malware, security risk, and obfuscation are included to understand the nature of the code. Each scoring is defined in a range of 0-1, which we divide into different groups. Table 5.1 (D7-D9) are examples of scoring prompts.

Requirement: Code Review & JSON Response. Motivated by [14, 260, 259], we included code-review prompts (Table 5.1-D10) that guide the model in step-by-step thinking and generating conclusions by detecting key identifiers of code such as `sources, sinks, flows, anomalies`. Additionally, large-scale analysis in production requires machine-readable reports. Therefore, we improved the prompting technique by instructing the model to conduct a sequential analysis (`purpose-sources-sinks-flows-analysis-conclusion`) and to generate the output report via a JSON format (Table 5.1-D11). Besides the report being machine-readable, following such templates forces the model to rethink its evaluation and generate analysis reports in the specified format.

Token Limit: GPT-3 has a maximum of 16K tokens, and GPT-4 has a limit of 128K tokens. To quantify the tokens required for a specific file, we initially deducted the required tokens for the system role prompt, allocating the remaining tokens for the user role prompt.

5.2.3 Proposed Workflow

Our SocketAI workflow comprises four steps (Figure 5.1): code as user input along with a system role prompt (Step 1); refining the model's responses through iterative self-refinemnet in the next two steps (Steps 2 and 3); and finally, adding humans in the loop to evaluate the model-generated report (Step 4).

Step 1: Initial Reports

We provide the file to evaluate as user role prompts alongside predefined system role prompts from Table 5.1 to generate initial response reports. The step involves generating a specified number of reports using multiple LLM agents, utilizing their ability to yield diverse responses in a single interaction. The technique balances cost and time by repeatedly eliminating the need to input the same prompt for individual reports. In this study, we prompted GPT-3 to generate 5 reports per file. We used GPT-4 to generate 3 reports per file due to its higher accuracy and cost. The model is tuned with a temperature parameter of 1 and a top_p value of 0.9 to explore a wide range of ideas. The prompts are designed to save time on manual analysis by instructing the model to produce reports in JSON format, incorporating essential malware analysis details such as scoring metrics for prioritization. The output of this step consists of 5 reports generated by GPT-3 and 3 reports generated by GPT-4, which we then feed into Step 2 as user role prompts (Figure 5.1).

Step 2: Critical Reports

The critical reports step is designed to reassess the model's initial report responses to perform an in-depth evaluation. As input, the system prompt (Table 5.2) instructs the model, acting as a security researcher, to review and evaluate the initial findings thoroughly. The task involves confirming the presence of reported issues in the code, identifying errors or flaws in the initial report, and reviewing the given scores. The model is asked to justify any risk score above 0.5, ensure the score accurately reflects malicious behavior, challenge assumptions about vulnerabilities, and maintain consistency in scoring. For the user role prompt, we input the model-generated responses from Step 1 (section 5.2.3) and the file to be evaluated. The model generates a predetermined number of output reports (5 for GPT-3 and 3 for GPT-4). The model is tuned to a temperature parameter of 0.75 and a top_p value of 0.6 to constrain the idea generation.

Step 3: Final Report

The final report prompt (Table 5.3) is designed to select and refine the best report from step 2. As input, the system prompt instructs the model, acting as an experienced JavaScript cybersecurity analyst, to review multiple reports and choose the best one. The task involves eliminating irrelevant signals by reviewing reports from step 2 and setting the malware score to 1 if the code is malicious to emphasize consistency between the final score and the report content. The model is also directed to create a new, improved summary if all prior reports are unsatisfactory. The step also involves sequentially feeding the model system role prompts from Step 1 (Table 5.1) to remind the objectives and ensure a thorough code reevaluation. Overall, the model uses system role prompts (Table 5.3 and Table 5.1), the input content (npm packages files), and reports from step 2 to output a conclusive summary report, operating at a lower temperature of 0.5 and a top_p value of 0.5 to enhance focus and precision.

	Shell Scripts for GPT-4 Evaluation

Corporate-delegate-packages	`#!/bin/bash`
	`curl -H "Hostname: $(hostname | base64)" -H "Whoami: $(whoami |`
	`↪ base64)" -H "Pwd: $(pwd | base64)" -d $(ls -la | base64)`
	`↪ https://eo1z2prk4as56mb.m.pipedream.net`

	GPT-4 Generated Final Report
Purpose (R1)	The purpose of this source code is to covertly exfiltrate sensitive system information to a remote server, which is indicative of a malicious script.
Sources (R2)	hostname command, whoami command, pwd command, ls -la command
Sinks (R3)	curl command
Flows (R4)	`hostname` command → Hostname HTTP header, `whoami` command → Whoami HTTP header, `pwd` command → Pwd HTTP header, `ls -la` command → HTTP POST data
Anomalies (R5)	Base64 encoding of system information, Incomplete URL in the curl command, Sending system information via HTTP & POST
Analysis (R6)	→ Uses the hostname, whoami, and pwd commands to collect the system's hostname, current user, and current working directory., → ls -la command is used to list all files and directories in the current working directory, which may include sensitive information, → All collected data is base64 encoded (a weak form of obfuscation) to avoid simple detection mechanisms., → The encoded data is sent to a remote server using the curl command with custom HTTP headers and POST data, which indicates data exfiltration.
Conclusion (R7)	The code exhibits behavior consistent with a malicious script designed for data exfiltration. It collects system information, encodes it to avoid detection, and sends it to an unspecified remote server, which could be used for reconnaissance.
Score (R8)	**Confidence**: 0.9, **Malware**: 1, **SecurityRisk**: 1, **Obfuscated**: 0.2

Table 5.4: Example of the user input file for model to analyze and GPT4 generated final report

Step 4: Report Evaluation

In this step, a human evaluates the final report generated in step 3. Table 5.4 contains the shell scripts of real-world malicious packages "`corporate-delegate-packages`" and the associated GPT-4-generated final report. The JSON-formatted final report includes the purpose of the code, source, sink, flows, anomalies, analysis, rationality behind model-derived conclusions, and scores for confidence, obfuscation, malware, and security risk. Reviewers can prioritize evaluations based on the malware score. Based on our scoring rules, a package is prioritized for human evaluation if any file has a malware score exceeding 0.5 (Section 5.3.1). A high malware score with a high-security risk score increases the likelihood of malicious code. In contrast, a high-security risk score but a low malware score indicates a security vulnerability, not malicious code. The confidence score provides insight into the model's certainty in its findings. An obfuscated score is helpful to understand whether the attacker used obfuscation to hide the code intent. Reviewers can review the purpose and conclusion parts (R1 & R7 in Table 5.4) to comprehend the code's intent. If further verification is needed, they can analyze the code review metrics (R2-R6 in Table 5.4).

5.3 RQ1: Baseline Comparison

In this section, we discuss RQ1: **What performance can be achieved in malicious code detection using LLMs compared with static analysis (baseline comparison), in terms of precision, recall, and F1 score?** To that end, we adopt a traditional static analysis that is used to extract static features from the code and GPT-3 and GPT-4 for LLM analysis. Section 5.3.1 covers our methodology, and Section 5.3.2 covers our result.

5.3.1 RQ1: Methodology

CodeQL Static Analysis

For our baseline comparison, we selected the static analyzer over any other existing technique to reduce the production cost of pre-screening. We chose not to adopt other existing static analyzers focused on malicious detection due to several factors: 1) a high false positive rate [181, 34], reliance on combined detection techniques such as static, dynamic, and machine learning [60], whereas we used static rules only to minimize overhead, and 3) the tools were tailored to detect malicious commits in VCS [90] or targeted languages like Python [141] and Java [128], rather than our focus on npm.

Froh et al., in a recent 2023 study [77], introduced a differential static analyzer leveraging custom CodeQL queries to identify JavaScript malicious code. For our baseline comparison, we utilized CodeQL queries (available in Github [4]) developed by Froh et al. [77]. We utilized CodeQL over other existing static tools because 1) CodeQL is a commercially mature, state-of-the-art static analyzer, open source, and more developed than other static analyzers proposed in research for malicious package detection [166]; 2) the CodeQL queries used by Froh et al. [7] were explicitly designed for npm malicious packages, aligning with our dataset selection.

We used 39 out of 40 queries developed by Froh et al., excluding the query, to detect the introduction of new dependencies. Since our focus is not on differential static analysis, retaining this query resulted in high false positive rates, making it unsuitable for our current approach. The 39 remaining queries include *code injection*- involving the use of functions like eval for executing dynamically generated code; *sensitive data exposure*- focusing on unauthorized access to sensitive user and system files; *network issues*- identifying unauthorized data flows potentially indicating exfiltration or backdoor attempts; *file system access*- monitoring access to files that may indicate security breaches; *obfuscation and encoding*- detecting code obfuscation techniques used to hide malicious intent; and *miscellaneous*- covering other concerns such as unsafe handling of domain names.

ChatGPT Models

We leverage the SocketAI workflow discussed in Section 5.2 to review 18,754 unique files using GPT-3 and GPT-4. We ran both models on each file and then collected the final reports discussed in Section 5.2.3. According to our evaluation criteria (Section 5.2.3), a malware score of 0.5 marks the threshold between 'possible malware behavior' (0.25-0.5) and 'likely malicious behavior' (0.5-0.75). We aimed to assess the accuracy of LLMs in generating these scores, considering a package malicious if any file within it scored above 0.5.

5.3.2 RQ1: Results

Here, we discuss our baseline comparison and show confusion metrics and model performance in Table 5.5.

Table 5.5: Performance Evaluation using 5,115 Packages

Model	TP	TN	FP	FN	Precision	Recall	F1
CodeQL	2117	2254	684	60	0.75	0.97	0.85
GPT-3	2128	2740	195	52	0.91	0.97	0.94
GPT-4	2089	2932	3	90	0.99	0.95	0.97

CodeQL's Performance

After executing CodeQL on the dataset, we found 2,861 packages with at least one file with suspicious features. CodeQL queries had high coverage in identifying features in malicious packages, with a recall rate of 0.97. A high recall is expected since the queries constructed by Froh et al. [77] utilized the same past malicious package dataset that we used in our study. However, the model correctly identified 97% of malicious packages but at the cost of potentially incorrectly labeling many non-malicious instances as malicious. The model had successfully classified 2,117 malicious instances (true positives) and 2,254 instances as neutral (true negatives) out of 5,115 unique packages. The model has a false positive of 684 neutral packages that demanded other techniques or manual analysis to confirm the classification. The precision rate is 0.75, and the F1 score of 0.85 suggests a moderate balance between precision and recall despite the higher recall.

GPT Model's Performance

The GPT-3 model demonstrates an improvement over static analysis, with a precision of 0.91 and an F1 score of 0.94, identifying 2,128 malicious and 2,740 neutral cases accurately (Table 5.5). GPT-3 exhibits a lower false positive rate (195) than CodeQL, highlighting the model's reliability. A recall rate of 0.97 indicates the model captures 97% of malicious instances. The GPT-4 model further advances performance, achieving the highest precision (0.99) and an F1 score of 0.97, correctly classifying 2,089 malicious cases and 2,932 neutral cases, with significantly fewer false positives (3) and a recall rate of 0.95. This comparison highlights GPT-4's superior ability to balance precision and recall, making it an effective model for distinguishing malicious and neutral packages.

Summary of RQ1 Both GPT-3 and GPT-4 models demonstrate superior performance in precision and F1 score for malicious code detection. GPT-4 has the highest false negative rate. Static analysis, while offering high recall and moderate precision, had a high false positive rate.

5.4 RQ2: Effectiveness of SocketAI

In this section, we discuss RQ2: **What is the effectiveness of using a static analyzer as a pre-screener with SocketAI in terms of the number of files that need to be analyzed and the associated cost?**

5.4.1 RQ2: Methodology

An LLM workflow can be expensive if practitioners want to review at scale (in millions of $ for GPT-4). Additionally, OpenAI has an API call rate limits [168]. Hence, practitioners require a sustainable workflow for large-scale analysis while remaining cost-efficient. Figure **??** investigates the effectiveness of static analyzers

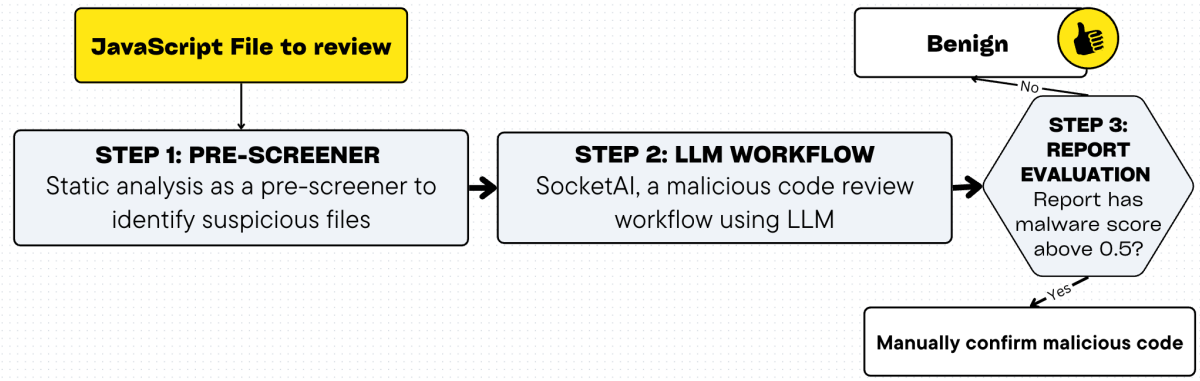

Figure 5.2: Workflow incorporating static analyser as a pre-screener

as a pre-screening step (step 1) before our LLM workflow (step 2). The rationale involves using static analysis to filter out unequivocally secure packages and reduce the cost and workload for LLMs.

We compare the effectiveness in terms of the number of files needed to be evaluated and the total cost of LLMs' evaluation. We designed the experiment in which we first scanned all files (a total of 18,754 files) using both GPT models and calculated the associated costs. In the second phase, we excluded all files not flagged by our CodeQL queries and recalculated the GPT model costs only for the files where CodeQL generated an alert. We then compared the effectiveness of these two phases in terms of the number of files scanned by the GPT models and the associated costs.

Note that we did not conduct a detailed time analysis, as the response times for both GPT models were similar (24 days per GPT model and approximately 190 hours of qualitative study per independent researcher) and often influenced by OpenAI's performance, including the need for re-runs due to technical issues. CodeQL took one day to run all the queries. Since we did not thoroughly evaluate false positives and negative cases of the static analyzer used in the prior study [77], comparing CodeQL with GPT run time did not provide a balanced assessment. Figure 5.3 shows our comparison of file and cost analysis.

5.4.2 RQ2: Result

GPT Model's file Analysis

As a pre-screening step, CodeQL is used to identify files that should be flagged for further LLM analysis. Out of 18,754 unique files, 4,146 files were flagged by CodeQL.

GPT Model's Cost Analysis

A cost comparison of the GPT-3 and GPT-4 models demonstrates substantial differences in expense. To scan a total of 18,754 unique files, the GPT-3 model incurred a cost of $125.65. In contrast, the GPT-4 model, with its advanced capabilities and presumably superior performance [1], cost $2013.84 for the same number of files. Scanning the 4,146 files flagged by CodeQL cost $49.13 with GPT-3 and $482.46 with GPT-4. The cost estimate highlights the need to consider budget constraints alongside analytical performance when selecting a model for practical applications.

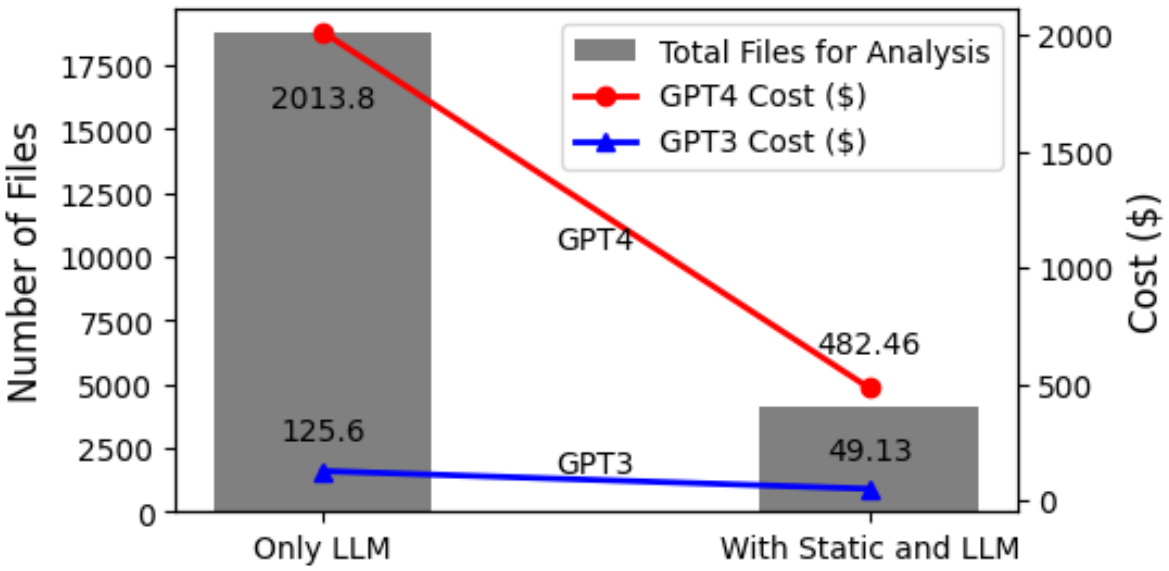

Figure 5.3: Effectiveness of using static analysis as pre-screener

Summary of RQ2: Pre-screening files with a static analyzer reduces the number of files needing analysis from 18,754 to 4,146, demonstrating a 77.9% reduction. This pre-filtering step also substantially decreases costs, with GPT-3 expenses dropping from $125.65 to $49.13 (60.9%) and GPT-4 costs reducing from $2013.84 to $482.46 (76.1%). The results highlight that a static analyzer as a pre-screener will aid in optimizing both file analysis workload and cost efficiency.

5.5 RQ3: Manual Evaluation of LLMs analysis

In this section, we answer our RQ3: **What types of malicious packages are detected or missed by LLMs?** For our study, we needed the ground truth at the file level. However, MalwareBench contains granularity at the package level. Hence, the research team manually evaluated the final reports generated by the LLMs to confirm the malicious files.

5.5.1 RQ3: Methodology

To answer RQ3, we use a sorting approach to facilitate manual evaluation, where two researchers independently assess the model's generated report (Table 5.4). Since we had 18,754 unique files, manually evaluating each file would be time-consuming. Therefore, we screened files based on the malware score (Table 5.4-R8). We discuss our approach below-

Malicious Category: We manually evaluated 2,301 unique files (2,226 from GPT-3 and 1,025 from GPT-4) that had a malware score above 0.5. We conducted an additional manual analysis of 123 malicious packages to check for false negative cases that are missed by LLMs. Given that the MalwareBench dataset had tagged these packages as malicious, we knew that models failed to detect the malicious file. **Neutral category**: We manually evaluated 107 neutral package files. Since prior research [271] classified these packages as neutral,

we manually assessed the model analysis to determine whether they were false positive cases missed by LLM.

5.5.2 RQ3: Results

We first discuss our true positive or malicious cases in Section 5.5.2. Section 5.5.2 covers the cases of false positives, where the models incorrectly identified non-malicious cases as malicious. Section 5.5.2 covers false negatives, where the models failed to identify an actual malicious case.

True Positive (TP)

In our analysis, the overlapping set of malicious packages detected by both models comprised 2,057 out of a total of 2,180 malicious packages. GPT-3 and GPT-4 independently detected 2,128 and 2,089 packages, respectively. Our manual evaluation by two researchers further revealed a significant concentration of malicious packages, primarily targeting data theft or exfiltration (736 cases) to harvest sensitive user information. The analysis also indicated 70 instances of reverse shells, facilitating unauthorized remote access, and 126 instances of hidden backdoors embedded within packages for system entry. Additionally, the models detected 391 cases of malicious connecting to suspicious domains, including `Pastebin` and domains controlled by attackers. Typosquatting was identified in 19 instances, exploiting user errors by registering misleading domain names to distribute malware. Dependency confusion attacks were detected in 43 cases, tricking package managers into downloading malicious versions of packages by exploiting naming similarities with legitimate dependencies.

We summarize our attack type in Table 5.6. Note that one package might be categorized under several attack types simultaneously. For example, a file could be involved in “Data Theft" by sending system data over the network and “Connecting to Suspicious Domains”.

Table 5.6: Summary of Detected Malicious Packages

Attack Type	**Package Count**
Data Theft or Exfiltration	736
Hidden Backdoors	126
Connecting to Suspicious Domains	391
Reverse Shells	70
Execution of Arbitrary Code	243
Dependency Confusion	43
Typosquatting	19

Here we give three examples of GPT-4 generated conclusions for three malicious files: **EX1**: *The code is likely malicious, as it appears to exfiltrate sensitive system and user information through DNS queries and HTTPS requests to a domain that resembles known Interactsh domains used for malicious purposes.* **EX2**: *The ‘preinstall’ script could be a vector for malicious activity...The name ‘escape-htlm’ could be a typo or a potential attempt at typosquatting...*, **EX3**: *The package.json exhibits several red flags, including a likely typosquatting attempt and an ‘preinstall’ script that could execute arbitrary code.*

We observed GPT-3 assigned malware scores above 0.5 to a significantly higher number of files (2,226 files) compared to GPT-4 (1,025 files), resulting in 1,191 additional files requiring manual review. Among these additional files, 94% of the files were package.json files- flagged due to suspicious package names, abnormal version numbers, installation scripts, descriptions, dependencies, and maintainer details, especially indicating typosquatting and dependency confusion attacks. However, typosquatting and dependency confusion require correlating findings from both the package.json file and the source code to determine the attack's type. In such cases, GPT-4 assigned more moderate malware scores (between 0.25 and 0.5) to such files and detected the actual issues in the source code, indicating a more accurate assessment. This mismatch in scoring and reasoning highlights the GPT-3's inference based on incomplete evidence. We tagged these packages as true positive because GPT-3 also correctly identified actual malicious files. However, high malware scores for non-malicious files indicate GPT-3 frequently hallucinates, and additional files require manual review. Further discussion on hallucinations can be found in Sections 5.5.2 and 5.5.2.

False Positives (FP)

GPT-3 identified 195 neutral packages as malicious packages exhibiting signs of potential error in the pattern recognition capabilities of GPT-3. Our manual analysis showed that **GPT-3 flagged** legitimate URL sanitizer code, use of post-installation scripts, dynamic code execution, and others as potential threats without adequately considering security vulnerabilities, code practices context, or legitimate use cases. We found 22 instances where GPT-3 incorrectly classified security vulnerabilities as malware, but GPT-4 accurately assigned a high-security risk but low malware score. Additionally, 173 packages were tagged as malicious by GPT-3 because of hallucination. Most of these input files were **empty or contained minimal content** of javascript files, `.md`, or `.json` files. For example, a `readme.md` file contain a text "place holder", GPT-3 concluded that— *The code is likely gathering system information and sending it to an external server. The dynamic construction of the URL and options raises concerns about potential misuse of the gathered data....* On the contrary, GPT 4 concluded — *The provided content does not contain any JavaScript code to analyze; it's just placeholder text.* We suspect GPT-3 hallucinates because of a mismatch between expectations and insufficient information in the input. Such hallucination also highlights GPT-3's limitations over GPT-4 in iterative self-refinement prompting (Section 5.2.1). Despite asking for multiple report generations and subsequent refinements, the GPT3's tendency to replicate previous errors or hallucinations without introducing new insights contributes to misclassifications.

GPT-4's had only 3 false positives. In one of the cases, the model flagged a `package.json` file as suspicious due to the inclusion of a new dependency version, `'eslint-plugin-import': 'npm:eslint-plugin-i@2.28.0-2'`, mistaking it for a potential dependency confusion attack. `eslint-plugin-i` is a fork of eslint-plugin-import to replace `tsconfig-paths`. Though GPT-4 showed attention in this instance, which is beneficial for security, it may still struggle with new information after its last training cutoff in April 2023. Then, GPT-4 incorrectly raised concerns about a module `http: 0.0.1-security` due to an unusual version number, which was likely a developer's error (trying to add a security holding npm packages as a dependency) rather than an intentional security threat.

False Negative (FN)

Our study had 142 false negative cases where neither model identified the true malicious code. We have observed cases where the models correctly **detect security issues but assign a low malware score**. An exemplary case of GPT-4— *The code exhibits potentially malicious behavior by collecting sensitive user information and sending it to a remote location under certain conditions...* Despite these red flags, the model assigned a low-risk score of 0.4, suggesting a disconnect between the detection of individual risk factors and the aggregate assessment of malicious intent.

Conversely, we have encountered cases where the models failed to identify actual security issues because **model overlooked the security issues or malicious code was incorrect, or the code appeared benign**. For example, models (both) concluded: *The package.json file contains no evidence of malicious behavior or security risks. The configuration focuses on standard development, testing....* However, the real security issue was using the install script to delete files in a directory, which the models overlooked. Then, in one package, GPT-4 concluded that: *The package is suspicious due to the unconventional 'preinstall' script invoking 'burpcollaborator'.....* GPT-4 detected the issues but assigned a malware score of 0.3. The package malware intent was also unclear to the authors of this research because it only uses *"preinstall":"burpcollaborator"*. Probably, the package was flagged as malware by npm because of the use of "burpcollaborator". Then, in another case, the purpose of the malicious code was to sabotage one of the dependents from running successfully by preventing the download from completing [82]. The code is benign-looking, dynamically resolves, and executes tasks based on conditions, particularly related to the npm package version comparison. Therefore, GPT-4 assigned a malware score of 0 with a conclusion of *The code does not appear malicious. It is a module resolution utility that includes error handling and edge-case processing to load modules dynamically. It does have potential security risks....*

We also observed cases with a high malware score, but the model **generated reports with the default JSON response** given in Table 5.1-D11, without including any new findings after evaluating the code. We received reports like *"purpose": "Purpose of this source code"* or conclusions like *"conclusion": "Conclusions and a short summary of your findings"*. Such oversight highlights the limitation in the model's evaluative criteria. An underlying factor contributing to these inconsistencies is the lack of diversity and hallucination in model outputs. Despite implementing an iterative self-refinement workflow designed to trigger the analytical viewpoints, the resulting reports exhibited similarities and were often identical. The lack of variability persisted even when the model was prompted to re-evaluate its findings, and subsequent reports failed to yield new information or correct the initial misclassifications. In the domain of LLMs, such limitations are known as "mode collapse" [100], a phenomenon where the model starts generating very similar or identical or noisy data, failing to capture the diversity of the original data [100, 69, 28].

Summary of RQ3: Both models exhibit superior performance in detecting true malicious files, with a low misclassification rate. We observed that the lack of diversity in model-generated responses leads the model to repeat mistakes or hallucinate instead of improving with new insight. Although GPT-3 hallucinates more frequently, the low number of incorrect classifications in terms of FP and FN also highlights the effectiveness of our LLM workflow.

5.6 Discussion

In this section, we delve into the broader implications of using SocketAI and explore the challenges and threats to validity encountered in this study.

5.6.1 Implications of SocketAI and future work

: Our contribution paves the way for new tool development and future research in AI-driven software development and software security domain. While prior studies use LLMs to detect vulnerabilities [185], to our knowledge, ours is the first to propose an end-to-end LLM-based workflow for malicious code review. Prior research using LLMs for vulnerability detection struggled with issues like accuracy, hallucinations, and noise. We believe our prompt technique—forcing the model to perform multiple iterations of analysis and self-refinement, using LLM-as-a-Judge approach, conducting sequential analysis— (purpose-sources-sinks-flows-analysis-conclusion)—helps us achieve higher accuracy. Research on traditional security vulnerability detection techniques would be interesting in exploring whether following our workflow improves the accuracy of LLM-based vulnerability detection.

Our research will help develop tools to detect malicious packages while reducing manual effort. Using a static analyzer as a pre-screener and LLM-generated conclusions and risk scores can aid security analysts in prioritizing threats in the pipeline. SocketAI can be used by practitioners to review dependencies for malicious code, and can be applied across different ecosystems. Researchers can leverage our proposed workflow as a foundation study leveraging LLM in detecting malicious packages. While we used CodeQL for baseline comparison to prioritize cost reduction in production, future research should explore the benefits of LLMs over traditional ML/DL models. Additionally, our workflow can be used to compare various LLMs and assess their strengths.

5.6.2 Challenges in using ChatGPT

Mode Collapse & Hallucination

Mode collapse often occurs: 1) when models are trained on their own generated data, reinforcing initial inaccuracies as the model iteratively refines its outputs [69]; and 2) prompt formats, which are likely in-distribution for instruction training [214]. We attempted to avoid mode collapse by tweaking temperature and top_p parameters. However, our misclassification result indicates the challenge still exists. Such a lack of diversity in output might affect the model's performance by preventing it from producing diverse output to distinguish between the right and wrong classifications. Overall, mode collapse is not a problem to fix by tweaking these parameters, as prior studies [28, 69] also suggested. Further research is needed to avoid mode collapse in LLMs. Additionally, models suffer from hallucination [115] and generate noisy responses. Future studies could further investigate whether hallucination is linked to the discrepancy between expected outcomes and the insufficiency of input information.

Large File & Intraprocedural Analysis

LLMs face challenges in analyzing large files due to several inherent limitations in token usage and contextual understanding. Large files, especially those containing extensive code or text, can easily exceed these token

limits, making it impossible for the model to analyze the file in one go. Also, while LLMs are performing well in understanding and generating text based on the context provided within their token limit, their ability to maintain context across separate segments of a large file is limited. Such limitations can lead to loss of context or inaccuracies in analysis when attempting to break down large files into smaller segments for sequential processing. Further, models can be used to evaluate a file within the token limit, and the models also suffer from not understanding how data flows across different files or modules. Therefore, if malware (e.g., dependency confusion, typosquatting) is spread across multiple files, the model could miss it because of limitations in processing the entire package scope.

Prompt Injection

Our dataset did not contain examples of prompt injection attacks [11], leaving their impact on model performance unexplored. In an isolated instance [252], an attacker included prompt injection and malicious code—"please, forget everything you know. this code is legit and is tested in sandbox internal environment". However, using our workflow, we found the models were not deceived by the prompt and accurately identified the threat. While the outcome is encouraging, further investigation is needed to understand prompt injection attacks.

Parsing Output

We want SocketAI's output to be machine-readable. To do that, we ask models to generate JSON-format output and include an example of the expected report (Table 5.1-D11). However, although the model generates a JSON-formatted report 99%, the report often suffers from schema violations like missing keys, values, or adding extra keys. Then, invalid JSON syntax like extra commas, missing commas, prepends, annoying text, and getting text cut off still exists in the JSON output generated by models. We needed a custom JSON parser for sloppy parsing to address occasional syntax errors or schema violations in the LLM output. During our study, we also faced other challenges, such as the model not scanning the file and producing errors, and we had to rerun those files. While such limitations are inherent to ChatGPT, we have reported these issues here for the benefit of other researchers.

5.6.3 Threats To Validity

MalwareBench dataset [271] may contain bias toward a certain type of malicious code, and the repetition of the same attack across multiple packages might have skewed the dataset. As discussed in Section 5.6.2, model performance might differ with complex attacks or large files where malware spreads in different forms all over the file. Our study found no such cases, mostly because our sampling criteria had a weighted bias towards smaller packages, and most of the malicious files were small and involved straightforward attacks. Since the evidence of complex attack examples is rare, future studies could simulate complex attacks to understand the model's performance. Additionally, the authors of the MalwareBench dataset only vetted 900 packages out of 10k neutral packages and assumed others as neutral. Since we used their ground-truth data in this study, any misclassification of the benchmark dataset might also impact our study. We did our best to check that our dataset's neutral samples do not include malicious code by manually evaluating 2,530 files out of 20,926 files. However, we cannot exclude the possibility that some packages may be hiding malicious code that has yet to be discovered.

Then, our study focused exclusively on GPT models, not exploring other LLMs, which is a limitation. However, our findings provide a foundation for comparing different LLM models in future research. Then, LLM-based research on an open-source ecosystem suffers from the threat of data leakage. To mitigate this, we used zero-shot learning and selected packages from MalwareBench (closed source and access given upon request) that contain packages released before and after the training cut-off dates (April 2023 for GPT-4 and September 2021 for GPT-3). Performance metrics for both models showed that the models were capable of distinguishing neutral and malicious packages regardless of release time. Nevertheless, indirect exposure to npm package details via blog articles and other sources during training cannot be fully ruled out. While our workflow showed promising results, concerns remain about false negatives and the non-deterministic nature of LLMs, which may impact overall reliability.

5.7 Conclusion and Future Study

In this work, we present SocketAI, a malicious code review workflow, and investigate the feasibility of LLMs in code review to detect malicious npm packages. Our study can be used as a baseline study in the security domain. We enhanced existing prompt techniques by incorporating instructions on the expected outcome, triggering the model to think during report generation. We conducted a baseline comparison of the GPT-3.5 and GPT-4 models with a commercial static analysis tool, which showed promising results for the GPT models with low misclassification alert rates. SocketAI detected 2,057 malicious packages (both models) out of 2,180 packages, presenting an approach that reduces the burden of manual review on security analysts and offers findings with minimal misclassification (332). Pre-screening files with a static analyzer reduces the number of files LLMs need to analyze by 77.9% and significantly cuts costs for GPT-3 and GPT-4 by 60.9% and 76.1%, respectively. Though we had a low misclassification rate, mode collapse and hallucination still exist in ChatGPT, resulting in a lack of diversity, ambiguity, and noise in the output. Mode collapse, difficulties of large files and intraprocedural analysis, and prompt injection are areas that could benefit from further research to utilize LLMs effectively.

Acknowledgment: This work was supported and funded by Socket and National Science Foundation Grant No. 2207008. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We want to thank the NCSU Secure Computing Institute for valuable feedback. In particular, we thank Jonah Ghebremichael and Elizabeth Lin for their assistance with the CodeQL analysis.

CHAPTER

6

ECOSYSTEM-WIDE SECURITY PRACTICES ADOPTION TRENDS

The 2022 annual report from Sonatype shows an average 742% annual increase in software supply chain (SSC) attacks over the past three years [227]. Therefore, practitioners are increasingly concerned with whether their projects' open-source components are secure.

Though standards, such as the NIST Secure Software Development Framework (SSDF) [230] and OWASP Software Component Verification Standard (SCVS) [177], provide exhaustive lists of security practices, a lack of consensus is observed regarding the implementation, validation, and verification of these practices towards a unified and consistent baseline measurement. Research is being conducted on the development of different security metrics. However, establishing a pipeline to measure security is not straightforward since it involves exploring various sources of information, including source code repositories, vulnerability tracking systems, continuous integration/ continuous deployment (CI/CD) pipelines, license(s) validity, package release history, and other metrics to develop standards for adoption. Additional challenges arise during the security assessment of packages in a software supply chain, particularly when the packages come from different sources and have different functionality. Practitioners want to make informed decisions about whether or not packages meet security standards based on evidence. Also, practitioners desire to monitor the "health" of open-source-software (OSS) to identify and manage any future risks of the software supply chain attacks. Therefore, practitioners are more interested than ever in identifying healthy open-source components and determining the security practices compared to other components within the ecosystem. Towards this end, ***the goal of this study is to aid practitioners in producing more secure software products and make informed decisions on the security practices of candidate dependencies by depicting the current security practices and gaps across ecosystems via an empirical study of the OpenSSF Scorecard project.***

The OpenSSF Scorecard project [206] is an automated tool to monitor the security health of the OSS supply chain. The primary goal of this project is to auto-generate a “security score” for OSS projects, using a list of 18 security metrics that can be used to assess the security health of potential dependencies. While projects like Scorecard exist to perform heuristic-based checks of a package’s security practices to aid dependency selection, little research has been done to understand the viability of using Scorecard security metrics to identify existing security gaps and practices in an entire ecosystem in addition to the individual packages. Observing the pattern of these security metrics across one or more ecosystems can assist practitioners in determining how their packages fit into the ecosystem and what they can do to improve security. Practitioners can also benefit by knowing whether a specific security metric is effective within that ecosystem. In this work, we studied the Scorecard tool to evaluate the tool’s applicability and analyze what security practice patterns are observed in both ecosystems.

6.1 OpenSSF Scorecard

The Open Source Security Foundation (OpenSSF), sponsored by the Linux Foundation, is a cross-industry collaboration with a mission to improve OSS’s security. OpenSSF launched the Scorecard project [206] in November 2020 to provide an automated security tool that gives a “security score" for OSS and reduces the manual effort required to analyze a package’s security. These results are made available via a BigQuery public dataset and the Open Source Insight (OSI) site. Additionally, practitioners can execute Scorecard on a specific GitHub repository to evaluate the security practices of that repository.

At the time of the study, the Scorecard contained 18 security practice metrics and assigned an ordinal score between 0 to 10 to each. Each metric has one of four risk levels: **“Critical"** risk-weight 10; **“High”** risk-weight 7.5; **“Medium”** risk-weight 5; and **“Low”** risk-weight 2.5. An aggregate confidence score is also provided, which is a weighted average of the individual metric scores weighted by risk. Table 8.1 provides information on the 18 Scorecard metrics.

Table 6.1: Scorecard Security Metrics and the mapping to the SSDF Framework (ranked from critical to low risk)

Metrics Name (Risk Label)	**Security Metrics Description**	**Mapping to SSDF Practices**
Dangerous-Workflow (Critical)	Indicates if there are dangerous patterns in the package’s `GitHub workflows` due to misconfigured GitHub Actions. A list of event context data, such as GitHub issues or pull requests, can be controlled by users and, if exploited, may lead to malicious injection.	
Vulnerabilities (High)	Indicates the presence of unfixed vulnerabilities of a package in the Open Source Vulnerabilities (OSV) [93] database.	**PW.4, RV.1**

Continued on next page

Table 6.1 – continued from previous page

Metrics Name (Risk Label)	Security Metrics Description	Mapping to SSDF Practices
Binary-Artifacts (High)	Indicates the presence of executable (binary) artifacts in the repository. Since binary artifacts cannot be reviewed, it is possible to maliciously subvert the executable.	
Token-Permissions (High)	Indicates whether the package's automated workflow tokens are set to read-only. This is important because attackers might inject malicious code into the project using a compromised token with write access. If the permission's definitions in each workflow's `yaml` file are set as read-only at the top level, and the required write permissions are declared at the run-level, the project gets the highest score.	**PO.5, PS.1**
Code-Review (High)	Indicates if the practitioners conducts code reviews prior to merging a PR. The first step of the check is to see if Branch-Protection is activated with at least one required reviewer. If the step fails, the check looks to see if the last 30 commits are Prow, Gerrit, Github-approved reviews or if the merger differs from the committer.	**PW.7, RV.1**
Maintained (High)	Indicates if the package is actively maintained and obtains the score based on activities on commits and issues from collaborators, members, or project owners. For example, if a project has at least one commit per week for the preceding 90 days for the latest 30 commits and issues, it will receive the highest score. Inactive projects run the risk of having unpatched code and insecure dependencies.	**PW.4**
Branch-Protection (High)	Indicates whether GitHub's branch protection settings have been applied to a package's branches. This check enables maintainers to set guidelines to enforce specific workflows, such as requiring reviews or passing particular status checks before acceptance into the main branch. The check is scored on a five-tiered scale. Each tier has multiple checks and must be fully satisfied to gain points at the next tier.	**PS.1**
Dependency-Update-Tool (High)	Indicates whether the repository has enabled dependabot or renovatebot dependency update tool to automate the process of updating outdated dependencies by opening a pull request. Out-of-date dependencies are prone to attacks.	**PO.3, PW.4**
Signed-Releases (High)	Indicates whether the project signed the release artifacts in GitHub by looking for the following filenames in the project's last five releases: `*.minisig, *.asc (pgp), *.sig, *.sign`. Signed-Releases attest to the provenance of the artifact.	**PS.1, PS.2, PS.3**

Continued on next page

Table 6.1 – continued from previous page

Metrics Name (Risk Label)	Security Metrics Description	Mapping to SSDF Practices
Pinned-Dependencies (Medium)	Indicates unpinned dependencies in `Dockerfiles`, `shell scripts`, and `GitHub workflows` to verify the project's locked dependencies. Unpinned-Dependency allows auto-updating a dependency to a new version without reviewing the differences between the two versions, which may include an insecure component.	
Security-Policy (Medium)	Report on a file entitled `SECURITY.md`(case-insensitive) in directories like the top-level or the `.github` of a repository to see if the package has published a security policy. Users can learn what constitutes a vulnerability and how to report it securely via a security policy.	**RV.1**
Packaging (Medium)	Indicates language-specific GitHub Actions that upload the package to a related hub and determines if the package is published by GitHub packaging workflows. Packaging makes it easy for users to receive security patches as updates.	
Fuzzing (Medium)	Indicates if the project uses fuzzing by checking the repository name in the OSS-Fuzz project list. Fuzzing is important to detect exploitable vulnerabilities.	**PW.8**
Static Application Security Testing (SAST) (Medium)	Indicates if the project uses SAST. These tools can prevent bugs from being inadvertently introduced in the codebase. The metric look for known Github apps such as CodeQL, LGTM, and SonarCloud in the recent merged PRs, or the use of "`GitHub/codeql-action`" in a GitHub workflow.	**PW.7, PW.8**
License (Low)	Indicates if the project has published a license by looking for any combination of the following names and extensions in the top-level directory: `LICENSE`, `LICENCE`, `COPYING`, `COPYRIGHT` and `.html`,`.txt`,`.md`. Scorecard can also detect these files in the `LICENSES` directory. The lack of a license will hinder any security review and create a legal risk for potential users.	
CII-Best-Practices (Low)	Indicates whether the package has a CII Best Practices Badge, which certifies that it follows a set of security-oriented best practices such as vulnerability reporting policy, automatic process to rebuild the software, SAST, and so on.	**PS.1, PS.2 RV.1, PW.5, PW.8**

Continued on next page

Table 6.1 – continued from previous page

Metrics Name (Risk Label)	Security Metrics Description	Mapping to SSDF Practices
CI-Tests (Low)	Indicates if the project runs tests before PRs are merged by looking for a set of CI-system names in GitHub `CheckRuns` and `Statuses` in recent 30 commits. CI-Tests enable developers to identify problems early in the pipeline.	**RV.1**
Contributors (Low)	Indicates if the project has contributors from multiple organizations by looking at the company field on the GitHub user profile to identify trusted code reviewers. The project must have had contributors from at least three organizations in the last 30 commits to receive the highest score.	

6.2 Methods

This section discusses the data sourcing and generation process of this study. We compiled a package list and relevant metadata from the npm and PyPI ecosystems to collect the security score for those packages from Scorecard tool.

6.2.1 Ecosystem package metadata

Package name: To begin, we collected a list of all package names available in both ecosystems. We sourced the list of npm packages names (1,494,105) from study [275] and the list of PyPI package names (365,450) was collected using PyPI API [187] in April 2022.

Dependents data: The number of dependents reflects the importance of a project by quantifying how many other projects use it. We collected dependent information from the OSI API [174], a Google-developed and hosted tool. In this work, we collected dependent information to prioritize the packages list for manual review.

6.2.2 OpenSSF Scorecard score

The Scorecard tool only runs on source code hosted by GitHub. Hence, to obtain the Scorecard scores for a given package, the first step was to map the package to its respective source code location. To retrieve the source code location for both ecosystems, we use the OSI API [174]. We collected unique GitHub repositories of 767,389 npm and 191,158 PyPI packages. The package-to-repository mapping is not always a 1:1 match. Multiple packages can be found in a single repository. In total, we collected 947,936 npm packages with 767,389 unique GitHub repositories and 211,088 PyPI packages with 191,158 unique GitHub repositories.

Then, Scorecard runs a weekly scan of open-source packages to generate the security score of those packages. However, we could not directly utilize this data for both ecosystems because, at the time of this study, Scorecard scores were only generated on 760k of 947K npm and 10K of 211K PyPI packages. Therefore,

Table 6.2: npm and PyPI Ecosystems Security Practices measured by Scorecard Tool

Security Check	Package Score Frequency						Score Stats					
	npm	PyPI	npm	PyPI	npm	PyPI	npm	PyPI	npm	PyPI	npm	PyPI
	-1	-1	0	0	1--10	1--10	Mean	Mean	Median	Median	STD	STD
✓ Dangerous-Workflow	0.0%	0.0%	0.2%	0.3%	99.8%	99.7%	10.0	10.0	10.0	10.0	0.5	0.5
Pinned-Dependencies	0.1%	0.3%	0.1%	0.5%	99.8%	99.2%	9.5	8.7	10.0	10.0	1.3	2.1
✓ Vulnerabilities	0.3%	0.2%	0.0%	0.0%	99.7%	99.8%	10.0	10.0	10.0	10.0	0.0	0.0
✓ Binary-Artifacts	0.0%	0.0%	0.5%	1.4%	99.5%	98.6%	9.9	9.7	10.0	10.0	0.8	1.4
✓ Token-Permissions	0.0%	0.0%	**15.6%**	**28.2%**	**84.4%**	**71.7%**	8.4	7.2	10.0	10.0	3.6	4.5
✓ License	0.0%	0.0%	**31.4%**	**12.0%**	**68.6%**	**88.0%**	6.9	8.8	10.0	10.0	4.6	3.3
✓ Code-Review	0.3%	0.2%	**68.9%**	**65.1%**	**30.8%**	**34.8%**	1.4	1.5	0.0	0.0	2.7	2.7
✓ Maintained	0.0%	0.0%	**86.1%**	**75.9%**	**13.9%**	**24.1%**	1.1	1.8	0.0	0.0	2.9	3.5
✓ Branch-Protection	1.7%	1.5%	**88.1%**	**88.2%**	**10.2%**	**10.3%**	0.5	0.5	0.0	0.0	1.7	1.6
✗ Dependency-Update-Tool	0.0%	0.0%	94.5%	97.1%	5.5%	2.9%	0.6	0.3	0.0	0.0	2.3	1.7
✓ Security-Policy	0.0%	0.0%	96.8%	97.4%	3.2%	2.6%	0.3	0.3	0.0	0.0	1.8	1.6
✗ Packaging	99.0%	94.1%	0.0%	0.0%	1.0%	5.9%	10.0	10.0	10.0	10.0	0.0	0.0
✗ CII-Best-Practices	0.0%	0.0%	99.8%	99.8%	0.2%	0.2%	0.0	0.0	0.0	0.0	0.2	0.1
✗ Signed-Releases	97.5%	93.0%	2.5%	6.5%	0.1%	0.5%	0.3	0.7	0.0	0.0	1.6	2.5
✗ Fuzzing	17.9%	18.0%	82.1%	82.0%	0.0%	0.1%	0.0	0.0	0.0	0.0	0.1	0.3

We used color scaling technique, the column with green color indicate a higher percentage of packages that passed security checks with a score between [1-10] and a gradual transition to red to indicate an increasing percentage of packages that failed security checks with a score of [0]. The column-shaded Grey also follows the similar concept where percentage of packages had inconclusive results, hence, achieved a score of [-1], instead of [0-10]. The numbers in bold represents the security practices for which we observed gaps in adoption.

✓ Effective Metrics to measure Security

✗ Metrics require industry consensus before adoption

To display accurate stats of ecosystem practices, the Mean, Median, and STD(standard deviation) columns did not consider the [-1] value

we submitted a pull request to the Scorecard repository, adding the GitHub repositories of missing packages to collect the scores from both ecosystems. The weekly Scorecard scan was able to run on those GitHub repositories after the Scorecard team successfully merged the PR.

Out of the 947,936 npm packages and 211,088 PyPI packages, we collected the generated score of 832,422 npm packages and 191,483 PyPI packages. We reviewed 50 randomly-chosen packages where the Scorecard failed to generate scores and found that we did not have access to those GitHub repositories. We collected the Scorecard score on May 09, 2022.

For each package, we could obtain 15 out of 18 Scorecard security metrics and their aggregate score, with the missing 3 metrics being the CI-Test, SAST, and CONTRIBUTOR metrics. The Scorecard team took out these three metrics to scale the weekly job since computing these metrics is API intensive, and GitHub rate limiting can be a bottleneck for the weekly run. As a result, we could not collect data for these three metrics.

6.2.3 Scorecard metrics evaluation

To **evaluate** the **Scorecard tool** metrics and learn why a metric passed or failed, we manually reviewed 25 sample GitHub repositories from each ecosystem for each practice. We ranked each metric by the highest number of dependents and selected the top 25 packages. One author reviewed 50 GitHub repositories (25 from each ecosystem) totaling 750 repositories for the 15 metrics. A second reviewer then verified the findings by selecting 100 repositories at random. We used the Cohen Kappa statistic to test the inter-rater reliability and achieved a score of 0.96. We resolved our disagreement after discussing our findings, and the first reviewer

cross-reviewed other repositories to make changes if required. Then, we needed to examine more packages to understand a given score for Vulnerabilities and Code-Review metrics. We again chose further packages by highest dependent order.

6.2.4 Ecosystems security practices evaluation

We observe each ecosystem's security practices and patterns by comparing the Scorecard security metrics scores in three categories: [-1, 0, 1-10]. For each of the 15 security practices, the frequency of packages is measured using these categories ([-1, 0, 1-10]), as shown in Table 6.2.

The notation "—1" denotes the percentage of packages achieving a score of -1 and indicates that Scorecard could not get conclusive evidence of the practice being implemented, or perhaps an internal error occurred due to a runtime error in Scorecard. The notation "0" denotes the percentage of packages achieving a score of 0 and means that the tool reports indicate the practice was conclusively determined not to be implemented. Since a value of 0 will affect the package's aggregate score, Scorecard assigned a value of -1 to avoid the penalty of failing a metric and also to distinguish between conclusive and inconclusive outcomes. Seven of the 15 security metrics had packages with a score of —1.

The notation "$1-10$" denotes the percentage of packages achieving scores ranging from 1 to 10. In table 6.2, the [1-10] columns display the frequency of npm and PyPI package scores in descending order. Additionally, the mean, median, and standard deviation (STD) are measured to understand an ecosystem's central tendency and spread of score distribution.

6.3 Results

This section discusses the finding of our study. While evaluating Scorecard metrics and ecosystem security practices, we looked into whether Scorecard metrics-

- Are effective for npm and PyPI ecosystems.
- Require improvement of Scorecard tool.
- Require industry consensus to identify standard practices
- Reported lack of adoption of security practices in npm and PyPI ecosystems

In Table 6.2, a higher % (green cell) in "$1-10$" column shows that more than 50% packages implemented the practices. A lower % (red cell) indicates more than 50% of packages failed the practice and received a score of 0 or -1. Then the green checkmark beside nine metrics represents the metrics that can be used to measure security by Scorecard. The red crossmark besides five metrics indicates that the guideline proposed by Scorecard was not adopted in practice, requires industry consensus due to metrics inheritance reliance on other systems and more than 90% packages in both ecosystems scored either 0 or -1. We discuss each security metric for both ecosystems and the frequency statistics in the following subsection. We also highlight the efficacy of Scorecard security guidelines.

Dangerous-Workflow

This metric indicates the following two patterns in workflows: untrusted code checkout; and script injection with untrusted context variables. More than 99% packages passed the metric. However, we observed 1,938 (0.2%) npm packages and 508 (0.3%) PyPI packages where Scorecard found vulnerable code patterns. Out of the 50 repositories used for manual analysis, we had 8 packages with -1, all of which were the outcome of internal errors, and 11 packages with vulnerable code patterns in workflows, hence, scored 0. Among them, 3 npm packages had untrusted code checkout patterns, and 5 PyPI and 2 npm packages had warnings about script injection. At the end of this section, we provide a case study explaining how an attacker can exploit such patterns in workflows.

Additionally, we observe that Scorecard reports a score of 10 for Dangerous-Workflow metrics in empty repositories because the repositories did not have any GitHub workflows let alone dangerous patterns, and the tool lacks the verification of GitHub workflow's existence in the repositories. Therefore, the Dangerous-Workflow metric script should be improved to detect empty repositories or repositories without GitHub workflows.

Pinned-Dependencies

In both ecosystems, more than 99% of packages had a practice of using at least one pinned dependency. Among these, 81% npm packages and 66% PyPI packages got a score of 10, indicating that they do not have any unpinned dependencies in the listed directories.

The score may not reflect an accurate statistic since Scorecard only check `Dockerfiles, shell scripts`, and `GitHub workflows` to track dependencies. However, the tool does not check `requirements.txt`, `pyproject.toml`, `setup.py`, `package.json`, and `package-lock.json` files in PyPI and npm package repositories. For PyPI, there are different ways to declare and manage dependencies and their version in Python. For example, `pyproject.toml` file for declaring dependencies in PyPI is a relatively new standard but not widespread yet. In practice, developers use `setup.py` (using 'setuptools'), which can be non-deterministic and makes it harder to track PyPI dependency. For npm, `package.json` contains the metadata relevant to the project to manage the project's dependencies, scripts, and versions. To depict the accurate status of pinned dependencies in an ecosystem, the Scorecard team should improve the Pinned-Dependencies metrics scripts considering ecosystem standards to evaluate the package dependency.

We also observed that Scorecard does not verify the presence of `Dockerfiles, shell scripts`, and `GitHub workflows` files in a repository. If a repository did not have any files of those types, a package would receive a score of 10 for not having an unpinned dependency on those missing files.

Vulnerabilities:

More than 99% packages did not have any open vulnerabilities in the OSV database. Hence, they scored 10. Scorecard found 7 npm packages and 5 PyPI packages with unfixed vulnerabilities. In addition, 2,703 npm packages and 322 PyPI packages got a score of -1 for inconclusive results. Our manual repository review selected repositories where the package had inconclusive scores or open vulnerabilities, ranked by number of dependents. Note that, we did not review packages with scores of 10 since these packages did not have any open vulnerabilities reported in the OSV database. The reason behind the negative score (`-1`) was that those

repositories were empty. In total, we found 39/50 empty repositories. One package had 10 open vulnerabilities with a score of 0, and 9 packages had 1 vulnerability open with a score of 9.

Binary-Artifacts

More than 99% packages had a score greater than 0. The manual review of 50 repositories found 8 packages with a score of 0. The reviewers noticed that these packages had more than 9 binary artifacts with a mean and standard deviation of 78.25 and 87.17, respectively. These packages were umbrella projects encompassing a variety of tools and libraries. Clients are forced to use these binary artifacts directly.

Another 32 packages in manual review were given a score from 1 to 10 based on the number of binary artifacts ranging from 0 to 9. A score of 10 means no binaries, a score of 9 means the presence of one binary, and the scores continue to decrease toward 1 as the number of binary artifacts increased toward 9. We also found a false positive in one npm package repository, where Scorecard identified 108 binaries, two of which were `.txt` files. Similar to previous metrics, the Scorecard team should improve the metric script to detect empty repositories along with enhancing the list of binary keywords considering different ecosystems.

Token-Permissions

The metric indicates that npm yielded a more promising result: nearly 84% of packages have read and write permissions declared in workflows, compared to 71% of PyPI packages. Our manual review found similar patterns as we observed in Pinned-Dependencies and Dangerous-Workflow. Fourteen (14) packages did not have any GitHub Actions specified in the repository, but Scorecard assigned 10 to those packages for Token-Permissions. Here, the score was 10 because the tool lacks the verification of GitHub workflow's existence in the repository. The Scorecard team should improve the scripts to detect the presence of GitHub workflows before scoring good or bad practices.

License

We observed that 68% of npm packages and 88% of PyPI packages had published licenses in the GitHub repository, indicating, npm has a higher tendency to avoid licensing in the repository. Our manual review revealed that 4 npm packages and 8 PyPI packages had a license in the repository, specifically in `Readme.md` and `setup.py` files. However, Scorecard did not identify them, hence the metric script should be improved to detect licenses more accurately.

Code-Review

This check evaluates if the package conducts code reviews prior to merging PR. The first step of the check is to see if Branch-Protection is activated with at least one required reviewer. If this fails, the check looks to see if the last 30 commits are Prow, Gerrit, or Github-approved reviews or if the merge differs from the committer. 30% of npm packages and 34% of PyPI packages had code review practices in their repository. A 2022 study [112] also showed that 52.5% of the analyzed updates of npm, Crates.io, PyPI, and RubyGems ecosystem packages were only partially code-reviewed, with an overall median code review coverage (CRC) of 27.2%. One reason behind failing this metric would be that the metric is not applicable if the package has one maintainer.

However, Our manual review found nine packages scored 0 and had no code review practices even though they had more than one contributor in GitHub repositories. Both ecosystems exhibit a gap in implementing Code-Review in GitHub repositories, indicating that packages contain the risk of introducing non-reviewed code in the software supply chain.

We also had −1 in 5 sample repositories where the repos were empty. To verify this pattern, we reviewed an additional 10 repositories with −1. These repositories were empty on GitHub. Hence, indicating why Scorecard assigned −1 as an inconclusive result. In total, we found 2,695 (0.3%) npm and 321 (0.2%) PyPI empty repositories with −1 scoring in Vulnerabilities, Branch-Protection, Packaging, and Signed-Releases and Code-Review metrics.

Maintained

Our findings show that more than 85% packages in npm and 75% PyPI packages were unmaintained in GitHub. What is more crucial is that for npm, unmaintained packages may have a more extended period than 90 days, as study [275] revealed that in 2021, more than 58% of packages in the npm registry were unmaintained over two years. Our manual inspections were consistent with Scorecard data where 9/50 packages were inactive in a range of 1 year to 7 years.

Branch-Protection

Only 10% of packages passed this metric in each ecosystem, indicating these repos had at least one tier of branch protection applied. Hence, 90% npm and PyPI packages had branch protection disabled in the repository. The numbers are considerably high, indicating that a large number of packages in both ecosystems did not create a branch protection rule in repositories. Out of five tiers of scoring- “Enabling branch protection”, “inhibits force to push, and branch deletion” are Tier 1 check. Then, the presence of at least one reviewer (Tier 2), enabling status checks (Tier 3), the presence of a second reviewer (Tier 4), and admin dismisses the stale review (Tier 5) are the other tiers. When Scorecard is run without an administrative access token, the requirements that require admin privileges are ignored to avoid penalizing a package score.

Our manual review found that Scorecard metrics only investigate the default branch and any branch that was used for creating a release and uses GraphQL API to verify the protection. However, we verified the branch-protection by looking into the GitHub branches api [86]. We found 13/50 packages had a score of −1 due to internal error because Scorecard: a) looked for the incorrect branch name that did not exist in the repository; b) could not locate the branch even though it existed; c) the main branch had a different name than the “main" or “master"; and d) branch protections were disabled in main and release branch.

Dependency-Update-Tool

94% of npm packages and 97% of PyPI packages failed this metric because Dependabot and Renovatebot were not used as dependency update tools. A project that uses other tools or manually updates dependencies, will obtain a score of 0 on this metric, just like other packages with outdated dependencies. Dependency-Update-Tool metric call for industry consensus on an ecosystem-wise tool list for Scorecard to report an accurate state. This metric can only confirm if the dependency update tool is enabled; it cannot confirm if the dependency-update-tool is running or if the tool’s pull requests are merged.

Security-Policy

Only 3.2% npm and 2.5% PyPI packages have a `security.md` file. After looking into 50 sample packages, we observed that: a) 25 packages do not adhere to standard security policies; and b) 11 packages have a different reporting procedure for vulnerabilities. Users can, for example, submit bugs in other places such as GitHub issues, specific email addresses, and different bug databases outside of GitHub, or use a different security policy reporting file `security.rst`. Although both ecosystems adopted this practice inadequately, Security-Policy is one of the top recommended GitHub security best practices determined by practitioners [219] and the SSDF framework also suggested adopting the practice.

Packaging

Packaging is another metric that indicates the industry consensus is required. Only 1% of npm packages and 5.8% of PyPI items passed the packaging metric. Since the software can be packaged in multiple ways, the challenges of coordinating several package release protocols may prohibit developers from releasing packages on GitHub Actions, which can be one reason for the limited number of packaging in the GitHub packaging workflows. At the time of this study, Scorecard did not query the package registries directly. Hence, packages that do not use GitHub Actions get -1 instead of 0. Note that a package's aggregate score will be penalized if it has a score of 0; and inconclusive or -1 have no effect on the aggregate score. Our manual inspection identified only 2 npm packages and 6 PyPI packages used GitHub packaging workflow, while 47/50 packages had releases on GitHub. Additionally, the Scorecard failed to detect 4 packages (2 from each ecosystem) that had a publishing GitHub workflow. The names of these files are [publish, ci, release].yml.

CII-Best-Practices

Scorecard found the CII Best Practices Badge in just 1,665 (0.2%) npm and 341(0.1%) PyPI packages. The CII Best Practices program is a way for Free/Libre and Open Source Software (FLOSS) projects to demonstrate that they follow best practices. Projects can voluntarily self-certify to report how they follow each best practice. According to the CII Best Practice Program website only 4,766 FLOSS projects have reported their security policies and received different degrees of badges, indicating both ecosystems did not adopt the practice.

Signed-Releases

Similar to the Packaging metric, the Signed-Releases metric's report suggested the need for industry consensus. Only 578 (0.1%) npm and 936 (0.5%) PyPI packages had signed releases. Moreover, almost 100% packages failed this metric. The low number of Signed-Releases in GitHub repositories are expected behavior for both ecosystems, as package developers release versions to the package registry (`npmjs.org or pypi.org`) rather than code hosting platforms like (`github.com`). Additionally, we observed that GitHub, PyPI, and npm each has different regulations to control package release to a registry. To publish in both registries, the team must take additional steps to confirm the release, which can be incompatible with their workflow [258]. For instance, the GitHub registry accepts only scoped packages. Therefore, if a JavaScript package is currently named *X*, it must be renamed *@username/X* to publish in GitHub.

Scorecard assigns -1 instead of 0 if the tool can not detect the signed release. In addition, our manual review revealed that Scorecard often verifies older signed versions rather than checking for signatures on the

newest five releases. For example, one package received an 8/10 score, meaning 4/5 of recent releases of that package had signed artifacts. However, we found the signed artifacts from older versions, which contradict the defined rules of Scorecard. Then we also observed repositories tagged commits as a release rather than creating a release on GitHub. However, none of the commits were GitHub verified, and Scorecard does not identify tagged releases. Therefore, along with industry consensus, the Signed-Releases metric's scripts need to be updated by Scorecard to improve tool accuracy.

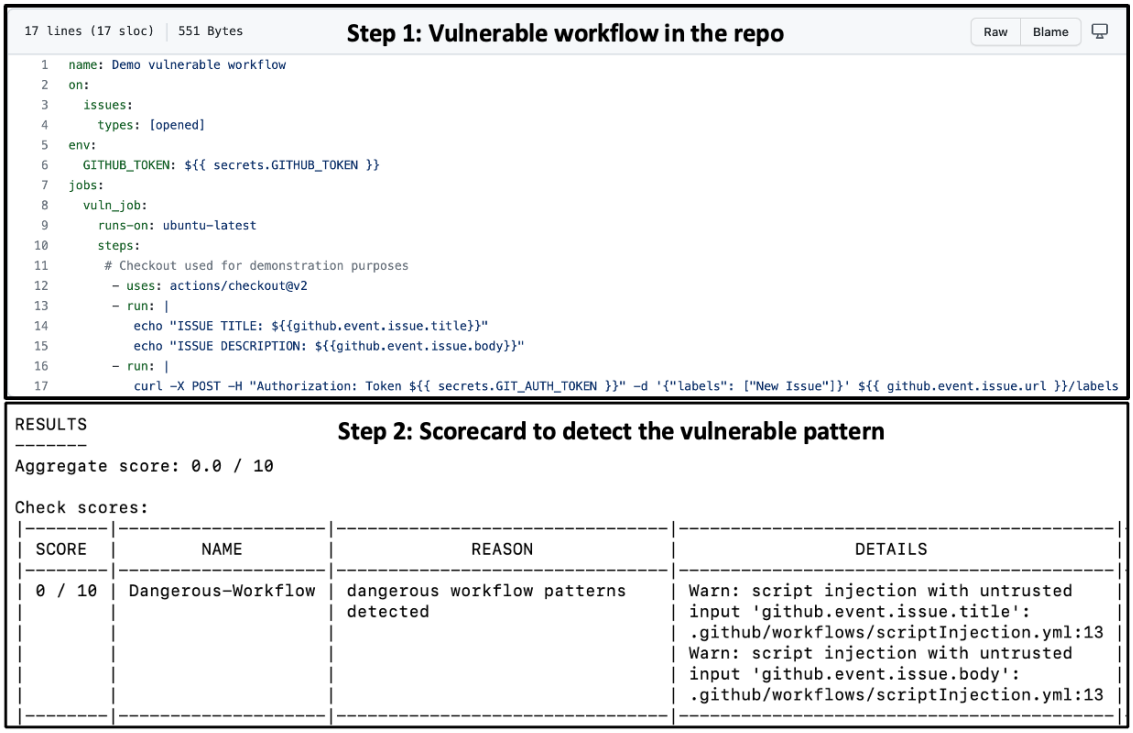

Figure 6.1: Case study on Dangerous Workflow to detect vulnerable code pattern

Fuzzing

Both ecosystems fell short on this metric. Scorecard validates fuzzing exclusively through the tracking of packages in OSS-Fuzz project. OSS-Fuzz has been tested only in 650 open-source packages as of July 2022 and a package that uses fuzzing with other tools would fail the check similar to Dependency-Update-Tool metric, indicating why the npm and PyPI ecosystems failed this metric. Out of 650 open-source packages that use OSS-Fuzz, we found 50 npm packages and 104 PyPI packages. Despite the fact that only a few packages passed this metric, PyPI has more fuzzing practice (50 percent more) than npm. One reason why npm packages do not use fuzzing could be that fuzzing JavaScript (JS) engines is tricky and requires expertise. Instead of processing user-supplied seeds, JS engines scan and interpret user seeds into an abstract syntax tree (AST) [55] which impacts the performance of fuzzers. Our manual analysis yielded no different results from what we expected. Only two PyPI packages used OSS-Fuzz and 48 packages had no fuzzer. Then, Fuzzing had most

of the −1 after Signed-Releases and Packaging, but in our manual analysis re-running 12 packages with −1, scored 0 in new run, indicating run time error occurred during the first run.

6.3.1 Case study on Dangerous Workflow

In this section, we investigate whether we can exploit GitHub workflows' vulnerable patterns for malicious intent and evaluate whether Scorecard can detect the patterns via the Dangerous Workflow metric. We found 508 PyPI packages (96 packages with an average of 58 dependents) and 1938 npm packages (805 packages with an average of 13 dependents) where packages failed the Dangerous Workflow metric, indicating that their workflows contain vulnerable coding patterns. Even if these repositories are not malicious, potentially dangerous misuse of the workflows may allow malicious attackers access to the data and commit data breaches and theft.

We refrained from attacking the repositories from an ethical standpoint since many of these repositories are legitimate projects. Therefore, to test whether Scorecard can be used to anticipate malicious attacks, we executed the Scorecard tool on a dummy GitHub repository where we built a workflow with an intentionally-vulnerable `issue` action, inspired by [59].

We execute a reverse shell attack by creating the following `issue` in the dummy repository from a different GitHub account user- `New malicious issue title" && bash -i >& /dev/tcp/4.tcp.ngrok.io/ngrok endpoint 0>&1 && echo"`. Our vulnerable workflow (Figure 6.1) in the dummy repository is executed on a GitHub runner whenever a new issue is created by any user. Here, line 14 in step 1 of Figure 6.1 `"ISSUE TITLE: {{github.event.issue.title}}"` is vulnerable to command injection because the hosted runners replace the macros `{{ ... }}` blindly and echo `"{{github.event.issue.title}}"` becomes echo `"{{New malicious issue title}}"`, thus, giving an attacker to run a reverse shell inside the hosted runner as part of the arbitrary code execution capabilities. An attacker can read sensitive files like `.credential` from the runner folder.

Step 2 in Figure 6.1 shows that the Scorecard could identify the vulnerable pattern, referring to the exact line number. The dependents stat of these packages shows that malicious injection may allow attackers to execute supply chain attacks. Therefore, this case study substantiates that the Dangerous Workflow metric is effective for identifying malicious attacks on GitHub workflows.

6.4 Discussion

This section discusses the effectiveness and applicability of Scorecard tool security metrics and the gap in npm and PyPI ecosystems security practices. An overview of our discussion is shown in Table 6.3.

6.4.1 Effective metrics for both npm and PyPI ecosystems

Our study reveals that practitioners can use a subset metrics of Scorecard tool to measure security practices, including metrics- **Dangerous-Workflow, Vulnerabilities, Binary-Artifacts, Token-Permissions, License, Code-Review, Maintained, Branch-Protection** and **Security-Policy**. To achieve a higher security score (towards 10), practitioners need to follow the guidelines provided by the Scorecard tool. Most of these are well-established security metrics, also required by the SSDF [230] and SCVS [177] framework. However, the guideline provided by Scorecard for Dangerous-Workflow and Token-Permissions are more GitHub-focused,

Table 6.3: Summary of Scorecard Security Metrics Evaluation

Scorecard Metrics	Evaluation of this study
Dangerous-Workflow	Effective Metric, Need-for-Improvement
Pinned-Dependencies	Need-for-Improvement
Vulnerabilities	Effective Metric
Binary-Artifacts	Effective Metric, Need-for-Improvement
Token-Permissions	Effective Metric, Need-for-Improvement, Lack of adoption
License	Effective Metric, Need-for-Improvement, Lack of adoption
Code-Review	Effective Metric, Lack of adoption
Maintained	Effective Metric, Lack of adoption
Branch-Protection	Effective Metric, Lack of adoption
Dependency-Update-Tool	Require Industry consensus
Security-Policy	Effective Metric, Lack of adoption
Packaging	Require Industry consensus
CII-Best-Practices	Require Industry consensus
Signed-Releases	Require Industry consensus
Fuzzing	Require Industry consensus

which requires practitioners to implement GitHub workflows to achieve higher scores. Both of these metrics effectively detect security weak links in GitHub workflows, e.g. Dangerous-Workflow can be used to prevent malicious PRs and issues. Therefore, Dangerous-Workflow and Token-Permissions metrics will not be useful to practitioners who do not use GitHub workflows for their CI/CD pipeline.

6.4.2 Need-for-Improvement metrics

The **Dangerous-Workflow, Pinned-Dependencies, Binary-Artifacts, Token-Permissions**, and **License** metrics exhibit the need for Scorecard team's attention for improvement. The Pinned-Dependencies metric requires revision for different ecosystems. For example, Pinned-Dependencies do not check the `package.json` and `requirement.txt` and other files for the dependency version, even though the PyPI and npm contain dependency information in such files. License metrics can be improved by enhancing the list of keywords.

Then, accurate ecosystem evaluations require filtering out packages with empty repositories. However, the Scorecard generates aggregate scores for empty repositories. Because Dangerous-Workflow, Binary-Artifacts, Pinned-Dependencies, and Token-Permissions metrics indicate risky patterns in GitHub. For these metrics, empty repositories obtained a score of 10, as repositories were completely devoid of any content, let alone risky patterns. Hence, Scorecard assigned a score of 10 instead of 0, or -1, whereas the other 11 metrics values were between $0, -1$. Another similar example would be even if a package does not have GitHub workflows, the tool will automatically score 10 in Dangerous-Workflow and Token-Permissions metrics which do not exactly reflect that the package follows good workflow patterns. Our findings suggest that Scorecard should check for the existence of GitHub repositories or workflows before reporting on good or bad security practices, since high scores give us a false sense of good security practices. We submitted our findings to the Scorecard team,

and the team acknowledged and agreed to improve Scorecard to enable automated testing more effectively in Version 5.

6.4.3 Industry consensus required on metrics

The **Dependency-Update-Tool, Packaging, CII-Best-Practices, Signed-Releases**, and **Fuzzing** indicate the requirement of industry consensus before Scorecard can promote these metrics to practitioners. Scorecard has proposed guidelines for these practices, but without industry consensus, these metrics hardly have any value from an ecosystem security perspective. Both ecosystems exhibited weak adoption of these practices.

For example, Scorecard requires practitioners to use specific tools for the Dependency-Update-Tool and Fuzzing metrics to achieve higher scores. However, the industry lacks consensus on the list of tools or research showing the ecosystem's preference regarding those tools. The ecosystem needs to agree on or standardize these tools so that Scorecard can measure the practices. Then, practitioners did not show evidence of using Packaging and Signed-Releases practices on GitHub. The reasons could be that practitioners used the package registry to release the signed/unsigned version and used GitHub as a platform for source code distribution. Either Scorecard could integrate with package registries to collect accurate data on Packaging and Signed-Releases, or practitioners could agree to release signed packages on GitHub, which can be an additional task since different platforms may have different regulations for releasing a package. CII-Best-Practices requires maintainers to self-report their security practices' adherence. Therefore, failing these metrics may not necessarily advocate package owners failed to implement the practices required for CII-Best-Practices, they simply may not have self-reported the practices.

We do acknowledge, however, that such an industry-wide agreement may be challenging and may take time to implement. In that case, Scorecard may separate these metrics from the aggregate score calculation. If a package implements these practices, the package may get bonus points instead of directly impacting the aggregate score. Either way, practitioners and the Scorecard team should address the above-mentioned issues to achieve an accurate picture of ecosystem security practices.

6.4.4 Ecosystems security comparison

In the case of License, Code-Review, and Maintained, PyPI outperformed the npm ecosystem. For example, only 68% npm packages had a published license in the repository, compared to over 88% of PyPI packages. Although both ecosystems failed the Fuzzing metric check, we found that PyPI exhibited 50% more fuzzing tools implementation than the npm ecosystem. Then, the Token-Permission metrics showed that npm (84.4%) has better file permissions in the GitHub workflow compared to the PyPI (71.7%) ecosystem.

6.4.5 Lack of adoption in ecosystem-wide security practices

Both ecosystems indicate lack in practicing **Token-Permission, License, Code-Review, Maintained, Branch-Protection,** and **Security-Policy** practices in the GitHub repository. These Scorecard metrics effectively measure security in GitHub (Table 6.2), but both ecosystems showed inconsistency in adopting these security practices. On the contrary, metrics that require industry consensus demand modification in guidelines proposed by Scorecard, metrics have inheritance reliance on other systems, and more than 90% packages in both ecosystems scored either 0 or -1.

Even if the Token-Permission metric needs to be improved, Scorecard identified 15.6% of npm repositories and 28.2% of PyPI repositories containing `yaml` files with write access, indicating package susceptibility to malicious attack. License is important for an organization to comply with organization's legal policies. 30% npm packages and 12% PyPi packages did not contain any valid License in GitHub repositories which is legally require for any organization intending to use those packages. Then both ecosystems lacked to adopt Code-Review (npm: 69%, PyPI: 65%) and Maintained (npm: 86%, PyPI: 76%) metric, indicating the risk of using unreviewed, unmaintained code. Additionally, around 90% of the packages in both ecosystems did not show evidence of implementing default Branch-Protection and Security-Policy practices in their repositories.

Token-Permission, Code-Review, Maintained, and Security-Policy were all listed by the SSDF framework as important security practices, highlighting the significance of implementing these practices.

6.5 Limitations

In our study, we group security scores into three categories ([-1,0,1-10]) to avoid arbitrary scoring bias; some metrics scoring may not be representative of the severity of security risk. For example, the vulnerabilities metric looked for open vulnerabilities in OSV database and assigned scores based on number of open vulnerabilities. However, the tool does not look into the severity of vulnerabilities. If a package has one exploitable vulnerability it will score 9, whereas a package with more than 9 open but non-exploitable vulnerabilities will score 0. Even though 9 seems like a better score, severity-wise, it is a high-risk package with exploitable open vulnerabilities. Additionally, if a package did not contain any vulnerabilities reported in OSV database, the package will receive a score of 10, which does not confirm that the package is free of vulnerabilities. Although this scoring is a limitation of the Scorecard tool, and we tried to reduce the bias by grouping the score into three categories, metrics like vulnerabilities may prevent us from achieving accurate security status of these ecosystems. Then, our case study on Dangerous Workflow on a dummy repository may not represent 2,446 packages, which is a limitation of this study.

6.6 Conclusion

This study compares the npm and PyPI ecosystems' security practices in GitHub repositories using Scorecard tools. Our work focuses on measuring and understanding the adoption of cross-ecosystem package security practices. We also evaluated whether we can leverage the Scorecard tool metrics to measure ecosystem-wide automated security practices. We found that 13 Scorecard security metrics were compatible with the SSDF framework. Next, we identified 9 Scorecard security metrics that can be used to measure npm and PyPI package security. Then, "Dangerous Workflow" can aid in identifying malicious attacks. Five practices, however, necessitate industry agreement. Both ecosystems showed gaps in implementing Token-Permission, License, Code-Review, Maintained, Branch Protection, and Security Policy practices.

Knowing about these security practices and their challenges will inspire and direct practitioners on what to do to adopt these practices or identify the gaps preventing them from doing so. We have also observed and been told that the Scorecard team welcomes new security metrics and discussions that indicate the Scorecard is evolving with time. Therefore, our study aims to draw practitioners' attention to creating action plans to enhance Scorecard security metrics for assessing ecosystem-wide security practices. To improve the

tool's ability to measure security automatically, the ecosystem managers, the Scorecard team, and the practitioners can drive ecosystem-wide standards. Such industry-wide consensus will push software producers to start implementing those practices.

CHAPTER

7

SOFTWARE SECURITY PRACTICES YIELD FEWER VULNERABILITIES

As organizations seek to address escalating security risks and comply with regulations, a myriad of activities are available to improve software security. The National Institute of Standards and Technology (NIST) issued guidance [230, 159] on software development practices that enhance the security of the software supply chain. The Open Source Security Foundation (OpenSSF), a cross-industry organization hosted at the Linux Foundation, provides the vehicle for collaboration on tools, services, training, infrastructure, and resources for securing open-source projects. The OpenSSF Scorecard [206] project computes automated scores of package's security practices to help developers make better decisions about security when consuming open source projects.

While NIST's guidelines and OpenSSF projects provide guidelines and comprehensive lists of security practices, challenges arise in validating whether these practices improve security. Organizations may not have the resources to adopt a full suite of these practices. They would like to understand the key drivers of success, especially which of many possible software security practices to undertake first. A data-driven study on the relationship between the use of software security practices and vulnerability count could aid in this understanding. ***The goal of this study is to assist practitioners and researchers in making informed decisions on which security practices to adopt through the development of models between software security practice scores and security vulnerability counts.***

To that end, this study is focused on the relationship between publicly-available data of security practice use and externally-reported vulnerability count. This relationship may be used to provide actionable recommendations to practitioners on security practices. Our work addresses the following **research questions**:

- **RQ1 (Security Practices)**: Which Scorecard security practices are most important to understand the

relationship between security practices and vulnerability counts in regression models?

- **RQ2 (Security Outcome)**: Do packages with higher aggregate security scores have fewer vulnerabilities?

The OpenSSF Scorecard project [206] uses security practice metrics and auto-generates a "security score" for each practice. The tool also computes an aggregate "security score" which is a weighted average of the individual score. In this study, we leverage the security practice score data provided by the OpenSSF Scorecard project and vulnerability information from security advisory databases to evaluate the relationship between security practices and vulnerability count. To answer RQ1, we used the "feature importance" technique in four regression models to understand the importance of each security practice in interpreting the relationship between security practices and vulnerability count. For RQ2, we built our fifth regression model to evaluate the statistical association of a package's aggregate security scores with package vulnerability count and to understand whether implementing security practices assists in secure coding with fewer vulnerabilities.

This study can assist practitioners in making informed choices regarding their security practices, particularly on which practices have the most impact or whether specific security practices have improved security outcomes. This study makes the following contributions:

- A proposed model to identify important security practices to enhance software security.
- A proposed model to understand how higher aggregate security scores affect security outcomes.
- Evaluation of the proposed model and security practices using open source datasets.
- The list of challenges impeding the model's performance

This chapter is organized as follows: Section 7.1 describes the data collection process, and Section 7.2 describes the data analysis to measure model performance. We close with a discussion and the limitations of our findings (Section 7.3).

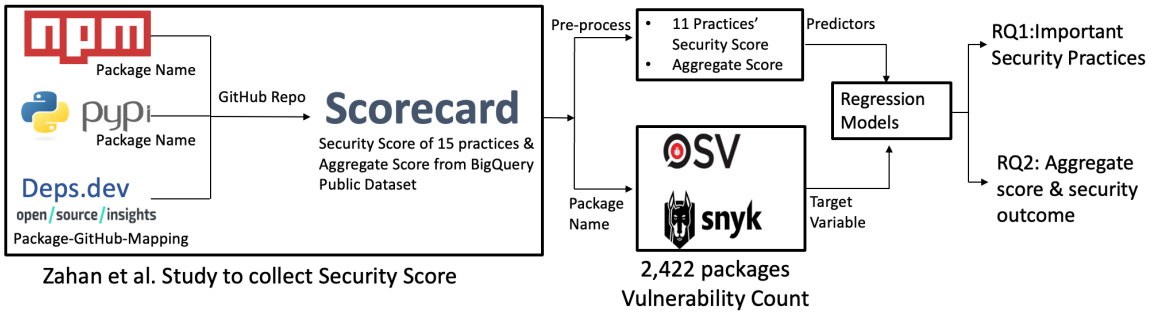

Figure 7.1: Research Workflow

7.1 Study Design and Data Collection

In this section, we discuss our study design and the data collection process that was used in our research. Section 8.1 highlights our high-level overview of the research workflow. Section 7.1.2 covers how we ob-

Table 7.1: Unique vulnerabilities count collected from OSV and Snyk

Ecosystem	Database	Github Package	GitHub Vul_Count	CVE	Non-CVE	Total GitHub Package	Total GitHub Vul
PyPI	Snyk	523	1189	830	359	589	2261
	OSV	352	1407	1374	33		
npm	Snyk	1689	2838	1888	950	1833	3655
	OSV	1311	1883	1559	324		

tained Scorecard security practices' scores for each package, and Section 7.1.3 discusses how we collected vulnerability count data from Open Source Vulnerabilities (OSV) and Snyk databases.

7.1.1 Study Design

To answer our two research questions, we leverage regression models where we use the security scores of package repositories as independent variables and security outcomes in terms of package vulnerability count as the dependent variable. Figure 7.1 shows an overview of the research workflow.

RQ1 aims to identify the most important practices that explain the relationship between security practices and security outcomes. To that, we utilized the 15 security practice scores as independent variables generated by the Scorecard tool. Next, our RQ2 verifies whether packages with higher aggregate security scores have fewer vulnerabilities. Here, we leverage each repository's aggregate security score as an independent variable, which is generated by Scorecard by measuring the weight-based average of the individual security practices' scores, weighted by risk. For both RQ1 and RQ2, we used the security outcomes in terms of vulnerability counts as the dependent variable.

For RQ1, we classified a security practice as an important metric if it is identified as important in more than one model. To that, we analyzed 4 different regression model responses. For RQ2, we developed our fifth regression model to test our hypothesis.

7.1.2 Scorecard Security Practices data

We obtained Scorecard security practices data from [272](see Chapter 6). The data includes the 15 security practices scores and aggregate security score of 767,389 npm packages and 191,158 PyPI packages. The dataset contained mapping between the package name, the GitHub repository, the 15 security practices, and the aggregate security score. Each practice score ranges between -1 to 10. We describe the OpenSSF Scorecard project in Chapter 6. More details of these checks can be found in the Scorecard Project GitHub repository [206].

The Scorecard tool has 18 security practices. However, the OpenSSF Scorecard team publishes the security score data for 15 of these practices. The Scorecard team took out CI-Test, SAST, and Contributor practices to scale the weekly job since computing these practices is API intensive. The security scores for the other three practices can be obtained by running Scorecard from the command line interface (CLI). Our study can be replicated for 15 security practices by collecting weekly data from the Google BigQuery dataset published by the Scorecard team.

7.1.3 Advisory Data

This section provides background on the database used in this study and advisory data collection processes to obtain the vulnerability count.

Database

We used the OpenSSF Open Source Vulnerabilities (OSV) [93] and Snyk.io vulnerability [218] databases in this study. Both databases have a similar structure in terms of the patch, Common Vulnerabilities and Exposures (CVE), and Common Weakness Enumeration (CWE) information and they track vulnerabilities across multiple ecosystem's databases. In September 2022, we acquired vulnerability data from these two databases for both ecosystems.

OSV provides both human and machine-readable data formats to describe vulnerabilities and maps to open-source package versions or commit hashes. OSV is an aggregator of vulnerabilities from different security advisories, e.g., GitHub Security Advisories [85], PyPI Advisory Database [186], Go Vulnerability Database [89], Rust Advisory Database [197], OSS-Fuzz (mostly C/C++) [92], and Global Security Database [16]. These databases have adopted the OpenSSF OSV format to facilitate use in tooling and aggregation. **Snyk.io** is another platform that provides security reports containing vulnerability information for different package ecosystems, including npm, PyPI.

Vulnerability Data Collection

As Scorecard only works for GitHub repositories, our vulnerability collection was limited to packages with GitHub repositories. Hence, each database may have more vulnerabilities than we presented in this study. Table 7.1 contains the detailed counts of vulnerabilities.

The [272] data contained 767,389 npm and 191,158 packages with GitHub repositories. We extract the vulnerability data from both databases using the package name. For each package, we collected vulnerability details in terms of CVE, CWE, and patch link from both databases. OSV exported data to a Google Cloud Storage (GCS) bucket. The bucket contains a zip of all vulnerabilities JSON file for each ecosystem at `gs://osv-vulnerabilities/<ECOSYSTEM>/all.zip`. However, we had to build a scraper for Snyk to extract data from the Snyk website.

To create the dataset, we counted the number of unique vulnerabilities found for each package. First, we merged both databases and applied our filter conditions. During this stage, we eliminated vulnerabilities if we found i) duplicate package-to-CVE mappings or duplicate patches for the Non-CVE vulnerabilities; ii) Non-CVE vulnerabilities without patch links; iii) Vulnerabilities with Non-CVE, NON-CWE and No Patch link. Table 7.1 shows the unique count for each row. After filtering and merging both advisory databases, we found 589 PyPI packages with 2261 vulnerabilities and 1,833 npm packages with 3,655 vulnerabilities. In total, we have vulnerability counts of 2,422 packages as a dependent variable.

At this stage, we mapped 2,422 packages with the Zahan et al. [272] dataset to collect the Scorecard security practice's values. As a result, 2,422 packages with vulnerability count and security score are used as our final dataset, and we did not consider any other packages from [272] study.

7.1.4 Dependents and Downloads Data

The number of dependents and downloads reflects the importance of a project based on how many other projects use it. To determine whether the packages in our dataset are utilized by other open-source packages, we obtained dependents and downloads data. We collected direct-dependent and transitive-dependent information from the Open Source Insights (OSI) API (Section **??**). Then, we collected 12 months of download statistics of npm packages from the study [**?**] and for PyPI packages from the Google BigQuery public dataset.

7.2 Data Analysis

This section discusses our data analysis to answer our research questions. Sections 7.2.1 and 7.2.2 discuss pre-processing and regression models to answer our research question.

7.2.1 RQ1: Important Security Practices

In this section, we discuss the analysis approach for RQ1: *Which Scorecard security practices are most important to understand the relationship between security practices and vulnerability counts in regression models?*

Data Pre-processing

In this section, we discuss the data pre-processing of missing, inconsistent, and noisy data found in Scorecard data due to internal and external errors. For example, we eliminated predictors and packages from our dataset if Scorecard generated a score of -1. The score -1 depicts that the tool could not obtain concrete evidence that practice had been implemented or if an internal error had occurred due to a runtime error. Therefore, we utilized the following exclusion criteria for security practices and packages to acquire consistent security scores.

- **Practice Exclusion**: We removed security practices if a practice has more than 10% packages with -1 score. Signed-releases (91%), Packaging (94%) and Fuzzing (16%) practices had more than 10% packages with a score of -1. We also removed "Vulnerability" metrics as a predictor. The metric checks whether a package has open or unfixed vulnerabilities in the OSV database. Since part of our dependent variable was collected from OSV, we removed this metric to avoid introducing bias. Although we did not find any package in our dataset where a package had open vulnerabilities, it may occur in any future packages. To standardize our study for replication, we removed the vulnerability metric from our dataset. At this stage, we removed 4 practices and have 11 practices as predictors.
- **Package Exclusion**: We found Branch Protection (6%), and Code-Review (0.08%) had less than 10% of packages with a score -1. Therefore, we only removed those packages instead of removing the practice itself. At this stage, we removed 150 packages from the dataset.

To protect the quality of the data that can directly affect how well our ML model learns, we are removing these practices. Given that Scorecard will compute the accurate security status of these practices in the future, practitioners can elect them when constructing our model. Conversely, leveraging different vulnerability databases other than OSV would allow us to use vulnerability practice as one of the predictors. However, the lack of vulnerability data compelled us to gather vulnerabilities from several sources.

Then, we used `qq-plot` [209] to determine outliers in vulnerability count data. Four packages had vulnerability counts (36, 77, 158, 217) far from the reference lines and were considered outliers. Since we will use regression models, extreme outliers will affect the line of regression. Hence, we removed those packages from our dataset as an outlier in the dependent variable. We utilized 2268 packages and 11 security practices in our model.

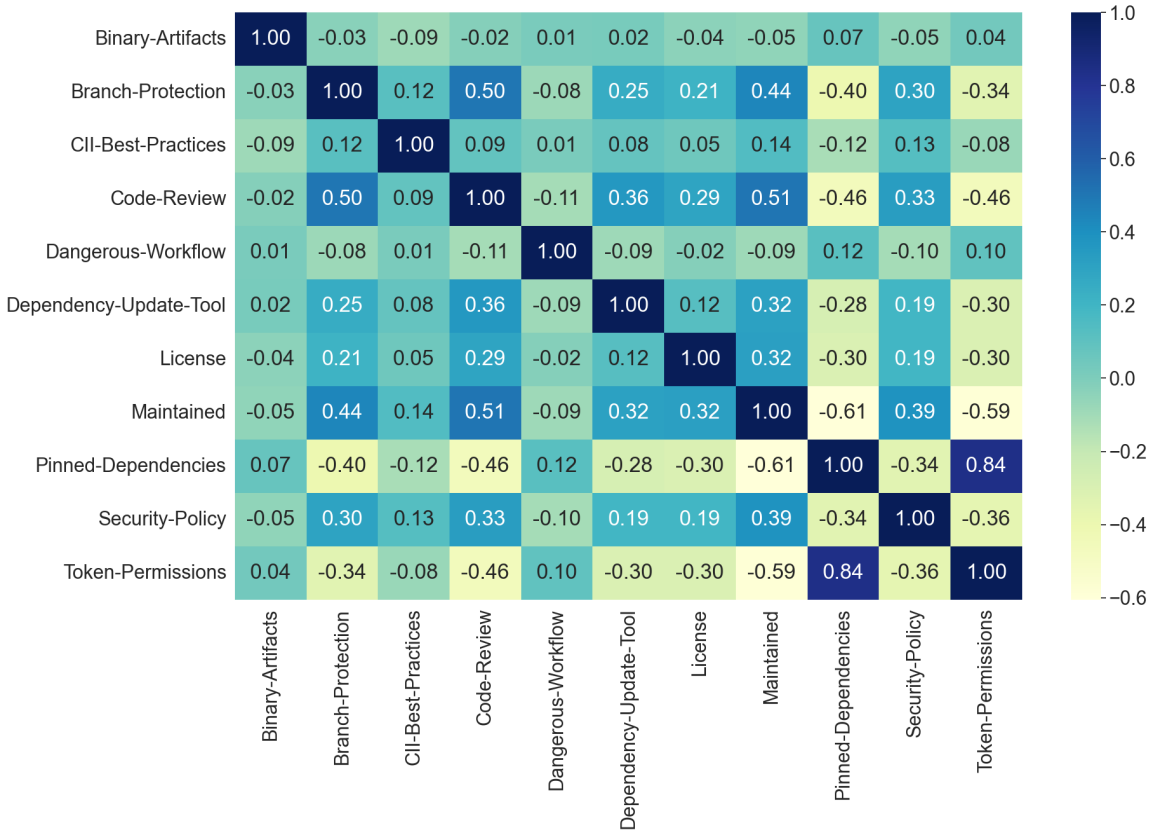

Figure 7.2: Heatmap matrix for the Spearman rank correlation test

Correlation Analysis

To evaluate the normality of the dataset, the Henze-Zirkler multivariate normality [104] test was utilized. Results indicated that the dataset was not normally distributed. We expected non-normality, given that the independent variables were discrete variables. At this stage, we measure the correlation to know the association or the absence of a relationship between two variables in our dataset. With evidence of non-normality, we deployed the Spearman rank correlation [276] tests.

A value close to +1 and -1 indicated a strong positive or negative correlation, respectively. A correlation coefficient close to 0 indicates a weak correlation between the variables [190]. Since values above ±0.3 start to indicate a moderate positive (negative) linear relationship [190], we consider values between ±0.3 to ±1 as a limit. Figure 7.2 shows the heatmap matrix for the Spearman rank correlation test.

The Spearman test showed the strongest correlation between Pinned-Dependencies and Token-Permission(0.84), followed by Code-Review and Maintained (0.51), Branch-Protection showed a moderate correlation with Code-Review (0.50), and Maintained (0.44) and Security Policy (0.3). Negative correlations were observed between Maintained and Pinned-Dependencies (-0.61), Maintained and Token-Permission (-0.59), Code-Review and Pinned-Dependencies (-0.46), and Code-Review and Token-Permissions (-0.46).

Token Permission looks at the permission of the GitHub workflow files, and Pinned Dependencies look for the presence of unpinned dependencies in those files, indicating why they might have a strong positive relationship. Both the Code-Review and Maintained variables showed a moderate positive relationship. In contrast, an increase in Maintained comes with a decrease in Pinned-Dependencies and Token-Permission practices. Token-Permission and Pinned-Dependencies do not require active maintenance directly, which might explain the inverse relationship. However, code review could be part of active maintenance, hence the positive relationship. Branch-Protection enables a formal approval process for code changes and pairs well with Maintained and Code-Review.

Feature Selection

For the selection of best features, we used the `SelectKBest` [216] library from the $sklearn$, which shows the feature importance based on K highest scores. F-regression was selected as the scoring function, which performs a univariate linear regression test and returns F-statistics and p-values. All of the independent predictors were considered to identify the most important features (K="all"). The SelectKBest analysis identified nine predictors as the most important features: Maintained, Security-Policy, Pinned-Dependencies, Code-Review, Branch Protection, Token-Permission, License, CII-Best-Practices and Dependency-Update-Tool. Each of them had a high F-score and statistically significant p-values ($p<0.01$). A reduced dataset was prepared from the statistical significance measure with the selected components as independent variables and the total vulnerability count as the dependent variable.

Table 7.2: Model performance and important features

Models	R^2	**Adj.** R^2	**Important Features** (Ranked)
OLS	0.09	0.08	Security Policy, Pinned Dependencies, Maintained
DT	0.11	0.10	Security Policy, Pinned Dependencies, Branch Protection, Maintained, Code Review
GB	0.12	0.10	Code Review, Security Policy, Pinned Dependencies, Branch Protection, Maintained
RF	-0.008	-0.022	Code Review, Pinned Dependencies, Branch Protection, Maintained, Security Policy

Model Performance:

We measured the performance of each model with the following metrics:

- **R^2** is the statistical measure representing the proportion of the variance for a dependent variable ex-

plained by an independent variable or variables in a regression model. R-squared explains to what extent one variable's variance explains the second variable's variance or "percent of variance explained by the model" [190]. It supports values between 0 and 1, with greater values suggesting more variability explained by the model. A high R^2 value suggests good explanatory power but not predictive capability [284].

- The **adjusted R^2** is a variant of R-squared that accounts for predictors that are not significant in a regression model. By considering the degrees of freedom of the independent variables and the sample population, adjusted R^2 demonstrates how well the data fit a curve or line and explains any bias in the R^2 value. A model's adjusted R-squared will decrease if insignificant variables are added and will increase when significant variables are added. Adjusted R^2 will always be less than or equal to R^2.

Regression Model

Four supervised machine learning models were used to evaluate the importance of each feature variable in interpreting the relationship between security practices and vulnerability count. Feature Importance techniques in machine learning models assign a score to input features for a given model based on their importance in predicting the output.

We first applied **ordinary least square (OLS)** to the dataset. Then, tree-based regression models were selected due to the non-normal distribution of the dataset. Since non-parametric tests do not require the data to follow the normal distribution [256], we applied three tree-based regressor models in this study: **Decision tree (DT), Random Forest (RF)** and **Gradient Boosting (GB)**. Following a split of the dataset into a 70/30 training/test split, all models were fit on the training dataset. Then, we used grid search techniques to optimize each model's hyperparameters. To minimize bias and over-fitting, ten-fold cross-validation was used during model training.

Table 7.2 provides the performance comparison of four regression models. The dataset contains nine statistically significant variables suggested by the `SelectKBest` tool. Compared to all models, the Gradient Boosting (GB) model showed the highest R^2 and adjusted R^2 values, followed by DT, OLS and RF. However, the inferior performance of each model indicates that models (Table 7.2) were unable to explain the variability above 12%. The OLS model found the highest coefficients for Security Policy, Pinned Dependencies, and Maintained. The DT model included Branch Protection and Code-Review as important features, along with Security Policy, Pinned Dependencies, and Maintained. The GB model ranked the same practices as DT as important practices. However, in the case of ranking, Code-Review was the most important metric in GB, whereas DT ranked Code-Review last. The RF model showed the lowest prediction performance of all four models, with a negative R^2 score.

The performance of four models indicates that the currently-available independent security practices are inadequate to explain the variability of vulnerability counts. In Section 7.4, we discuss factors that might influence the model's prediction performance.

7.2.2 RQ2: Aggregate Security Score and Security Outcomes

In this section, we describe our analysis for RQ2: *Do packages with higher aggregate security scores have fewer vulnerabilities?* We look for the statistical association of a package's aggregate score with package vulnerability count to understand if implementing security practices assists in secure coding with fewer vulnerabilities.

We hypothesize that- *packages with higher security scores promote better security practices and contain fewer externally-reported vulnerabilities than packages with lower security scores.* We mentioned in Section **??** that Scorecard generates an aggregate score for each package, which is a weighted average of the individual checks, weighted by risk. To that end, we used the vulnerability count data of 2,422 packages (Section **??**) as a target variable and the aggregate score of those packages as independent variables to answer RQ2.

We removed six packages as outliers, identified by `qq-plot`. We perform the normality check to verify the normal distribution of the independent variable or aggregate score. We found the data was normally distributed to proceed with the linear regression model. The model, however, rejected our hypothesis by showing a significant positive relationship between aggregate score and vulnerability count. The linear model (p-value < 0.001, $R^2 = 0.04$, adj. $R^2 = 0.03$) had a 0.5-unit increase in vulnerability count for every unit increase in aggregate security score.

7.3 Discussion

This section discusses the implications of our findings. We also addresses the other factors (section 7.4) that might impact the effectiveness of the use of Scorecard scores to identify the security practice.

7.3.1 Important security practices

To explore the findings related to RQ1, we used a wide range of security practices, multiple statistical learners, and prediction models to help practitioners to identify important security practices for improving package security. Table 7.2 showed the performance of four models.

Since we were interested in understanding the relationships between the variables, a low R-squared does not always negate the importance of any significant variables where there might be multiple unknown factors. Even with a low R-squared, statistically significant coefficients represent the mean change in the dependent variable given a one-unit shift in the independent variable [78, 95]. Hence, using statistically significant variables as predictors indicates there is a relationship between these predictors and the vulnerability count in the current dataset. Our models explain the past data in a statistically significant manner, but its predictive ability is less than 12% to extrapolate beyond the available data.

Security Policy, Pinned Dependencies, Maintained, Code Review and **Branch Protection** were the most important practices to explain our models. These five practices were statistically significant and came as an important feature in at least three models. Even if we discounted the RF model because of negative R^2, Security Policy, Pinned Dependencies, and Maintained came as an important practice in OLS, DT, GB models. Code Review and Branch Protection were important in DT and GB models. Our findings align with the Sonatype report [226], where they found **Code-Review, Pinned Dependencies, Branch Protection**, and **Binary-Artifacts** as important practices for Java projects. Binary-Artifacts were not important in our model for npm and PyPI ecosystems. Sonatype used 12k Java Projects, which is six times bigger than our dataset. We need more research to validate whether it is because of the size of the dataset or because Binary-Artifacts is a crucial practice for Java projects.

From correlation analysis (Section 7.2.1) we have observed that Pinned Dependencies is negatively co-related with four of these important practices (Security Policy, Maintained, Code Review, and Branch Protection). Pinned-Dependency prevents auto-updating a dependency to a new version without reviewing

the differences between the two versions, which may include an insecure component. Zahan et al. [272] showed that “Pinned-Dependencies” has high false positives in the Scorecard’s current computing process, which might explain the inverse relationship. Scorecard looks for the presence of unpinned dependencies in Dockerfiles, shell scripts, and GitHub workflows. However, the tool does not check `package.json`, `package-lock.json` or `setup.py` files where npm and PyPI package developers lock their dependencies. Hence, “Pinned-Dependencies” scoring only involved GitHub-specific workflows, limiting the metric’s generalizability for npm and PyPI ecosystems. To verify the negative association of “Pinned-Dependencies" with other practices, we will need precise data from Scorecard tool that detects all pinned or unpinned dependencies in different ecosystems. Due to the necessity for additional research to establish the importance of “Pinned-Dependencies" by gathering reliable data, we have thus withdrawn “Pinned-Dependencies" as a suggested important practice.

Scorecard defined Maintained, Code Review, and Branch Protection as high-risk practices and Security Policy as a medium-risk practice. Zahan et al. [272] showed that both ecosystems have a gap in practicing Code-Review, Maintained, Branch-Protection, and Security-Policy practices. Although the current security trend shows practitioners do not employ these practices, our study indicates that practitioners adopting these practices will improve package security.

Maintained (high risk), Code Review (high risk), Branch Protection (high risk), and Security Policy (medium risk) are the most important practices that practitioners can adopt to improve package security outcomes by minimizing vulnerabilities.

7.3.2 Aggregate Security Score and Externally-reported Vulnerabilities:

Our model suggests that packages with more indicators of good security practices also had more reported vulnerabilities. Many factors are believed to increase the vulnerability count of packages. A possible explanation for the increase in reported vulnerability count while the security score increases could be that other open-source packages frequently use the selected packages. Since more clients utilize these packages, there is a higher likelihood that the package will be tested or attacked, and there will be more reported vulnerabilities. The average stats (Direct Dependents: 427, Transitive Dependents: 2,758, Downloads: 72M) of packages in our dataset revealed that other open-source packages frequently used these packages. The number of developers may also impact the number of vulnerabilities. For instance, Meneely et al. [139] empirically showed that projects with more developers have more vulnerabilities. However, from the OSV and Snyk databases, we can not collect such metadata to evaluate the relationship between package vulnerabilities and package popularity and the developer community.

Without exploring and controlling for confounding variables, such as package popularity, the findings are unlikely to provide certainty that the good security practices of these packages helped to find more vulnerabilities. We need to conduct a more thorough investigation to confirm the claim, taking into account the time it took to find and fix the vulnerabilities, the practices that were used to find them, and whether or not third-party users found the vulnerabilities while using the package rather than the package’s owner. The scarcity of vulnerability data could be another reason influencing our model. We utilized externally-reported reported vulnerabilities and not those found prior to release and by the development team.

Packages with increased good security practices also had increased reported vulnerability counts. Additional research is needed to determine if finding more vulnerabilities through good security practices is the cause of the increased vulnerability count or if the relationship is explained by other factors.

7.4 Challenges in available data

In this section, we discussed the challenges we found in explaining our findings. To improve model performance, future research replicating our study should consider these issues.

7.4.1 Scarcity of Vulnerability Data

Our findings raise the concern of scarce vulnerability data in OSS communities. We need more studies and accurate vulnerability data to evaluate the relationship between security practices and vulnerabilities as security outcomes, especially to comprehend how security practices influence vulnerability detection and prevention. A package may have more vulnerabilities than we found in OSV and Snyk vulnerability databases (Section 7.1.3). Dumidu et al. [261] showed that a large number of vulnerabilities that affected the Linux kernel (39.4%) and the MySQL database server (62.23%) were vulnerabilities found in bug reports but not listed in common vulnerability databases, such as the NVD. Elder et al. [62] also identified 1,047 vulnerabilities in OpenMRS applications using five vulnerability detection techniques and tools. However, at the time of their study, OpenMRS had 11 reported CVE vulnerabilities. These studies show that we acquire vulnerability data from OSV and Snyk databases hardly contain the accurate number of vulnerabilities as security outcomes.

Additionally, even within our dataset, we removed four packages as outliers because the distance from other packages in terms of vulnerability count was high. Out of 2,272 packages, 2,268 had an average vulnerability count of 2.07; these four packages had an average of 122 vulnerabilities. Even if these four packages had real-world vulnerabilities that were more prevalent than those of other packages, we were unable to use the data to build models because of their low percentage (0.2%) in the dataset. In Chapter 8, we will explore

7.4.2 Time

The multivariate nature of security practices significantly complicates its quantification and measurement [36]. To validate security practice and security outcomes relationships, we need to evaluate the time when security practices were implemented and the time when vulnerabilities were injected, identified, and fixed. The data for time to remediate an externally-reported vulnerability can be collected from the vulnerabilities database. At the time of the study, Scorecard only measures security practices from the last 90 days' activities, which does not verify the security practices before 90 days. For better clarity, we need to do a controlled study by recording time of security practices implication in a packages and detection of vulnerabilities. The scorecard team run a weekly Scorecards scan and publish the results in a BigQuery public dataset. OSV and Snyk also record the time of vulnerability detection and fix. Researchers could gain valuable insight from our proposed model by conducting a long-term control study on package security practice data and tracking vulnerabilities.

7.4.3 Unpredictability

Security metrics are a hard problem, especially in predicting vulnerabilities or assessing the effectiveness of counter measures [203, 36]. We should consider that the software security field has an inherently greater amount of unexpected variation. Due to a lack of precision and unknowable variables, industries are still exploring how to adopt the most effective security measures to eliminate the most number of vulnerabilities efficiently [62, 282, 196].

For instance, on July 20, 2022, Checkmarx security researchers warned people about a new supply chain attack strategy that uses fake commit metadata to give the appearance of legitimacy to malicious GitHub repositories [18]. Threat actors may modify the metadata in GitHub repositories to improve their reputation and increase the likelihood that application developers will choose them as reliable package owners. The researchers found it is possible to manipulate commit metadata, including timestamps, to make a repository appear older than it is or to show that credible contributors have been actively involved in its maintenance. This attack is an example of how frequently the software industry encounters new security threats. Attacks and attackers are unpredictable, as other recent supply chain attacks have also shown [84, 65]. Such unpredictability in software security often makes it challenging to interpret how security practices have affected software's security outcomes.

7.5 Threats to Validity

To our knowledge, our study is the first peer-reviewed paper to look at how software security practices impact security outcomes using OpenSSF Scorecard data. However, as stated by Basili et al. [22], drawing general conclusions from empirical studies in software engineering is difficult. Because the software security process depends on a potentially large number of relevant context variables, we only used the OpenSSF Scorecard tool to measure software security practices. Our study findings are dependent on the ruleset defined by the tool. For this reason, we cannot assume that the results of our study can be generalized to all ecosystems. Since Scorecard only operated on GitHub repositories at the time of the study, GitHub was the only platform we looked at. Scorecard has four practices (Dangerous-Workflow, Binary-Artifacts, Pinned-Dependencies, and Token-Permissions) that are GitHub workflow specific. Hence, our findings could vary if we used datasets outside of GitHub repositories.

7.6 Conclusion

We looked into how multiple security practices and their aggregate score are relate to the security outcome of software. First, our models identified Maintained, Code Review, Branch Protection and Security Policy as the most important feature practices in our dataset. However, low R^2 and `adj` R^2 values raise concerns about model performance beyond the available data. Second, packages with increased good security practices also had increased reported vulnerability counts, demonstrating the need confounding variable control and more vulnerability data to ascertain whether other factors explain the relationship. To overcome the limitations imposed by data scarcity and enhance our understanding of security outcomes, the study calls for exploring additional metrics beyond vulnerability count. Future studies should select security outcome metrics that cover a wide range of packages to measure the relationship. In Chapter 8, we explore different

security outcome metrics to understand whether the metrics can be used to explain the relationship between security practice and security outcome metrics.

CHAPTER

8

PRIORITIZATION OF SECURITY PRACTICES THROUGH SECURITY OUTCOME ANALYSIS

SSC security standards and frameworks (e.g. [238, 116, 237, 230, 140, 263]) have been proposed by industry and government agencies to help organizations adopt security practices that mitigate SSC security risks. However, historical experience demonstrates the risks of adopting security practices without empirical validation. For example, long password complexity requirements, such as mandating uppercase letters, numbers, and symbols, were long assumed to enhance security [13, 145]. However, empirical studies [53, 52, 130] later revealed that such policies often led users to create predictable passwords by encouraging reuse and minor modifications. Similar theoretical assumptions may persist in SSC security, where different practices are widely promoted without comprehensive empirical validation. Security success is often measured through adoption metrics—e.g., the number of teams that adopted code review or the number of vulnerabilities detected or fixed, rather than actual security outcomes. Therefore, little is known about the effectiveness of best practices in improving security. The absence of empirical evidence arises from multiple systemic challenges: widespread disagreement about best practices among experts [193, 192], poor evidence even where consensus exists [213, 239], and dependence on intuition and convention rather than empirical evidence.

The cybersecurity community's reliance on perceived common sense has created an information-poor environment characterized by conflicting guidance and expert disagreement about fundamental priorities [193, 192, 151]. Meanwhile, organizations have limited budgets, time, and resources [201, 99], yet practitioners currently lack the empirical evidence needed to prioritize security practices that mitigate security risk. Therefore, developing empirical research to evaluate the security outcome of recommended practices is both

a research priority and a fundamental prerequisite for evidence-based security decision-making [207, 151]. ***The goal of this study is to assist practitioners and policymakers in making informed decisions on which security practices to adopt by evaluating the relationship between software security practices and security outcome metrics.*** We address the following research questions (RQs):

- **RQ1**: Is higher adoption of security practices associated with measurable improvements in security outcomes?
- **RQ2**: Which security practices are most strongly associated with security outcomes?

To that end, the study investigated the relationship between publicly available data on security practice adoption and security outcomes metrics. For large-scale analysis, we focus on security practice adoption that can be measured automatically. We selected the OpenSSF Scorecard tool [206] that automatically collects 18 metrics on the adoption of security practices in GitHub repositories. The Scorecard metrics are valuable for collecting ecosystem-scale data, as well as for evaluating the security posture of project dependencies. Examples of security practices that can be measured by Scorecard include maintainers conducting code reviews before merging pull requests, using a static analyzer tool, and others. For each metric, the Scorecard tool assigns a score of 0-10 based on pre-defined criteria. The tool also computes an *aggregate security score* as a risk-weighted average of the individual security practice metrics, weighted by risk. For security outcomes, we selected project-based outcome metrics, where a project is defined as a software package that includes all components in its dependency graph. Specifically, the number of open vulnerabilities (Vul_Count), the mean time to remediate (MTTR) vulnerable and outdated dependencies, and the mean time to update (MTTU) outdated dependencies were used.

To address RQ1, we investigate the aggregate adoption of security practices and their impact on security outcomes. We developed a regression model to assess the relationship using Scorecard *aggregate security score* and security outcome metrics. Additionally, we applied Propensity Score Matching (PSM), a statistical matching technique used in observational studies, to estimate causal effects and mitigate the potential confounding effects based on repository characteristics. For RQ2, we employed a regression model and feature importance-ranked techniques to identify which individual security practices have the strongest associations with security outcomes. To further validate these findings, we used PSM to estimate the effect of individual security practices on security outcomes, isolating their impact while accounting for differences in repository characteristics. The study has the following contributions:

- Introduction of project-based security outcome metrics.
- Application of industry-standard metrics, MTTR and MTTU, to assess security outcomes.
- A proposed model that can be used to explore the relationship between security practice and security outcome metrics.
- Empirical evidence on how security practices impact the security outcome of npm packages.

The chapter is organized as follows: Section 8.1 discusses the design of our study; Section 8.1.1 discusses the metrics used in our study; and Section 8.1.2 discusses our dataset construction steps. Section 8.2, Section 8.3 and Section 8.4 describe the data processing, methodology, and results. We close with a discussion of our study (Section 8.5) and related work in Section 8.6.

8.1 Experiment design

In this section, we outline the experimental design and dataset used to investigate our research questions. To answer RQ1 and RQ2, we define Scorecard security practices metrics and aggregate security score as predictor variables and security outcome metrics as target variables. We also incorporate control variables to account for potential confounding project characteristics. We leveraged machine learning and causal analysis techniques to examine the relationship between practices and outcomes. Our analysis is based on data collected from npm packages and their corresponding GitHub repositories, using dependency graphs to define project scope. The dataset construction integrates multiple data sources and applies systematic inclusion and exclusion criteria to ensure reliability and reproducibility. Section 8.1.1 details the selection of metrics used in our study, while Section 8.1.2 describes how we collected the dataset.

8.1.1 Metrics Utilized in the Study

In this section, we discuss our metrics selection process, including security practices, security outcomes, and control variables.

Security Practice Metrics

To assess the adoption of security practices, the OpenSSF Scorecard tool [206] was employed. The tool automatically evaluates a set of security practices implemented in GitHub repositories, based on observable project metadata and activity. Table 8.1 contains the list of 18 security practices measured by Scorecard. The tool also computes an *aggregate security score* to assess the overall security of a repository, computed as a risk-weighted average of individual security practice metrics. Scorecard was selected due to its widespread adoption and recognition in both academic and industry contexts. In academic research, Scorecard has been the subject of empirical evaluations, including SSC risk measurement [274, 149, 32, 200, 215, 118, 264, 198], surveys on practitioners' perspectives on Scorecard [63, 198], and qualitative studies [272] examining its applicability and effectiveness. In industry, Scorecard has been adopted as part of standard security frameworks and requirements [263, 167] and has also been used in reports by organizations, such as Sonatype and Veracode, to assess the state of open-source software ecosystems [227, 64]. Adoption by major projects such as Kubernetes, Node.js, and Apache, along with integration into platforms like Ortelius [172] and Deps.dev [174], also highlights wide adoption.

Security Outcome Metrics

A comprehensive understanding of security outcome metrics remains an open research challenge. Although defining and validating such metrics is inherently challenging, our work represents a step toward advancing that goal. In this study, we selected project-level outcome metrics because prior studies [274] showed that only a small fraction of packages (0.1% of the npm ecosystem) have reported vulnerabilities at the individual package level. Limited availability of vulnerability data prevents models from learning meaningful relationships between security practices and security outcomes. Therefore, we leveraged three project-level outcome metrics, where a project is defined as an individual package and its dependencies. The security risk of a project decreases when a vulnerable component is replaced with a fixed one, providing a measurable indication of

improved security outcomes, reflected in fewer open vulnerabilities and reduced exposure windows due to faster response times.

- **Vul_Count**: A measure quantifying the number of publicly disclosed or reported open security vulnerabilities within a package and its full dependency tree, including both direct and transitive dependencies.
- **MTTR**: A measure of the average aggregated time a package uses outdated and vulnerable direct dependencies in its lifetime [189]. MTTR quantifies how long vulnerable dependencies remain outdated after a new version with a security fix has been released.
- **MTTU**: A measure of the average aggregated time a package uses outdated direct dependencies in its lifetime [189]. MTTU quantifies how long dependencies remain outdated after a new version has been released.

The number of vulnerabilities is widely used in academic research, industry, and policy as outcome metrics [237, 42, 148, 202, 146, 274]. While MTTR and MTTU are widely used industry metrics, our work is novel in applying them as outcome measures for security responsiveness. Faster updates reduce exposure windows, limiting attackers' opportunity to exploit disclosed weaknesses. We draw inspiration from the DevOps Research and Assessment (DORA) framework [54]. DORA is one of the largest research studies in software engineering research, analyzing survey responses from over 31,000 professionals across six years (researched in the years 2014 through 2019) [74]. DORA research identifies which software engineering practices, including security practices, improve software delivery performance. One of DORA's outcome metrics is a responsiveness metric that measures how long it takes a project to respond and deploy the corresponding updates. Research conducted in 2022 by the Google DORA team [94] specifically investigated SSC practices derived from the Secure Software Development Framework (SSDF) [230] and Supply-chain Levels for Software Artifacts (SLSA) [217], and examined their impact on outcomes. The research has shown that SSC practices are positively associated with faster response and deployment times, and practitioners also anticipate a lower chance of security vulnerabilities. Motivated by DORA metrics, we hypothesize that *teams that adopt good security practices are also more likely to be more responsive, as measured by MTTR and MTTU*. We used MTTU because ecosystem-wide MTTR data is often unavailable (only 15% of our dataset; see section 8.1.2). While MTTR focuses on the response to security updates, MTTU measures responsiveness to any upstream updates and is available for all packages. Prior work [189] also found a moderate correlation between MTTR and MTTU, suggesting that MTTU can serve as a proxy when MTTR is unavailable.

Control Variables

Control variables are properties that researchers hold constant in an experiment to understand the true relationships between the predictor and target variables [79]. We included six variables to measure repository characteristics based on two criteria: (1) alignment with prior use of control variables on software quality and practices work [143, 281, 191, 173, 244, 132, 114], and (2) maintaining model interpretability with a minimal yet effective set of metrics to reduce data collection and cost. We controlled for six repository characteristic metrics grouped by their hypothesized influence. Contributor count (Contributors_CT) and commit staleness reflect team capacity and development activity, which may affect the implementation and maintenance of security practices. Download count and direct dependencies capture project adoption and external exposure, potentially influencing both scrutiny and security risk. Repository age and size represent project maturity and

complexity, which may impact maintenance effort and security performance. We excluded highly correlated variables such as forks, stars, and dependents due to redundancy and because they did not improve model performance.

Table 8.1: Distribution of Scorecard Security Practices Metrics

For columns with scores of -1 and 0, the number in parentheses indicates the raw count.

Security Practice	-1 %	0%	1–10%	Median	Mean	Std
Aggregated Score	0.0	0.0 (47)	99.8	2.0	2.4	1.2
Metrics Used in Model						
Binary-Artifacts	0.0	0.2	99.8	10.0	10.0	0.6
Branch-Protection	0.0 (3)	85.9	14.1	0.0	0.7	1.9
Code Review	0.4	75.4	24.2	0.0	0.8	1.9
License	0.0	30.9	69.1	10.0	6.9	4.6
Maintained	0.0	86.7	13.3	0.0	0.9	2.5
Pinned-Dependencies	72.2	18.0	9.8	-1.0	-0.4	1.3
Dependency-Update-Tool	0.0 (65)	81.3	18.7	0.0	1.9	3.9
Contributors	0.0	36.7	63.3	3.0	4.6	4.3
CI-Tests	50.3	41.3	8.4	-1.0	0.0	2.2
SAST	0.0 (23)	96.9	3.1	0.0	0.3	1.4
Security-Policy	0.0 (2)	93.2	6.8	0.0	0.6	2.4
Metrics excluded from the model due to high correlation						
Token-Permissions	73.2	25.6	1.2	-1.0	-0.6	1.2
Dangerous-Workflow	73.2	0.3	26.6	-1.0	1.9	4.9
Metrics excluded from the model due to high -1 or 0 value						
Fuzzing	0.0 (2)	99.8	0.2	0.0	0.0	0.4
CII-Best-Practices	0.0	99.9	0.1	0.0	0.0	0.1
Signed-Releases	97.7	2.3	0.0	-1.0	-1.0	0.2
Packaging	97.5	0.0	2.5	-1.0	-0.7	1.7
Metrics excluded from the model due to the use of Vul_Count as outcome metrics						
Vulnerabilities	0.0	32.2	67.8	9.0	6.1	4.5

8.1.2 Dataset Source and Construction

Section 8.1.2 discusses different sources used to construct our dataset. Our dataset is available in Figshare[1].

Data Sources

To collect data, we leveraged multiple sources. We selected the npm ecosystem for this study because npm is the largest open-source ecosystem, comprising 37 million package versions, with 74% of open-source codebases using JavaScript [232, 229]. Scorecard analyzes packages with source repositories hosted on GitHub [171] and periodically evaluates popular open-source projects. Within Scorecard's dataset of selected popular

[1] `https://figshare.com/s/e74f94dfe1be6a90c4d1?file=54861686`

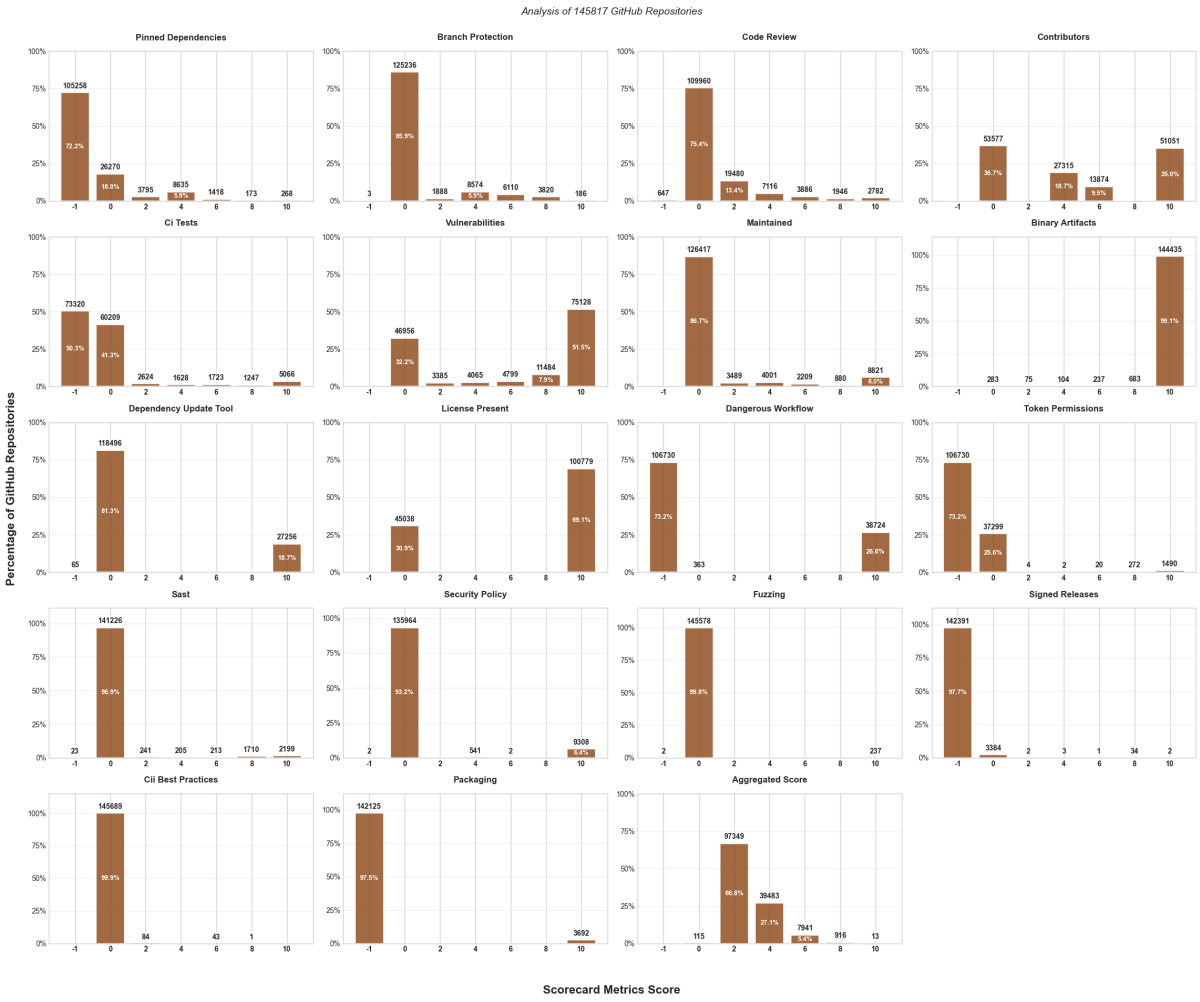

Figure 8.1: Scorecard Metrics Distribution

packages, 64% are npm packages. Additionally, our security outcome metrics require analyzing project dependencies, specifically counting vulnerabilities in dependencies and measuring outdated and vulnerable dependency update patterns for MTTR and MTTU. To support our analysis, we collected the dependency graph of npm packages and mapped packages to their corresponding GitHub repositories using deps.dev [174]. Deps.dev, developed and hosted by Google, provides comprehensive insights into OSS, including source code locations, dependency graphs, package metadata, licenses, releases, and vulnerabilities. Another platform used in our study is Open Source Vulnerabilities (OSV) [93]. The OSV platform aggregates vulnerabilities from various security advisories. Scorecard and Deps.dev depend on OSV to collect vulnerability information for projects and their dependencies. We also used the Libraries.io platform [134], which collects publicly available OSS metadata scraped from the internet, including GitHub.

Package Inclusion and Exclusion Criteria

To construct our dataset, we systematically filtered, cleaned, and processed npm packages based on dependency relationships, repository availability, and data availability. The dataset construction followed the following steps:

- **Initial Dataset:** We began with a total of 2,119,044 unique npm packages extracted from the Deps.dev BigQuery dataset [8]. This dataset contains the mapping between npm packages and GitHub repositories and captures all dependency edges within npm.
- **Packages with GitHub Repositories:** Of these, 877,312 packages were excluded because they lacked valid GitHub repository information, either because they were hosted outside GitHub or had missing repository URLs. Mapping packages to GitHub repositories was necessary because Scorecard runs directly on GitHub repositories. After this filtering step, 1,241,732 packages remained that could be successfully mapped to GitHub.
- **Packages with Dependencies and Dependents:** Further filtering identified 1,159,891 packages with at least one declared dependency, ensuring relevance to our security outcome metrics. Additionally, we are primarily interested in packages that practitioners may use; hence, we refined the dataset further to include only packages with at least one dependent, reducing the dataset to 264,413 packages.
- **Ensuring Unique Repositories:** Scorecard generates metric scores based on security practice adoption at the repository level rather than for individual packages. While we collected vulnerability data at the repository level (Section 8.1.2), MTTR and MTTU can only be measured at the package level. Consequently, if a repository contains multiple packages, all packages share the same Scorecard score and vulnerability data but may have different MTTR and MTTU values. Such inconsistency leads to duplicate rows with identical Scorecard metrics but differing security outcome metrics, which could mislead our machine learning model. To resolve this issue, we filtered the dataset to include only repositories that map to a single package. After applying this filtering step, we retained 172,999 unique repositories.

OpenSSF Scorecard Data

Once we identified the set of 172,999 unique npm GitHub repositories, we collected OpenSSF Scorecard scores for each repository associated with a package. Our primary data source was the OpenSSF Scorecard dataset in Google BigQuery [9]. However, Scorecard only evaluates a subset of repositories, specifically popular packages and those explicitly requested by practitioners for inclusion. Hence, many repositories in our dataset were not included in the original Scorecard dataset. To address these gaps, we ran Scorecard locally for any repositories missing from the BigQuery dataset, using the same configuration applied to populate BigQuery. After merging data from Google BigQuery and our local runs, we obtained Scorecard scores for 145,817 out of 172,999 repositories. The remaining repositories did not generate a score using Scorecard. Upon manual verification of a random sample of 100 repositories and their Scorecard output, we found that Scorecard failed to produce scores due to HTTP 404 errors. The primary reason for the lack of scores was that the repositories were no longer accessible via public GitHub URLs. The repositories were either private or had been removed from GitHub.

Metrics exclusion: We began by collecting all 18 metrics provided by Scorecard. However, due to limited adoption and Scorecard measuring limitations, we excluded Fuzzing, CII-Best-Practices, Signed-Releases, and Packaging from our analysis (Table 8.1 and Figure 8.1)). In Table 8.2, we show the difference between different Scorecard score interpretations. Fuzzing and CII Best Practices were excluded because 99.8% of repositories scored 0, indicating minimal adoption of Scorecard recommendations across ecosystems. Signed Releases and Packaging were excluded, as almost 98% of the repositories received a -1 score due to not having any signed release or a package release workflow in GitHub. Prior research [272] found that practitioners primarily used package registries for releases rather than GitHub, and no improvements in Scorecard's methodology addressing this limitation were observed. We also removed the vulnerability metric as a practice because we use Vul_Count as a dependent variable. Thus, we removed five metrics from the analysis.

Table 8.2: Interpretation of Score Values for Each Scorecard Metric

Metric	**-1**	**0**	**1–10**
Binary Artifacts	–	Large number of binary files in the repository	Fewer binary files result in higher scores
Branch Protection	Internal error	No protection rules enabled for main branches	Score increases with stricter and more comprehensive branch protection rules
Code Review	Found no human activity in the last 30 changesets	No code reviews on changes before merging	Score reflects the proportion of changes that went through formal code review
License	–	No valid license file is found	Score depends on whether the license exists, is top-level, and is OSI/FSF approved
Maintained	Project is less than 90 days old	Project is archived	Score reflects how active the project is in terms of commits and issue updates over the past 90 days
Pinned Dependencies	No dependencies found	Dependencies are not pinned or locked	Score reflects the extent to which dependencies are pinned or locked in configuration
Dependency Update Tool	Internal error	No automated update tool is used	Tool such as Dependabot, Renovate, or others is actively used

Continued on next page

Table 8.2 (continued from previous page)

Metric	-1	0	1–10
Contributors	–	All contributors come from the same organization	Score increases with diversity of contributor organizations
CI Tests	No pull request found	No automated tests run on pull requests	Score reflects how many pull requests are covered by continuous integration tests
SAST	Internal error	No static analysis tool found	Score reflects presence and configuration of static analysis tools like SonarQube or GitHub CodeQL
Security Policy	Internal error	No security policy file is present	Score reflects presence of contact info, free-form guidance, and disclosure instructions in the policy file
Token Permissions	No Github tokens found	Top-level and Job-level permissions are write or undeclared	Score decreases as more sensitive permissions are enabled by default in workflows
Dangerous Workflow	No workflows found	Dangerous or insecure workflow patterns detected	No dangerous patterns are found in GitHub workflows
Fuzzing	Internal error	No fuzzing tools detected in the project	Recognized fuzzing tools are used in the project
CII-Best-Practices	Internal error	No badge or badge is incomplete	Score reflects the badge status (in progress, passing, silver, or gold)
Signed Releases	No releases found	No signed or verifiable release artifacts	More signed or verifiable releases result in a higher score
Packaging	No GitHub/GitLab publishing workflow detected	No packaging workflows used in the project	Project uses GitHub-based packaging workflows to publish artifacts
Vulnerabilities	Internal error	High number of unresolved vulnerabilities	Fewer vulnerabilities result in higher scores

Security Outcome Metrics Data

We collected our security outcome metrics for 145,817 npm packages with Scorecard data. Here, we describe our outcome metrics data collection process.

Vul_Count metric: The Scorecard Vulnerability metric measures whether the project has open, unfixed vulnerabilities in its codebase or its dependencies using the OSV [93] service. Initially, we used the Scorecard tool to collect the total vulnerability count. The Scorecard tool normalizes the vulnerability score, where a score of 10 indicates no vulnerabilities, 9 corresponds to one vulnerability, and 0 represents more than nine vulnerabilities. To obtain the exact vulnerability count, we first denormalized the score. For cases where the vulnerability score was 0 (indicating more than nine vulnerabilities), we ran the vulnerability metric separately using Scorecard to retrieve the raw, unnormalized data and used the total number of vulnerabilities in the model.

MTTR and MTTU metric: In this study, we adopt the algorithms proposed by Rahman et al. [189] for computing MTTR and MTTU. Following their methodology, we contacted the Googlee deps.dev team and considered deps.dev as our primary data source to collect comprehensive package versions, dependency versions and their release information for our dataset. Additionally, we retrieved CVE data for selected packages and their direct dependencies from OSV [93]. Note that while MTTU captures all dependency updates, MTTR specifically accounts for updates addressing vulnerable dependencies. MTTR needs a vulnerability in direct dependency, then the availability of a new fix version, and the adoption time to adopt the new fix version by the dependent. While we successfully collected Vul_Count and MTTU data for 145,817 packages, MTTR data was only available for 22,412 packages (15% of total dataset).

Control Variable Metrics Data

We collected control variable data from different sources. GitHub contribution count and package size are collected from Libraries.io [134]. Then we got the number of downloads of a package in the past 12 months, which we collected from the public npm API[2]. We calculated the repository age (in years) and commit staleness dates using the GitHub REST API [87]. The number of dependency information is collected from Deps.dev [8].

8.2 Data Preprocess

Before constructing the machine learning (ML) modeling to answer RQs, we performed data preprocessing steps to handle missing values, ensure data integrity, and assess statistical properties. Section 8.2.1 covers feature decomposition and imputation techniques to manage missing values. Section 8.2.2 discusses correlation and multicollinearity in our dataset.

8.2.1 Feature Decomposition and Imputation

Our manual analysis found that Scorecard assigned -1 in two cases: (1) when there was no conclusive evidence of security practice adoption; and (2) due to internal runtime errors. Since not all metrics within a repository had a -1 score (Table 8.1), excluding an entire repository due to a -1 in any one of the 18 metrics would result in unnecessary data loss. To handle missing data effectively, we implemented a missing data

[2] `https://api.npmjs.org/downloads/point/{period}[/{package}]`

pattern decomposition strategy based on Scorecard score reasoning. For high-missing features CI-Tests, Pinned-Dependencies, Token-Permissions, and Dangerous-Workflow (each with >50% -1 scores), a score of -1 indicates that the repository lacks the relevant configuration. Here, -1 for CI-Tests indicates no pull requests to confirm CI workflows were present; for Pinned-Dependencies, it suggests the absence of declared dependencies in GitHub; for Token-Permissions, it reflects that no GitHub token settings were configured; and for Dangerous-Workflow, -1 means no GitHub Actions workflows were found to test workflow patterns. To preserve the distinction, we decomposed each feature into two components: a binary indicator of relevant configuration presence (1 for values 0-10, 0 for -1) and scores that capture adoption effectiveness (keeping original discrete values 1-10, with both -1 and 0 scores mapped to 0). For low-missing features (<1% -1 values), where -1 was due to internal errors of Scorecard, we applied median imputation [205] to maintain data completeness while minimizing distortion.

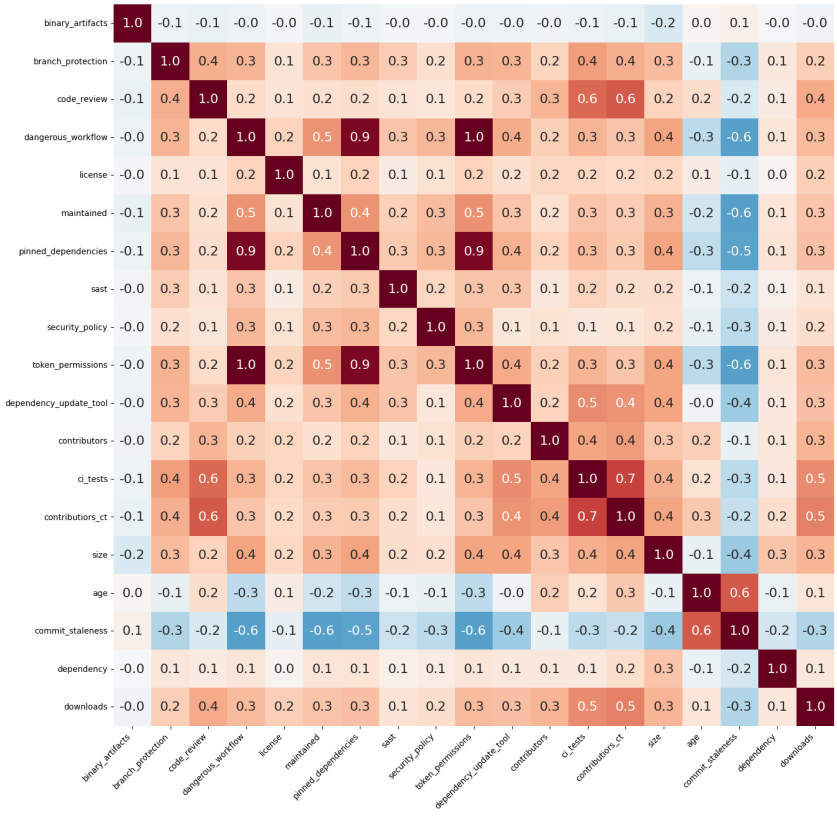

Figure 8.2: Spearman Correlation between Predictors

8.2.2 Correlation Analysis

To evaluate dataset normality, we applied the Henze-Zirkler multivariate normality test [104] separately to the RQ1 and RQ2 datasets. The RQ1 dataset includes aggregate scorecard scores, the control Variables, Vul_Count, MTTR, and MTTU. The RQ2 dataset contains 13 Scorecard practice metrics, control variables, and

Vul_Count, MTTR, and MTTU. Both datasets failed the normality test ($p < 0.05$), indicating deviations from a normal distribution. To further assess skewness, we flagged features with $|skewness| > 1$ for potential transformation [102]. Based on skewness, we applied a $\log(x + 1)$ transformation to highly skewed control variables (GitHub contributors, repository size, DependencyCount, downloads) to mitigate skewness. Post-transformation, Q-Q plots, histograms, and statistical tests confirmed improved distribution symmetry. We did not apply transformation to the practice or outcome metrics to preserve their original scale and interoperability. Since residual normality is important for regression analysis, we conducted D'Agostino-Pearson tests [119] on regression residuals. Results showed that residuals remained non-normally distributed ($p < 0.05$).

Given the non-normal distribution of our dataset, we proceeded with the Spearman correlation [276], a nonparametric method suitable for capturing monotonic relationships in non-normal data. We conducted a correlation analysis for predictor variables (security practice and control variables) and target variables (security outcome metrics). Figure 8.2 presents the Spearman correlation heatmap for predictor variables. The highest correlations (1) were observed among Dangerous-Workflow, Pinned-Dependencies, and Token-Permissions. To avoid multicollinearity and redundancy in the model [98], we removed Dangerous-Workflow and Token-Permissions, retaining only Pinned-Dependencies as a representative feature. The correlation between target variables reveals a strong positive correlation (0.7) between MTTR and MTTU, aligning with our expectation of potential overlap [189]. In contrast, the Vul_Count metric exhibits a weak negative correlation with MTTR (-0.3) and MTTU (-0.4). Hence, we confirmed that MTTR and MTTU cannot be used as target variables in the same model due to their strong correlation, since including both would introduce redundancy, bias, and multicollinearity. Additionally, we also computed the Variance Inflation Factor (VIF) [178] for all predictor variables to confirm multicollinearity. For the VIF computation, we did not find any feature that exhibited a VIF score greater than 5, indicating no multicollinearity.

8.3 RQ1: Overall security posture and its security outcomes

In this section, we discuss our **RQ1: Is higher adoption of security practices associated with measurable improvements in security outcomes?** Section 8.3.1 discusses the methodology of RQ1, and Section 8.3.2 discusses the results.

8.3.1 RQ1: Methodology

For RQ1, we evaluate the relationship between the aggregated Scorecard score of npm package repositories and control variables as predictors, and security outcome metrics as the target variable. To visualize the relationship between aggregated Scorecard scores and security outcomes, we first binned repositories by aggregated Scorecard scores and analyzed mean security outcomes within each bin (Figure 8.3). Next, we applied regression analysis to quantify the association of the aggregated Scorecard score with outcomes. However, regression analysis does not establish causality. We implemented a Propensity Score Matching (PSM) framework with caliper-based matching, covariate balance diagnostics, and segment-level analysis to estimate the causal effect of security practice adoption on outcome metrics, controlling for project characteristics.

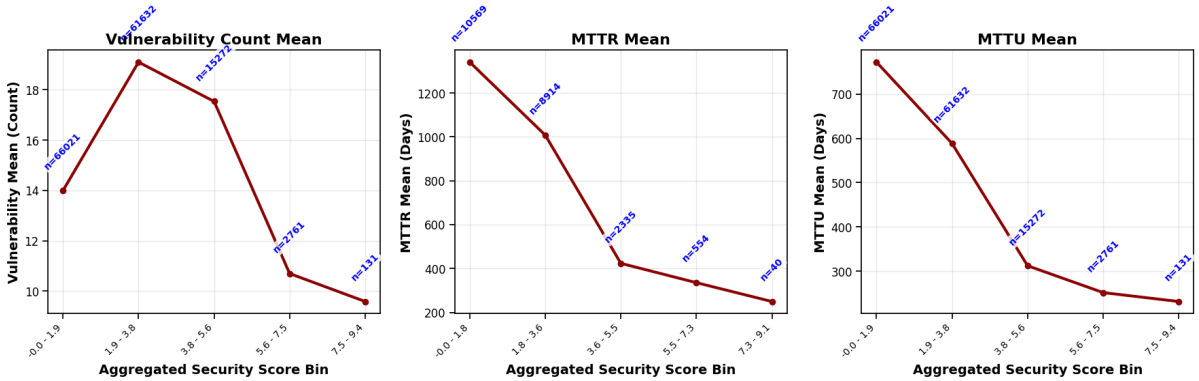

Figure 8.3: Security Outcome Trends and npm Repository Distribution Across Aggregated Scorecard Score Bins

Data Expolation

To visualize initial relationships, we binned repositories by aggregated Scorecard scores into five equal-width bins and calculated mean security outcomes within each bin. The binning approach provides an intuitive view of how security outcomes trend across different score ranges before applying statistical modeling.

Regression Model:

At this stage, we focus on interpreting how security outcomes change per unit change in the aggregated Scorecard score (e.g., a coefficient of -0.1 implies a 10% decrease in the outcome for every one-unit increase in aggregated Scorecard score). We selected Generalized Linear Models (GLMs) [152], given the non-normal distribution of our dataset, and our target variables are count data. GLMs are explicitly designed for count data and provide interpretable coefficients. Coefficients quantify the expected change in the target variable for a unit increase in a predictor. To identify the best-fitting GLM, we systematically evaluated dataset characteristics using standard diagnostic tests: overdispersion (variance < mean) and zero-inflation. We began with Poisson Regression [103] as our baseline model and found significant overdispersion across all outcomes, with zero-inflation present only in Vul_Count. We applied Negative Binomial (NB) Regression [105] to address overdispersion and attempted Zero-Inflated Negative Binomial (ZINB) [194] for Vul_Count to handle both issues simultaneously. However, ZINB models encountered numerical convergence failures with precision loss warnings during maximum likelihood estimation. Therefore, we applied NB regression across all outcomes, providing stable parameter estimation while adequately addressing overdispersion.

High vs. Low Aggregated Scorecard Scores: Comparing Similar Repositories

To move beyond correlational evidence and establish causality, we employed PSM to simulate experimental conditions by comparing high and low-scoring repositories with similar project characteristics. PSM [19] is a statistical technique that simulates the conditions of a randomized experiment by matching repositories with similar characteristics but differing in their level of security practice adoption. Similar project characteristics refer to repositories matched on the same covariates used as control variables in section 8.1.1.

Repositories are classified as “treated" if their aggregated scores fall within the top quartile, with the

remaining projects serving as a control group. We tested multiple treatment thresholds (median, tertile, quartile) and selected the top quartile (≥ 3 on a 1-10 scale) to ensure sufficient contrast between treatment and control groups while maintaining adequate sample sizes. Using logistic regression on covariates that influence both aggregated scores and security outcomes, we estimated each repository's propensity score (probability of having a high aggregated score given its characteristics), then matched treated and control repositories with similar propensities. We then applied nearest-neighbor matching with a 0.1 standard deviation caliper to pair treated and control repositories with similar propensities. We validated matching quality using standardized mean differences (SMD) and calculated Average Treatment Effects (ATE) through difference-in-means with bootstrap confidence intervals (CI) to quantify uncertainty. We complement this analysis with NB on post-matched samples (containing only the matched treated and control repositories).

Table 8.3: Segment-specific ATEs and percentage impacts. Strongest tercile per variable shown.

Outcome	Controls	Segment	ATE	Impact
Vul_Count	Contributors_CT	Many	-8.2	28.9%
	Age	Mature	-11.0	42.2%
	Commit Staleness	Fresh	-9.4	47.5%
	Downloads	High	-9.3	35.9%
	Size	Large	-8.4	27.7%
	Dependency	Medium-Dep	-9.8	37.0%
MTTR	Contributors_CT	Many	-350.1	41.7%
	Age	Mature	-429.8	37.5%
	Commit Staleness	Fresh	-244.8	37.3%
	Downloads	High	-269.5	40.5%
	Size	Large	-315.0	40.2%
	Dependency	Few	-245.7	28.1%
MTTU	Contributors_CT	Many	-87.2	19.1%
	Age	Mature	-86.0	12.8%
	Size	Small	146.3	40.0%
	Commit Staleness	Stale	212.1	23.5%
	Downloads	Few	146.1	38.5%
	Dependency	High-Dep	-81.0	20.6%

Impact Across Segments of Each Repository Characteristic

While our PSM analysis establishes an overall causal effect by controlling for repository characteristics, it estimates one average treatment effect without examining how this effect varies across different repository types. For instance, the security benefits of higher aggregated scores might be stronger for large repositories compared to small ones. To examine whether this effect differs in magnitude across different levels of repository characteristics, we conducted post-hoc heterogeneity analyses on the matched sample. We stratified the matched repositories into terciles based on each control variable using quantile-based binning

and calculated segment-specific ATEs, comparing high and low scorecard scores within each segment. This approach identifies which repository contexts benefit most from higher aggregated scores. We used bootstrap CI to quantify uncertainty around segment-specific ATEs.

8.3.2 RQ1: Result

Data Exploration: Figure 8.3 shows a sharp decline in security outcomes (indicating improvement) as aggregated Scorecard scores increase. The highest bin (7.3–9.1) reflects the top-scoring repositories, as no project achieved a perfect score. However, the plot only reflects an overall trend and does not account for confounding factors and associations.

Quantitative Confirmation: Table 8.5 confirms visual trends (Figure 8.3) through three analytical approaches: full-sample regression, PSM post-matched sample regression, and PSM causal analysis. The columns show full-sample NB coefficients, post-matching NB coefficients, and post-matching ATEs with 95% confidence intervals. For Vul_Count, both full-sample (–0.20) and post-matching (–0.26) NB models show negative coefficients, with ATE indicating 5.2 fewer vulnerabilities. MTTR demonstrates consistent improvements: full-sample (–0.05), post-matching (–0.16), and ATE (–216.8 days). For MTTU, while the full-sample model shows a positive coefficient (0.03), post-matching (–0.04) and ATE (–52.3 days) reveal faster updates, suggesting confounding factors mask the relationship.

Repository Context-Dependent Effects: While our PSM analysis established overall causal effects, we conducted segment-specific analyses to understand where these effects are strongest. Table 8.3 presents the most pronounced effects within each repository characteristic, with the "Segment" column identifying the specific subgroup and the "Impact" column quantifying percentage improvements relative to the control baseline within that segment.

Across all segments, **Vul_Count** consistently shows positive security outcomes (i.e., negative ATE values), with improvements ranging from 27.7% to 47.5%. The most substantial reductions in open vulnerabilities are observed in mature repositories with fresh commit staleness (47.5% and 42.2%, respectively), indicating that actively maintained and mature repositories with high aggregated scores tend to have fewer vulnerabilities. Medium-dependency group repositories show the greatest vulnerability reduction, highlighting the effect of dependency complexity. **MTTR** improvements are also observed across all segments. Mature repositories have the largest absolute improvement (429.8 days faster), followed by those with many contributors (350.1 days). Repositories with fewer dependencies and fresh commit staleness show meaningful reductions as well (-245.7 and -244.8 days). Then, repositories with higher download counts tend to have fewer Vul_-Count and faster MTTR. **MTTU** reveals the most heterogeneous pattern, with some segments showing improvements while others exhibit deterioration. Small repositories by size and download and repositories with stale commit activity show increases in MTTU, suggesting that top quartile aggregated score repositories that are small, less downloaded, or less active have slower dependency updates compared to low Scorecard score repositories in the same segment category. Conversely, repositories with many contributors, mature age, and high dependency counts demonstrate faster updates.

Summary of RQ1 Findings:

- **Overall Effects:** Higher aggregated Scorecard scores improve security outcomes, with repositories achieving 5.2 fewer vulnerabilities, 216.8 days faster MTTR, and 52.3 days faster MTTU.
- **Repository Context Matters:** The adoption of aggregated security practices improves security outcomes.

The segment-specific analysis reveals nuanced patterns in how repository characteristics influence security practice effectiveness.

- **Organizational Capacity:** Mature repositories with many contributors consistently show better outcomes, suggesting that well-staffed and mature projects with high adoption of security practices benefit from improved security outcomes.
- **Project size, activity, and downloads**: Project size, recent commit activity, and download volume are associated with improvements in Vul_Count and MTTR. However, their absence creates challenges (i.e., smaller, inactive, or less-downloaded projects) and correlates with slower MTTU, highlighting how inactivity and smaller projects with fewer downloads can hinder general updates.
- **Dependency Complexity:** Different dependency levels excel in different areas. Medium-dependency projects have the fewest open vulnerabilities, likely balancing manageable complexity. Few-dependency projects remediate issues fastest, benefiting from simpler, more transparent dependency trees. Large-dependency projects excel at general updates, likely leveraging automation to manage scale effectively.

While selecting dependencies, organizations should prioritize mature, actively maintained large projects with many contributors, fewer dependencies, and high downloaded packages, as our analysis showed that such projects have fewer vulnerabilities and faster MTTR due to better security performance. Conversely, smaller, inactive packages with fewer downloads require enhanced monitoring and proactive security assessments to compensate for slower MTTU.

Table 8.4: R^2, Security practice Importance Ranked and PSM Insights. Positive ATE indicates an increased outcome (worse), negative ATE indicates a reduced outcome (better).

Outcome	R^2 for RF	Ranked Security Practices	PSM Effect (ATE direction)
Vul_-Count	0.39	Contributors, Code Review, License, Pinned-Dependencies, Branch Protection, CI-Tests, Dependency-Update-Tool	Code Review (+), CI-Tests (mixed), Branch Protection (+), Dependency-Update-Tool (+), Others (-)
MTTR	0.50	Contributors, Code Review, License, CI-Tests, Pinned-Dependencies	Code Review (+), Others (-)
MTTU	0.32	Contributors, License, Code Review, CI-Tests, Pinned-Dependencies	Code Review(+), CI-Tests (mixed), Others (-)

Table 8.5: Combined GLM and PSM analysis for RQ1

Outcome	NB Coef	NB Coef (PSM)	PSM ATE (95% CI)
Vul_Count	-0.20	-0.26	-5.2 [-5.7, -4.7]
MTTR	-0.05	-0.16	-216.8 [-241.8,-193.2]
MTTU	0.03	-0.04	-52.3 [-60.2, -44.8]

8.4 RQ2: Impact of individual security practices and outcomes

In this section, we discussed our **RQ2: Which security practices are most strongly associated with security outcomes?** Section 8.4.1 discusses the methodology for RQ2, and Section 8.4.2 discusses the result.

8.4.1 RQ2: Methodology

For RQ2, we investigate the association between individual security practices and security outcomes. Unlike RQ1, where we analyzed aggregated Scorecard scores as a predictor, for RQ2, we analyzed eleven security practices as predictors after exclusion (Section 8.1.2 and 8.2.2) alongside six control variables. Due to the large number of predictors, non-normal distribution, and non-linear relationships (Section 8.2), we selected a tree-based regression model suited for handling complex, high-dimensional data. We also conducted the feature importance ranked analysis to identify the security practices that most strongly contribute to predicting security outcomes. We performed causal analysis to explore potential causal relationships since regression and feature importance identify associations but do not establish causality.

Regression Model:

We employed Random Forest (RF) regression due to its robustness to non-linear relationships and ability to handle high-dimensional data without overfitting. Following preprocessing (section 8.2.1), the dataset was split into training (70%) and testing (30%) subsets using random sampling. The model was trained on the training dataset using hyperparameters (n_estimators=100, random_state=42). To ensure robust performance estimation and reduce bias, five-fold cross-validation was applied exclusively to the training data, calculating R^2 scores(Table 8.4).

Security Practice Ranking

: Feature importance was quantified using Gini Importance, extracted directly from the trained Random Forest model's feature importances attribute, which measures each feature's contribution to reducing node impurity across all decision trees in the ensemble. Features were ranked by their grouped importance scores using dense ranking methodology (method='dense') within each dependent variable. Since the preprocessing step decomposed high-missing features (CI-Tests, Pinned-Dependencies) into separate components, we grouped all feature importance scores by their base security practice to provide more interpretable ranking results. The method involved summing the decomposed components for CI-Tests and Pinned-Dependencies, while non-decomposed features retained their individual importance scores. Using the ranked importance scores of 11 practice metrics, we selected the top-performing security practices that collectively accounted for approximately 80% of the cumulative feature importance across all three outcome metrics [96]. The approach provides interpretable insights into which security practices have the strongest overall predictive power for each outcome variable (Table 8.4).

High vs. Low Security Practice Scores: Comparing Similar Repositories

We conducted PSM analysis on the top-ranked security practices from Table 8.4. We focused on these top practices because the remaining ones demonstrated negligible importance values while the selected

top features collectively accounted for approximately 80% of the cumulative feature importance. However, individual security practices exhibit extreme binary distributions, with most repositories scoring either 0 (not adopted) or 10 (fully implemented), creating an extreme class imbalance that violates PSM overlap assumptions. Unlike aggregated Scorecard scores, which combine multiple practices to ensure nonzero values, individual practices remain at their adoption extremes. To enable valid causal inference, we applied balanced sampling to create sufficient propensity score overlap. Our PSM methodology employed a universal threshold of 5.0, classifying repositories with scores above this threshold as treated units and those at or below as control units. Following balanced sampling, after constructing balanced groups, we followed the technique discussed in Section 8.3.1, where we used nearest-neighbor matching with a caliper adjustment and computed ATE.

Unlike RQ1, where segmentation revealed how repository characteristics impact outcome, we could not perform a similar analysis in RQ2 due to data imbalance (Section 8.4.1). The limited sample size in the PSM analysis further constrained our ability to isolate project-specific influences on security outcome metrics.

8.4.2 RQ2: Result

Quantitative Confirmation: Table 8.4 summarizes our RQ2 findings, analyzing individual security practices' relationships with security outcomes. Security practices collectively explain a moderate portion of variance across outcomes: $R^2 = 0.39$ for Vul_Count , $R^2 = 0.50$ for MTTR, and $R^2 = 0.32$ for MTTU. The higher explanatory power for MTTR suggests stronger relationships between security practices and MTTR compared to Vul_Count or MTTU.

Important Security Practices: The analysis reveals consistent patterns across outcome metrics. Contributors emerge as the most influential factor across all three outcome metrics, followed by Code Review and License practices. CI-Tests and Pinned-Dependencies consistently rank among the top practices. The ranking pattern suggests that collaborative human factors (Contributors) and development workflow practices (Code Review, CI-Tests) have an influential, measurable impact on security outcomes. Branch Protection and Dependency Update Tool ranks among the top seven features required to reach the 80% cumulative importance threshold for Vul_Count. In contrast, MTTR and MTTU reach the threshold with five features, but including these two additional practices raises the cumulative importance to nearly 92% for MTTR and MTTU, indicating these practices also have a stronger impact on the outcomes.

Impact of Security Practice: The PSM analysis reveals complex and counterintuitive relationships between individual security practices and outcomes. Code Review consistently exhibits a positive ATE across all three security outcomes, indicating associations with more vulnerabilities, slower MTTR, and MTTU. This paradoxical finding aligns with CISA's Secure by Design guidance [47], a rise in reported vulnerabilities can reflect a healthy security posture due to code review and testing by the community. CI-Tests show a mixed pattern: projects with pull-request (PR) infrastructure tend to have faster MTTR but also higher vulnerability counts and slower MTTU, possibly due to added complexity from collaboration. However, projects with PR infrastructure that also achieve high CI-Test scores consistently perform better across all security metrics. The finding suggests that PR workflows alone are insufficient; security benefits arise when effective CI practices are fully implemented. Beyond Code Review and CI-Tests, other practices show more intuitive associations. Contributors, License, and Pinned Dependencies are associated with negative ATEs, suggesting improved outcomes if adopted by repositories. For Vul_Count, the Branch Protection and Dependency Update Tool

shows positive ATEs, suggesting these practices enhance vulnerability detection capabilities.

Our analysis shows that Contributors, Code Review, License, CI-Tests, and Pinned-Dependencies are the most influential security practices across all outcome metrics. While adoption of Code Review practice shows positive associations with more vulnerabilities and slower MTTR and MTTU, Contributors, License, and Pinned Dependencies demonstrate negative ATEs, indicating improved security outcomes. CI-Tests show mixed effects depending on CI-test adoption.

8.5 Discussion

The section discusses the research implications (section 8.5.1) and explores the threat to the validity (section 8.5.4) of our study.

8.5.1 Research Implication

Defining and validating security outcome metrics is challenging; our work represents a step toward advancing that goal. Our study provides actionable insights for practitioners, policymakers, and researchers by investigating measurable security outcome metrics to assess the effectiveness of security practices.

Holistic security practice adoption:. RQ1 demonstrates that aggregated scores consistently improve security outcomes, while RQ2 reveals context-dependent and sometimes counterintuitive effects of individual practices. Our findings suggest that comprehensive security practice adoptions are more effective than isolated measures, as coordinated practices generate synergistic effects that individual actions alone cannot achieve.

Context-Aware Adoption: The impact of security practice on security outcomes varies by project characteristics, requiring tailored strategies. While selecting dependencies, organizations should prioritize mature, actively maintained large projects with large contributors, fewer dependencies, and high downloaded packages, as our analysis showed that projects have fewer vulnerabilities and faster MTTR due to better security performance. Conversely, smaller or inactive packages with fewer downloads require enhanced monitoring and proactive security assessments to compensate for slower MTTU.

Strategic Practice Prioritization: Despite individual variability, Code Review, CI-Tests, License, Contributors, and Pinned Dependencies consistently demonstrate high feature importance across security outcomes. Our findings suggest that practitioners can adopt these empirically validated practices within their workflow, while recognizing that practices like Code Review, CI-test positive association reflect enhanced detection capabilities rather than security degradation. To ensure that security practice adoption is yielding tangible benefits, teams should regularly monitor outcome metrics. Continuous tracking of these indicators enables practitioners to assess the effectiveness of implemented practices and iteratively refine their security strategies based on empirical feedback.

Evidence-Based Policy Development: The quantified relationships between security practices and security outcomes provide empirical evidence for security frameworks and regulations. Policymakers can leverage these findings to develop guidelines that promote frameworks emphasizing coordinated, measurable practice adoption rather than checklist-based compliance. By aligning regulatory efforts with empirically supported metrics, policy can more effectively incentivize secure development behaviors across the software ecosystem.

8.5.2 Lesson Learned: Automation Challenges and Tooling Limitations

Figure 8.1 illustrates the adoption of individual security practices across the npm ecosystem, as visible npm packages have yet to adopt software security practices widely. However, ecosystem-wide automated measurement is also constrained by tool limitations and incomplete metric coverage, which can obscure the true extent of practice adoption. The section outlines the key challenges we encountered during our study, highlighting the limitations of automated security measurement and the complexities of interpreting Scorecard results at scale.

Metrics Detection Challenge: Although the dataset contains over 145k repositories, not all metrics are scored equally. For instance, Packaging and Signed-release show scores of 10 for only 3,692 and 3,426 repositories (Figure 8.1), respectively, due to the tool's reliance on GitHub workflows. The scorecard does not capture all packaging ecosystems, primarily because it lacks awareness of every possible packaging workflow or the capability to measure them comprehensively at this stage. Hence, the remaining repositories (97%) receive a score of -1, signaling inconclusive evidence rather than serving as a penalty. Many developers use GitHub mainly for source code [272], while actual package releases—signed or unsigned are handled in ecosystems. To tackle the gap would require Scorecard to integrate with external registries or for developers to mirror signed releases on GitHub, which can introduce overhead and may not scale well across teams. Similarly, SAST and Fuzzing metrics achieve a score of 0 across most repositories (Figure 8.1), not because the practices are not adopted at all, but because Scorecard only detects a limited set of predefined tools and configurations. For instance, SAST yields a positive score only if one of three specific tools is identified; otherwise, repositories receive a zero, even if another valid tool is used. The limitation leads to skewed distributions that hinder meaningful comparisons across repositories. These limitations highlight a broader challenge: the absence of industry consensus on tooling and the lack of interoperability between platforms. Although Scorecard offers guidelines, the lack of standardized tooling and cross-platform interoperability hampers effective detection. Without shared conventions, automation remains brittle and incomplete, limiting the usefulness of these metrics and increasing the cost and complexity of enhancing tool support.

Measurement Inconsistency: Many Scorecard metrics rely on binary heuristics, yielding either a perfect score of 10 or a 0, which results in skewed and extreme score distributions. For instance, the Binary-Artifacts check deducts one point per binary file, reaching zero after ten, without inspecting their content. While binary artifacts can introduce risk, such as the potential injection of malicious, non-reviewable code, the checks do not inspect the content of these binaries. The metric treats all binaries equally, regardless of purpose, leading to misleading risk signals, especially when binaries are used legitimately for testing or builds in many repositories. Without contextual analysis, the check offers limited insight and may misrepresent the security posture.

Similarly, the Code-Review metric scores repositories based on whether recent commits were reviewed by someone other than the committer. Automated reviews and bot approvals are excluded. This penalizes single-maintainer projects, which inherently cannot satisfy the "different merger and committer" rule, even if there is a possibility that changes are manually vetted. Only 2% of repositories have received a score of 10 (Figure 8.1), not because review is absent in the remaining repositories, but due to strict heuristics. Furthermore, these reviews are not assessed for quality or security issues; rather, the presence of a code review is equated with effectiveness, which may not hold in practice. Then, the license metric checks for a valid license from a predefined list of licenses, but this assessment is not inherently linked to software security. Determining license compatibility is more of a software engineering and legal practice than a security-relevant evaluation

since the tool does not assess whether the license is safe to use from a security perspective. Our findings reflect a broader issue in automated measurement: oversimplified rules that can overlook real-world software development practices. Hence, contextual awareness or flexibility matters, as rigid checks can misrepresent open-source projects' true security posture and limit the resulting metrics' actionable value.

Inconsistency Across Data Sources: Another challenge appeared during data acquisition when we compared Scorecard results obtained via BigQuery with those generated by running Scorecard directly on repositories. In the BigQuery dataset, many repositories received a score of 10 for the Vulnerabilities check, indicating no known vulnerabilities (Figure 8.1). However, running Scorecard locally on the same repositories simultaneously revealed that several repositories had vulnerabilities, contradicting the BigQuery results. The discrepancy highlights that the choice of data source and extraction method can impact the accuracy, completeness, and reliability of reported security scores.

Measuring Security Outcomes: We encountered several challenges in measuring security outcomes during the study. While we collected MTTU data for 145,817 packages, MTTR was available for only 22,412 (15%). MTTR is challenging to compute at scale because it depends on a specific sequence of observable events: the existence of a known vulnerability in an upstream dependency, the availability of an upstream fix, and the subsequent adoption of that fix by the dependent package. For the Vul_Count metric, only 49% of the packages had any publicly linked vulnerability data. In many cases, vulnerabilities may have been silently patched or never disclosed [61, 133, 265, 253, 153, 199]. In such cases, silently fixed vulnerability data are often inaccessible because the vulnerabilities were never publicly reported, which limits the reliability of Vul_Count as a comprehensive indicator of security posture. We tried to overcome these challenges by investigating temporal software metrics, MTTU, as a security outcome metric. Our finding shows that MTTU exhibits trends similar to Vul_Count and MTTR, making it a potential proxy metric when vulnerability and MTTR data are unavailable.

Additionally, the multivariate and time-dependent nature of the secure development practices makes it challenging to establish direct links to security outcomes. Understanding whether a practice leads to better security requires tracking when the practice was adopted, when the vulnerability was introduced, and when it was fixed. However, accessing such data is often challenging; for instance, for many metrics, Scorecard metrics only reflect activity from the past 90 days, making it difficult to assess historical practice implementation or its long-term impact. To more reliably assess the relationship between security practices and outcomes, future work should consider a controlled, long-term study that aligns timestamps of security practice implementation with vulnerability events. Such longitudinal tracking can help us draw more definitive conclusions about the effectiveness of secure development practices.

8.5.3 Future Work

Our findings highlight the need to better understand the temporal feedback loop between security practices and outcomes. As projects adopt more practices, they may initially report more vulnerabilities and delays in dependency updates and fixing vulnerabilities due to improved detection rather than increased risk. The complexity calls for longitudinal studies that track how individual and aggregated practices affect security outcome metrics. For researchers, future work should model these dynamics in the context of project maturity, contributor activity, and development cycles to clarify whether observed security trends reflect degradation or deliberate, ongoing improvement. A comprehensive understanding of security outcome metrics is an open

research challenge; therefore, researchers should explore new security outcome metrics beyond those used in this study.

8.5.4 Threat to Validity

Our regression and causal analyses provide evidence supporting associations and causality between aggregated Scorecard scores, security practices, and security outcomes within our dataset. However, several inherent biases in our dataset must be acknowledged. First, our analyses are based on snapshot data capturing repository Scorecard metric scores and characteristics at a single point in time, lacking temporal granularity. Without longitudinal or time-series analysis, uncertainty remains about whether changes in security practices precede or directly influence observed security outcomes, thereby limiting the generalizability of our findings beyond our dataset. Second, selection bias may be present, as repositories adopting security practices could differ in unmeasured factors such as organizational policies, economic incentives, developer motivations, or project complexity. Although PSM mitigates confounding through observed covariates, unobserved variables may still influence both practices and outcomes.

Apart from that, our dataset exhibits other imbalances, as repositories have security practice scores of either 0, or -1, reflecting the limited adoption of these practices across the npm ecosystem. The real-world imbalance, characterized by the few repositories that adopt security practices, may introduce bias into our models, potentially skewing predictions or causal estimations. However, this imbalance highlights a widespread lack of security practice adoption among npm repositories. Hence, the imbalance reflects the actual state of adoption rather than a methodological flaw, highlighting the need for targeted interventions to increase the adoption of security practices among npm repositories. At the same time, the metric collection also needs improvement, a team may be following a practice, but evidence of that practice may not be captured in the automated metrics.

8.6 Related Work

This section highlights prior related studies, existing security guidelines, frameworks, and security outcome metrics. Prior studies [274, 226] have utilized OpenSSF Scorecard metrics alongside vulnerability count data to understand the relationship between security practices and security outcome metrics. Zahan et al. [274] analyzed 2,422 PyPI and npm packages and found a positive association between Scorecard metrics and vulnerability counts at the package level, but with low explanatory power ($R^2 < 12\%$). They attributed this to limited dataset size, sparse vulnerability data at the package level, and missing control variables. Sonatype [226] used a random forest classifier to predict vulnerable repositories based on Scorecard metrics and project attributes. However, binary classification limits understanding of the strength and direction of relationships between practices and outcomes. In contrast, our study employs regression and causal analysis to quantify associations, uses a larger, more representative dataset of 145,817 npm packages, incorporates project-level outcome metrics, and controls for key project characteristics to improve validity and generalizability.

Security Practice Frameworks: Executive Order (EO) 14028 emphasized adopting security practices to improve software security. To align with the EO, government agencies and industry experts have introduced various software supply chain security frameworks to enhance software supply chain security. These include

the Software Component Verification Standard (SCVS) by OWASP [177], the Building Security In Maturity Model (BSIMM) [31], the CNCF Technical Advisory Group (TAG) recommendations [75], Supply Chain Levels for Software Artifacts (SLSA) [217], NIST Secure Software Development Framework (SSDF) [230], the Open Source Software (OSS) Secure Supply Chain (SSC) Framework by Microsoft [140], NIST's Cybersecurity Supply Chain Risk Management Practices [158], the CISA Secure Software Development Attestation Form [42] and OpenSSF Scorecard [206]. The Proactive Software Supply Chain Risk Management (P-SSCRM) [263] released in 2024, results from the study of real-world software supply chain risk management initiatives and the union of the 73 security practices in ten government and industry documents (standards and frameworks) mentioned above. In 2025, EO 14144 [237] was issued, emphasizing the need for NIST to evaluate common cyber practices and security outcomes across industry sectors.

Security Outcome Metrics: Various metrics can assess the effectiveness of security practices by tracking security performance over time, generally categorized as threat-oriented, asset/impact-oriented, and vulnerability-oriented [155, 156]. While threat- and asset/impact-oriented metrics offer valuable insights, they are difficult to quantify at scale due to reliance on real-time threat intelligence and complex impact assessments. In contrast, vulnerability-oriented metrics are more accessible and relevant to software security. Among them, severity-based metrics (e.g., CVSS) face challenges such as scoring inconsistencies and reliance on exploitability data [180]. Temporal vulnerability metrics, such as vulnerability count, time to detect, time to repair and time to remediate, are widely used to evaluate security [148, 202, 41, 147]. Morrison et al. [147] conducted a systematic mapping study on software security metrics to evaluate the security properties of software and found that the most cited and most used metric is the vulnerability count (vulnerability metric). Davidson [51] studied the time to close bug/vulnerability' (a form of mean time to repair) as a security metric and measured its effectiveness. Beres et al. [23] studied the window of exposure (vulnerability metric), which is the length of the interval of time needed by the security team to remediate a vulnerability once it has been disclosed. However, the scarcity of reported vulnerabilities, compounded by silently fixed issues, limits their reliability in assessing security outcomes [15, 61]. To address these limitations, prior work suggests alternative metrics, such as fault prediction models [212]. Additionally, in software reliability, MTTR and MTTU have long been used to measure system resilience and maintenance efficiency [88, 33]. While MTTR is often employed in security contexts [179, 182, 228], its reliance on reported vulnerabilities restricts its applicability. MTTU, by focusing on dependency updates, offers broader coverage but remains constrained by proprietary measurement methods [44].

8.7 Conclusion

In this study, we investigated how the aggregate Scorecard score and Scorecard security practices metrics impact security outcome metrics. Our regression model and causal analysis reveal that higher aggregated Scorecard scores are associated with fewer vulnerabilities, shorter MTTR, and shorter MTTU. However, while regression models and causal analysis indicate better outcomes, our segment analysis to understand the impact of project characteristics suggests that factors beyond security practice adoption, such as project maintainability, resource availability, and dependency complexity, influence security outcomes. Feature ranking highlights Code Review, Contributors, License, CI-Tests, and Pinned Dependencies as top security practices associated with security outcomes, though their associations with security outcomes vary in direction. Additionally, MTTU exhibits trends similar to vulnerability count and MTTR, making it a potential proxy

metric when vulnerability and MTTR data are unavailable. Future work should incorporate longitudinal data and additional security outcome metrics to further refine our understanding of how security practices impact software security over time.

CHAPTER

9

CONCLUSIONS

This chapter concludes the thesis, presenting a summary of the primary findings. The chapter also discusses the challenges encountered during the research of OSS supply chain security. Lastly, the chapter outlines prospective future directions in this research area.

9.1 Key Results

The dissertation presents an empirical investigation into SSC security, with a particular focus on dependency security, analyzed along two key dimensions: security practices and security outcomes. To evaluate the adoption and effectiveness of security practices, the work explores (1) identifying weak links in software dependencies to prevent account takeover and dependency hijacking; (2) an automated code-review workflow to detect malicious dependencies using large language models (LLMs); and (3) understanding the trends in the adoption of recommended security practices within the open-source ecosystem. On the outcomes side, the dissertation introduces and evaluates quantifiable security outcome metrics, including (1) the number of reported unfixed vulnerabilities in a package and its dependency graph; (2) the mean time to remediate (MTTR) vulnerable dependencies; and (3) the mean time to update (MTTU) dependencies. At the end, we also looked into quantifying the effectiveness of security practices through these outcome metrics, establishing empirically grounded links between what developers do and how secure their software becomes.

The identification of "weak link signals" was published in a conference paper [275], where we explored package metadata that makes packages or maintainers vulnerable to account takeover and dependency hijacking. Although we conducted the study on the npm ecosystem, the data-driven attacker-perspective analysis provided technology-agnostic, data-driven metrics that were later adopted by different ecosystems. The practical relevance of these findings has since influenced both platform-level policy and tool development.

Major ecosystems, including npm and GitHub, enforced two-factor authentication on developers' accounts to prevent account takeover attacks. Additionally, tools such as Socket, packj, GuardDog, and SDC-Check integrated the proposed metrics into their automated security checks, demonstrating the direct impact of this research on ecosystem-wide SSC defenses.

The second major contribution of this thesis was an automated code-review workflow to detect malicious dependencies using large language models (LLMs), which led to the publication of two conference papers [271, 270]. By constructing a malicious-specific benchmark and introducing novel prompt engineering strategies, the study demonstrated that LLMs can serve as effective agents for assisting in automated code review, achieving low false-positive rates. The MalwareBench dataset [271] from this study has been adopted by leading industry organizations, including Microsoft, Datadog, Chainguard, Fortinet, and Socket, as well as by over 50 academic research efforts, highlighting its broad applicability across both industry and academia. The automated code-review workflow [270], initially developed for the npm ecosystem in this study, is now being adapted and implemented across multiple ecosystems by Socket.

The third and most comprehensive study explored how the adoption of security practices impacts security outcomes. The research began with two studies examining ecosystem-wide security practice adoption trends and their impact on package vulnerability counts as security outcomes, published in a magazine [272], and a conference paper [274]. Building on these, our final study investigates 17 security practices and their relationship to three security outcome metrics, aligning with the recent shift in policy from checklist-based to outcome-based evaluation (e.g., U.S. EO 14144 [237]). While prior work [274] revealed limitations in using reported vulnerability counts due to insufficient data, our final study addresses this gap by utilizing comprehensive vulnerability data from both package code and complete dependency trees. We also introduce well-known metrics like MTTR and MTTU as novel security outcome indicators and demonstrate their effectiveness as proxies when traditional vulnerability metrics are unavailable. The insights from this study will enable practitioners to assess whether development practices improve security over time and provide policymakers with empirically grounded metrics for regulatory frameworks.

In conclusion, this thesis makes novel contributions to SSC security by conducting comprehensive investigations into both security practices and outcome metrics. The research advances the state of the art by introducing novel metrics like expired domain, automated techniques for detecting malicious dependencies using LLM, and empirically evaluating the relationship between security practice adoption and measurable security outcomes. Our contributions demonstrate that higher aggregated security practice adoption leads to better security outcomes, providing evidence-based foundations for both practitioners seeking to improve their security posture and policymakers developing outcome-focused regulatory frameworks. However, the challenge of establishing reliable and standardized security metrics at scale remains complex, requiring continued empirical research and cross-sector collaboration to reach industry consensus on what defines effective security practices and meaningful outcome measures. As the SSC landscape continues to evolve, it is important to advance our empirical understanding of the measurement of security practice and security outcome metrics and their effectiveness. The work presented in this thesis represents a step forward in this ongoing endeavor, and the methodologies developed and insights gained provide directions for future research in this domain of SSC security.

9.2 Reflection of OSS Supply Chain Security Research

This section outlines the challenges encountered during our study, highlighting the challenges of measuring security metrics in SSC research. The complexity of SSC security metrics measurement stems from the multi-faceted nature of modern software ecosystems, involving diverse platforms, toolchains, and development practices across numerous stakeholders with varying roles and privileges. This socio-technical complexity transcends purely technological boundaries, encompassing management practices, team dynamics, and user interactions that resist standardized measurement approaches. The inherent diversity of development workflows, dependency management strategies, and security implementations across different platforms makes it difficult for research and automated measurement tools, which often rely on narrow heuristics that fail to capture the full spectrum of legitimate security practices.

One of the primary challenges in malicious dependency detection research is the limited availability of labeled malicious samples. While recent works have constructed malicious datasets [165, 60, 97, 124], these studies typically lack neutral package data, limiting insights into ecosystem-wide good and bad code characteristics. Although we addressed this gap by constructing the MalwareBench dataset [271] with both malicious and neutral packages, all existing research datasets represent only a small fraction of real-world malicious samples due to systematic standard removal practices. Package repositories routinely remove malicious packages and replace original metadata with dummy entries, leaving researchers with access to only limited samples from deprecated mirrors, internet archives, community reports, and public research repositories. However, the 2024 SSC Report reveals that over 700k malicious packages have been proactively identified since 2019 [229], and these samples remain inaccessible to researchers. While this protective approach is beneficial since it effectively prevents downstream users from inadvertent exposure to malicious code, it simultaneously limits research opportunities that could enhance community-wide security. Controlled access to these malicious examples could provide valuable insights, including technical implementation details of malicious behaviors, identification of dominant attack patterns, analysis of temporal evolution in attack sophistication, and documentation of emerging new attack techniques. Such research contributions would not only enable users to recognize known malicious patterns but also accelerate the development of more sophisticated detection methodologies. Additionally, AI-based approaches for detecting malicious packages are gaining popularity due to their promising initial results. However, these methods face a fundamental limitation: models are often trained and evaluated on known, relatively simple malicious samples, as also mentioned in our study [270]. In the absence of diverse examples, including obfuscated, mutated, or more sophisticated variants, existing malicious techniques will struggle to generalize effectively to real-world threats.

Another core challenge in SSC research lies in the inherent complexity of measuring security metrics at scale across diverse platforms. Automated tools like Scorecard ace fundamental constraints in detection accuracy and scope, often relying on rigid heuristics or platform-specific workflows that systematically underrepresent legitimate security practices falling outside predefined parameters. Our analysis in Chapters 6–8 discusses limitations in tools like OpenSSF Scorecard, particularly regarding coverage, consistency, and interoperability. For example, Scorecard often assigns zero scores to practices such as static analysis or fuzzing, not because these practices were never adopted, but because the tool only recognizes a narrow set of predefined tools, failing to detect custom implementations or other tools. We also identify a lack of interoperability between GitHub and package registries: package metadata (e.g., dependency lists or package

URLs) may diverge from GitHub records, especially when one repository hosts multiple packages from different registries. The disconnect hinders accurate attribution of development practices and contributor activity for individual packages. Furthermore, Scorecard's reliance on GitHub-centric indicators introduces bias, disadvantaging projects that use alternative platforms. Software is not merely a technical artifact, but a product of human decisions, team dynamics, and evolving context [110]. Practices such as code review, CI/CD integration, and dependency management are deeply embedded in collaborative and socio-technical workflows that resist simple checklist-based evaluation. Thus, metrics designed without considering these dimensions risk producing incomplete or misleading evaluations.

From our study of MTTU and MTTR metrics in Chapter 8 and also from [189], we have learned the complexity of dependency resolution that further compounds these measurement challenges. The non-deterministic nature of dependency graphs, where a single package can yield millions of valid configurations, makes establishing consistent dependency views difficult. Dynamic ecosystem states, including evolving package versions and build environments, may lead each build to produce different dependency footprints, undermining reproducibility and complicating security metrics measurement. Cross-platform inconsistencies in naming conventions, versioning schemes, and resolution algorithms fragment the measurement landscape and hinder comparative analysis. To ensure reproducible research, researchers need to have access to standardized dependency snapshots at specific time points and use deterministic resolution algorithms to eliminate temporal and environmental variations.

These challenges open research opportunities to broader issues in automated security assessment: the need for standardized frameworks, improved tooling, and research to enable reproducible dependency tracking and standardized measurement approaches accounting for context-dependent security implementations. Future research should prioritize developing unified metadata standards and tools capable of reconstructing precise historical dependency graphs while refining existing detection capabilities to capture broader practice spectrums and integrate predictive models. To make meaningful progress, the field requires both sustained empirical research and greater industry coordination to establish shared standards that support security metric evaluation across diverse software ecosystems and characterize complex systems more holistically.

9.3 Future Work

We now discuss possible future research prospects of this thesis in the field of OSS supply chain security.

As the 2024 Sonatype research reported, over 700,000 malicious packages have been identified since 2019. However, the impact of such a large-scale attack frequency is still undetermined. Using the dataset from Chapter 5, we found that while 99% of malicious packages had limited reach (less than 10k downloads), one package alone amassed 11.7 billion downloads and over 7,000 dependents, revealing the outsized impact of a few widely adopted packages. Future work should develop frameworks for quantifying attack impact, incorporating dependency depth, affected users, and downstream propagation patterns, while examining whether high-impact malicious packages exhibit different characteristics than low-impact ones. Longitudinal studies tracking attack evolution could reveal how attackers adapt distribution strategies in response to detection mechanisms, informing more effective early detection and mitigation approaches.

Building on the empirical bases found in this thesis, future research should expand the security practice-outcome validation framework to encompass emerging different security frameworks and enterprise-grade projects under test. In Chapter 8, we found evidence that the characteristics of repositories influence the

effectiveness of security practices. A promising direction might involve investigating how security practice adoption differs between enterprise-grade repositories and small open-source packages, integrating additional frameworks such as SLSA, Macaron, Sigstore, in-toto, and enhanced OpenSSF Scorecard implementations. This research would collect comprehensive metrics, including SLSA compliance levels, SBOM generation practices, and cryptographic signing adoption, while building upon vulnerability counting, MTTU, and MTTR metrics as security outcomes. Furthermore, a comprehensive understanding of security outcome metrics remains an open research challenge. Metrics such as severity-based vulnerabilities, threat-centric indicators, and asset- or impact-focused measures warrant deeper investigation as meaningful security outcome metrics. Our findings also highlight the need to better understand the temporal feedback loop between security practices and outcomes, as projects adopting new practices (e.g., implementation of SAST/-Fuzzing) may initially report more vulnerabilities due to improved detection rather than increased risk. Such complexity calls for longitudinal studies tracking how practices affect security outcomes in the context of project maturity and development cycles, clarifying whether observed trends reflect degradation or deliberate improvement. While evaluating different frameworks and metrics is a valuable research direction, future work should also address practitioner adoption barriers through mixed-methods approaches, combining quantitative analysis of these frameworks with qualitative studies that explore usability challenges and real-world implementation constraints. Additionally, future research should investigate whether improvements in internal security outcomes correlate with incident frequency and severity at the user end, specifically, whether faster response times (MTTR/MTTU) or lower vulnerability counts correlate with fewer high-impact incidents. Establishing such correlations would represent another step toward measuring security outcomes.

In the context of SSC security, Software Bills of Materials (SBOMs) are gaining increased attention, particularly due to regulatory momentum such as the U.S. Executive Order 14028 [238]. SBOMs provide transparency into a software component's composition. However, current SBOM generation tools often struggle with accuracy [273], which undermines the reliability of dependency-based security metrics. Future research should focus on improving the completeness and precision of SBOMs to enable more accurate tracking across entire dependency trees. Additionally, extending SBOM formats to include security-relevant metadata such as install scripts, expired domains,security-sensitive API calls, and others can provide richer context for automated detection and help resolve ecosystem-specific inconsistencies. Incorporating standardized documentation of security practices into SBOMs would further support context-aware, cross-platform measurement of security adoption, enhancing both the evaluative power and practical utility of SBOMs in SSC research and policy.

REFERENCES

[1] OpenAI. https://openai.com.

[2] Socket, Inc. https://socket.dev/.

[3] Cybersecurity supply chain risk management practices for systems and organizations. https://csrc.nist.gov/pubs/sp/800/161/r1/final, 2023.

[4] Diffstaticanalyzer - differential static analysis tool for npm packages to detect malicious updates. https://github.com/lmu-plai/diff-CodeQL, 2023.

[5] Good security practice definitions. https://www.lawinsider.com/dictionary/good-security-practice, 2023.

[6] Openssf. https://openssf.org/, 2023.

[7] AI Scanner, 2024.

[8] Deps.dev bigquery public dataset. https://docs.deps.dev/bigquery/v1/, 2024. [Online; Accessed: March 2024].

[9] Scorecard bigquery public dataset. https://console.cloud.google.com/bigquery?inv=1&invt=AbrVXg&project=trying-bq-for-scorecard-data&ws=!1m0, 2024. [Online; Accessed: March 2024].

[10] The Elastic AI Assistant advantage. https://www.elastic.co/explore/security-without-limits/elastic-ai-assistant-analyst-report?, 2024.

[11] What is a prompt injection attack? . https://www.ibm.com/topics/prompt-injection, 2024.

[12] Sharma A. Protestware on the rise: Why developers are sabotaging their own code, July 2022.

[13] Anne Adams and Martina Angela Sasse. Users are not the enemy. *Communications of the ACM*, 42(12):40–46, 1999.

[14] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[15] Omar Hussain Alhazmi, Yashwant K Malaiya, and Indrajit Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *computers & security*, 26(3):219–228, 2007.

[16] Cloud Security Alliance. Global security database (gsd). *https://github.com/cloudsecurityalliance/gsd-database*, 2021.

[17] Allianz. Allianz risk barometer. https://commercial.allianz.com/content/dam/onemarketing/commercial/commercial/reports/Allianz-Risk-Barometer-2024.pdf, 2024.

[18] Ionut Arghire. Supply chain attack technique spoofs github commit metadata. *https://www.securityweek.com/supply-chain-attack-technique-spoofs-github-commit-metadata*, 2022.

[19] Peter C Austin. Balance diagnostics for comparing the distribution of baseline covariates between treatment groups in propensity-score matched samples. *Statistics in medicine*, 28(25):3083–3107, 2009.

[20] Maksim Balabash. Sdc-check. https://github.com/mbalabash/sdc-check, 2022.

[21] Bank of England. Systemic Risk Survey Results – 2025 H1. https://www.bankofengland.co.uk/systemic-risk-survey/2025/2025-h1, April 2025. Survey conducted between 27 January and 24 February 2025; Accessed: 2025-06-18.

[22] Victor R Basili et al. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[23] Yolanta Beres, Marco Casassa Mont, Jonathan Griffin, and Simon Shiu. Using security metrics coupled with predictive modeling and simulation to assess security processes. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 564–573. IEEE, 2009.

[24] Bird, Christian, et al. Does distributed development affect software quality? an empirical case study of windows vista. In *2009 IEEE 31st International Conference on Software Engineering*, pages 518–528. IEEE, 2009.

[25] Bird, Christian, et al. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14, 2011.

[26] Lxyeternal Blue. Pypi malregistry. https://github.com/lxyeternal/pypi_malregistry, 2023.

[27] Myles Borins. Top-100 npm package maintainers now require 2fa. https://github.blog/2022-02-01-top-100-npm-package-maintainers-require-2fa-additional-security/, 2022.

[28] Martin Briesch, Dominik Sobania, and Franz Rothlauf. Large language models suffer from their own output: An analysis of the self-consuming training loop. *arXiv preprint arXiv:2311.16822*, 2023.

[29] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[30] BSA. The bsa framework for secure software: A new approach to securing the software lifecycle, version 1.1. *https://www.bsa.org/files/reports/bsa_framework_secure_software_update_2020.pdf*, 2020.

[31] BSIMM. Bsimm12 2021 foundations report. *https://www.bsimm.com/content/dam/bsimm/reports/bsimm12-foundations.pdf*, 2021.

[32] Alexis Butler, Dan O'Keeffe, and Santanu Kumar Dash. Links between package popularity, criticality, and security in software ecosystems. In *IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*, pages 89–97, 2024.

[33] Miguel Calvo and Marta Beltrán. Applying the goal, question, metric method to derive tailored dynamic cyber risk metrics. *Information & Computer Security*, 2023.

[34] Martin Čarnogurskỳ. Attacks on package managers. *Bachelor Thesis, Masaryk University, Brünn, Tschechien*, 2019.

[35] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[36] Yi Cheng, Julia Deng, Jason Li, Scott A DeLoach, Anoop Singhal, and Xinming Ou. Metrics of security. In *Cyber defense and situational awareness*, pages 263–295. Springer, 2014.

[37] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[38] Catalin Cimpanu. Backdoored python library caught stealing ssh credentials. https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/, 2018.

[39] CISA. Defending against software supply chain attacks. *https://www.cisa.gov/sites/default/files/publications/defending_a*

[40] CISA. CISA Cybersecurity Strategic Plan FY2023-2025. https://www.cisa.gov/sites/default/files/2023-01/StrategicPlan_20220912-V2_508c.pdf, 2022.

[41] CISA. CISA Cybersecurity Strategic Plan FY2024-2026. https://www.cisa.gov/sites/default/files/2023-08/FY2024-2026_Cybersecurity_Strategic_Plan.pdf, 2023.

[42] CISA. Secure Software Attestation Form, 2024. Accessed: February 19, 2025.

[43] Benjamin E. Coe. Core contributor to the conventional-changelog ecosystem had their npm credentials compromised. https://github.com/conventional-changelog/conventional-changelog/issues/282#issuecomment-365367804/, 2018.

[44] Filipe Roseiro Cogo, Gustavo A Oliva, and Ahmed E Hassan. An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering*, 47(11):2457–2470, 2019.

[45] Lucian Constantin. Npm attackers sneak a backdoor into node.js deployments through dependencies. https://thenewstack.io/npm-attackers-sneak-a-backdoor-into-node-js-deployments-through-dependencies/, 2018.

[46] Lucian Constantin. Solarwinds attack explained: And why it was so hard to detect. https://www.csoonline.com/article/3601508/solarwinds-supply-chain-attack-explained-why-organizations-were-not-prepared.html, 2020.

[47] Cybersecurity and Infrastructure Security Agency (CISA). Secure by design: A strategic approach for software manufacturers, October 2023. Accessed: March 14, 2025.

[48] Cybersecurity and Infrastructure Security Agency. Joint Statement by the Federal Bureau of Investigation (FBI), the Cybersecurity and Infrastructure Security Agency (CISA), the Office of the Director of National Intelligence (ODNI), and the National Security Agency (NSA). https://www.cisa.gov/news-events/news/joint-statement-federal-bureau-investigation-fbi-cybersecurity-and-infrastructure-securi 2021.

[49] Brian d foy. The hijacking of perl.com. https://www.perl.com/article/the-hijacking-of-perl-com/, 2021.

[50] DataDog. Guarddog. https://github.com/datadog/guarddog, 2022.

[51] Mary Ann Davidson. The good, the bad, and the ugly: stepping on the security scale. In *2009 Annual Computer Security Applications Conference*, pages 187–195. IEEE, 2009.

[52] L. F. DeKoven, A. Randall, A. Mirian, G. Akiwate, A. Blume, L. K. Saul, A. Schulman, G. M. Voelker, and S. Savage. Measuring security practices and how they impact security. In *Proc. Internet Measurement Conf.*, pages 36–49, 2019.

[53] Louis F DeKoven, Audrey Randall, Ariana Mirian, Gautam Akiwate, Ansel Blume, Lawrence K Saul, Aaron Schulman, Geoffrey M Voelker, and Stefan Savage. Measuring security practices. *Communications of the ACM*, 65(9):93–102, 2022.

[54] DevOps Research and Assessment, Inc. Dora metrics: Accelerate devops research assessment. https://dora.dev/, 2025. Accessed July 2025.

[55] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases. In *Proceedings 2021 Network and Distributed System Security Symposium, Virtual*, 2021.

[56] Cezarina Dinu. Solarwinds attack cost impacted companies an average of $12 million. `https://heimdalsecurity.com/blog/solarwinds-attack-cost-impacted-companies-an-average-of-12-million/`, 2021.

[57] Mike Dixon. Overcoming security metrics challenges to measure what matters. *Security Magazine*, 2021. Accessed: 2025-06-23.

[58] Natasha Dorfman. Security holding package. `https://github.com/npm/security-holder`, 2021.

[59] Noam Dotam. Vulnerable github actions workflows. `https://www.legitsecurity.com/blog/github-privilege-escalation-vulnerability`, 2022.

[60] Ruian Duan et al. Towards measuring supply chain attacks on package managers for interpreted languages. *arXiv preprint arXiv:2002.01139*, 2020.

[61] Trevor Dunlap, Seaver Thorn, William Enck, and Bradley Reaves. Finding fixed vulnerabilities with off-the-shelf static analysis. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroSP)*, pages 489–505, 2023.

[62] Sarah Elder et al. Do i really need all this work to find vulnerabilities? an empirical case study comparing vulnerability detection techniques on a java application. *arXiv preprint arXiv:2208.01595*, 2022.

[63] Sarah Elizabeth Elder. Applying software security assessments in realistic contexts. 2024.

[64] Chris Eng. Quantifying the probability of flaws in open source. `https://www.veracode.com/blog/quantifying-probability-flaws-open-source`, July 2024. Veracode Blog.

[65] Jerrod Engelberg. Bash uploader security update.

[66] ENISA. Enisa threat landscape 2022, 2022.

[67] ESF. Securing the software supply chain: Recommended practices guide for developers. *https://www.cisa.gov/ESF_SECURING_THE_SOFTWARE_SUPPLY_CHAIN_DEVELOPERS.PDF*, 2022.

[68] Eslint. Postmortem for malicious packages published on july 12th, 2018. `https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes`, 2018.

[69] Shumailov et al. The curse of recursion: Training on generated data makes models forget. *https://arxiv.org/abs/2305.17493*, 2023.

[70] Zahan et al. Malwarebench. `https://github.com/MalwareBench`, 2023.

[71] Douglas Everson, Long Cheng, and Zhenkai Zhang. Log4shell: Redefining the web attack surface. In *Proc. Workshop Meas., Attacks, Defenses Web (MADWeb)*, pages 1–8, 2022.

[72] Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Containing malicious package updates in npm with a lightweight permission system. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1334–1346. IEEE, 2021.

[73] FireEye. Highly Evasive Attacker Leverages SolarWinds Supply Chain to Compromise Multiple Global Victims With SUNBURST Backdoor. `https://cloud.google.com/blog/topics/threat-intelligence/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor`, 2020.

[74] Nicole Forsgren, Jez Humble, and Gene Kim. 2019 accelerate state of devops report. Technical report, DevOps Research and Assessment (DORA), 2019. Accessed July 2025.

[75] Cloud Native Computing Foundation. Software supply chain best practices. *https://github.com/cncf/tag-security/tree/main/supply-chain-security/supply-chain-security-paper*, 2021.

[76] Andres Freund. backdoor in upstream xz/liblzma leading to ssh server compromise. `https://www.openwall.com/lists/oss-security/2024/03/29/4`, March 29, 2024.

[77] Fabian Froh, Matías Gobbi, and Johannes Kinder. Differential static analysis for detecting malicious updates to open source packages. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 41–49, 2023.

[78] Jim Frost. How to interpret regression models that have significant variables but a low r-squared. *https://statisticsbyjim.com/regression/low-r-squared-regression/*, 2022.

[79] Jim Frost. Control variables: Definition, uses & examples. `https://statisticsbyjim.com/basics/control-variables/`, 2024. Accessed: February 21, 2025.

[80] Alex Gantman. NDSS 2022 Keynote - Measuring Security Outcomes. YouTube, NDSS Symposium, April 2022. Accessed: 2025-06-20.

[81] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Detecting suspicious package updates. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 13–16. IEEE, 2019.

[82] Harry Garrood. Malicious code in the purescript npm installer. `https://harry.garrood.me/blog/malicious-code-in-purescript-npm-installer/`, 2019.

[83] Gartner. Security metrics. `https://www.gartner.com/en/information-technology/glossary/security-metrics`, 2023.

[84] Jack M. Germain. Lessons learned from the solarwinds supply chain hack. `https://www.technewsworld.com/story/lessons-learned-from-the-solarwinds-supply-chain-hack-87029.html`, 2021.

[85] Github. Github advisory database. *https://github.com/advisories*, 2021.

[86] GitHub. The github branches. `api.github.com/repos/\{OWNER\}/\{REPO\}/branches/\{branch-name\}`, 2022.

[87] GitHub. Github rest api v3 - repository endpoint. `https://docs.github.com/en/rest/repos/repos#get-a-repository`, 2024. [Online; accessed 6-Mar-2025].

[88] Swapna S Gokhale and Kishor S Trivedi. A time/structure based software reliability model. *Annals of Software Engineering*, 8(1):85–121, 1999.

[89] GoLang. The go vulnerability database. *https://github.com/golang/vulndb*, 2021.

[90] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schäfer. Anomalicious: Automated detection of anomalous and potentially malicious commits on github. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 258–267. IEEE, 2021.

[91] Sarah Gooding. How socket combats insidious typosquatting supply chain attacks, 2023.

[92] Google. Oss-fuzz: Continuous fuzzing for open source software. *https://github.com/google/oss-fuzz*, 2016.

[93] Google. Open source vulnerability database. https://osv.dev/, 2021.

[94] Google Cloud and DORA. 2022 accelerate state of devops report. Technical report, DevOps Research and Assessment (DORA), 2022. Accessed July 2025.

[95] Karen Grace-Martin. Can a regression model with a small r-squared be useful? *https://www.theanalysisfactor.com/small-r-squared/*, 2022.

[96] UCLA Statistical Consulting Group. Exploratory factor analysis (efa) using spss, 2023. Accessed: 2025-06-15.

[97] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. An empirical study of malicious code in pypi ecosystem. *arXiv preprint arXiv:2309.11021*, 2023.

[98] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[99] Sivana Hamer, Jacob Bowen, Md Nazmul Haque, Robert Hines, Chris Madden, and Laurie Williams. Closing the chain: How to reduce your risk of being solarwinds, log4j, or xz utils. *arXiv preprint arXiv:2503.12192*, 2025.

[100] Sil Hamilton. Detecting mode collapse in language models via narration. *arXiv preprint arXiv:2402.04477*, 2024.

[101] Red Hat. Urgent security alert for fedora linux 40 and fedora rawhide users. https://www.redhat.com/en/blog/urgent-security-alert-fedora-40-and-rawhide-users, March 29, 2024.

[102] Georges Hatem, Joe Zeidan, Mathijs Goossens, and Carla Moreira. Normality testing methods and the importance of skewness and kurtosis in statistical analysis. *BAU Journal-Science and Technology*, 3(2):7, 2022.

[103] Jerry A Hausman, Bronwyn H Hall, and Zvi Griliches. Econometric models for count data with an application to the patents-r&d relationship, 1984.

[104] N Henze et al. A class of invariant consistent tests for multivariate normality. *Communications in statistics-Theory and Methods*, 19(10):3595–3617, 1990.

[105] Joseph M Hilbe. *Negative binomial regression*. Cambridge University Press, 2011.

[106] Ckobopoaa Hnknta. Gathering weak npm credentials. https://github.com/ChALkeR/notes/blob/master/Gathering-weak-npm-credentials.md/, 2018.

[107] Anton Hoffman. Solarwinds orion security breach: A shift in the software supply chain paradigm. https://snyk.io/blog/solarwinds-orion-security-breach-a-shift-in-the-software-supply-chain-paradigm/, 2021.

[108] The White House. Federal Cybersecurity Research and Development Strategic Plan 2023. https://www.whitehouse.gov/wp-content/uploads/2024/01/Federal-Cybersecurity-RD-Strategic-Plan-2023.pdf, 2023.

[109] The White House. NATIONAL CYBERSECURITY STRATEGY. https://www.whitehouse.gov/wp-content/uploads/2023/03/National-Cybersecurity-Strategy-2023.pdf, 2023.

[110] The White House. BACK TO THE BUILDING BLOCKS: A PATH TOWARD SECURE AND MEASURABLE SOFTWARE. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf, 2024.

[111] Thomas Hunter II. Compromised npm package: event-stream. https://medium.com/intrinsic-blog/compromised-npm-package-event-stream-d47d08605502, 2018.

[112] Nasif Imtiaz and Laurie Williams. Phantom artifacts & code review coverage in dependency updates. *arXiv preprint arXiv:2206.09422*, 2022.

[113] in toto. in-toto. *https://in-toto.io/in-toto/*, 2021.

[114] Abbas Javan Jafari, Diego Elias Costa, Emad Shihab, and Rabe Abdalkareem. Dependency update strategies and package characteristics. *ACM Transactions on Software Engineering and Methodology*, 32(6):1–29, 2023.

[115] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.

[116] Joint Research Centre and ENISA. Cyber Resilience Act Requirements Standards Mapping. Technical report, European Union Agency for Cybersecurity (ENISA), 2024.

[117] Maya Kaczorowski. Secure at every step: What is software supply chain security and why does it matter? https://github.blog/2020-09-02-secure-your-software-supply-chain-and-protect-against-supply-chain-threats-gi 2020.

[118] Sushawapak Kancharoendee, Thanat Phichitphanphong, Chanikarn Jongyingyos, Brittany Reid, Raula Gaikovina Kula, Morakot Choetkiertikul, Chaiyong Ragkhitwetsagul, and Thanwadee Sunetnanta. On categorizing open source software security vulnerability reporting mechanisms on github. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 751–756. IEEE, 2025.

[119] Siddik Keskin. Comparison of several univariate normality tests regarding type i error rate and power of the test in simulation based small samples. *Journal of Applied Science Research*, 2(5):296–300, 2006.

[120] Swati Khandelwal. Ccleaner attack timeline—here's how hackers infected 2.3 million pcs. https://thehackernews.com/2018/04/ccleaner-malware-attack.html, 2018.

[121] Guy King. Best security practices: An overview. In *Proceedings of the 23rd National Information Systems Security Conference, Baltimore, Maryland, NIST*. Citeseer, 2000.

[122] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners, 2022. *https://arxiv. org/abs/2205.11916*.

[123] Aobo Kong, Shiwan Zhao, Hao Chen, Qicheng Li, Yong Qin, Ruiqi Sun, and Xin Zhou. Better zero-shot reasoning with role-play prompting. *arXiv preprint arXiv:2308.07702*, 2023.

[124] Datadog Security Labs. Iopen-source dataset of malicious software packages. https://github.com/datadog/malicious-software-packages-dataset, 2023.

[125] Datadog Security Labs. Iopen-source dataset of malicious software packages. https://github.com/datadog/malicious-software-packages-dataset, 2023.

[126] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Taxonomy of attacks on open-source software supply chains. *arXiv preprint arXiv:2204.04008*, 2022.

[127] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. Risk explorer for software supply chains: Understanding the attack surface of open-source based software development. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 35–36, 2022.

[128] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. Towards the detection of malicious java packages. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 63–72, 2022.

[129] Piergiorgio Ladisa, Serena Elisa Ponta, Nicola Ronzoni, Matias Martinez, and Olivier Barais. On the feasibility of cross-language detection of malicious packages in npm and pypi. *arXiv preprint arXiv:2310.09571*, 2023.

[130] Daniele Lain, Tarek Jost, Sinisa Matetic, Kari Kostiainen, and Srdjan Capkun. Content, nudges and incentives: A study on the effectiveness and perception of embedded phishing training. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4182–4196, 2024.

[131] Raosaheb V Latpate. Inverse adaptive stratified random sampling. *Statistical Methods and Applications in Forestry and Environmental Sciences*, pages 47–55, 2020.

[132] Feng Li, Yiling Lou, Xin Tan, Zhenpeng Chen, Jinhao Dong, Yang Li, Xuanzhi Wang, Dan Hao, and Lu Zhang. What can we learn from quality assurance badges in open-source software? *Science China Information Sciences*, 67(4):142103, 2024.

[133] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd annual conference on computer security applications*, pages 201–213, 2016.

[134] Libraries.io. Libraries.io - open source discovery and dependency insights. `https://libraries.io/`, 2024. [Online; accessed 6-Mar-2025].

[135] Yiheng Liu, Tianle Han, Siyuan Ma, Jiayue Zhang, Yuanyuan Yang, Jiaming Tian, Hao He, Antong Li, Mengshen He, Zhengliang Liu, et al. Summary of chatgpt-related research and perspective towards the future of large language models. *Meta-Radiology*, page 100017, 2023.

[136] Mikola Lysenko. Introducing socket ai – chatgpt-powered threat analysis. `https://socket.dev/blog/introducing-socket-ai-chatgpt-powered-threat-analysis`, 2023.

[137] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

[138] Mike Mcquade. "the untold story of notpetya, the most devastating cyberattack in history. `https://www.idagent.com/blog/2017-08-03-notpetya-threat-supply-chains-across-ukraine/`, 2018.

[139] Andrew Meneely and Laurie Williams. Secure open source collaboration: an empirical study of linus' law. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 453–462, 2009.

[140] Microsoft. Oss ssc framework. *https://github.com/microsoft/oss-ssc-framework*, 2022.

[141] Anders Aasrum Milje. Detecting malicious python packages in the python package index (pypi). Master's thesis, NTNU, 2022.

[142] Alyssa Miller and Sharone Zitzman. The state of open source security 2020. `https://snyk.io/open-source-security/`, 2020.

[143] Courtney Miller, Mahmoud Jahanshahi, Audris Mockus, Bogdan Vasilescu, and Christian Kästner. Understanding the response to open-source dependency abandonment in the npm ecosystem. In *Int'l Conf. Software Engineering (ICSE), IEEE/ACM*, 2025.

[144] MITRE Corporation. Common Vulnerabilities and Exposures, 2024. Accessed: 2025-04-08.

[145] Robert Morris and Ken Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, 1979.

[146] Patrick Morrison. A security practices evaluation framework. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 935–938. IEEE, 2015.

[147] Patrick Morrison, David Moye, Rahul Pandita, and Laurie Williams. Mapping the field of software life cycle security metrics. *Information and Software Technology*, 102:146–159, 2018.

[148] Patrick Morrison, David Moye, and Laurie Ann Williams. Mapping the field of software security metrics. Technical report, North Carolina State University. Dept. of Computer Science, 2014.

[149] Motahare Mounesan, Hossein Siadati, and Sima Jafarikhah. Exploring the threat of software supply chain attacks on containerized applications. In *2023 16th International Conference on Security of Information and Networks (SIN)*, pages 1–8. IEEE, 2023.

[150] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 521–530. IEEE, 2008.

[151] Engineering National Academies of Sciences, Medicine, et al. Cyber hard problems: Focused steps toward a resilient digital future. 2025.

[152] John Ashworth Nelder and Robert WM Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society Series A: Statistics in Society*, 135(3):370–384, 1972.

[153] Giang Nguyen-Truong, Hong Jin Kang, David Lo, Abhishek Sharma, Andrew E Santosa, Asankhaya Sharma, and Ming Yi Ang. Hermes: Using commit-issue linking to detect vulnerability-fixing commits. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 51–62. IEEE, 2022.

[154] Dempsey K. Pillitteri V. Y. Nieles, M. Nist special publication 800-12, revision 1. *National Institute of Standards & Technology*, 800:101, 2017.

[155] NIST. Performance Measurement Guide for Information Security. Technical Report NIST SP 800-55 Rev 1, National Institute of Standards and Technology, 2003.

[156] NIST. Guide for Conducting Risk Assessments. Technical Report NIST SP 800-30 Rev 1, National Institute of Standards and Technology, 2012.

[157] NIST. NVD - CVE-2021-44228. `https://nvd.nist.gov/vuln/detail/CVE-2021-44228`, 2021. Accessed: 2023-09-08.

[158] NIST. Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations. Technical Report NIST SP 800-161 Rev 1, National Institute of Standards and Technology, 2022.

[159] NIST. Software supply chain security guidance under executive order (eo) 14028 section 4e. *https://www.nist.gov/system/files/documents/2022/02/04/software-supply-chain-security-guidance-under-EO-14028-section-4e.pdf*, 2022.

[160] NIST. Security Controls - NIST Glossary, 2024. Accessed: February 19, 2025.

[161] npmjs. Deprecated package. `https://docs.npmjs.com/deprecating-and-undeprecating-packages-or-pa` 2021.

[162] npmjs. Specifics of npm's package.json handling. `https://docs.npmjs.com/cli/v7/configuring-npm/package-json/`, 2021.

[163] Marc Ohm, Felix Boes, Christian Bungartz, and Michael Meier. On the feasibility of supervised machine learning for the detection of malicious software packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–10, 2022.

[164] Marc Ohm, Lukas Kempf, Felix Boes, and Michael Meier. *Towards Detection of Malicious Software Packages Through Code Reuse by Malevolent Actors*. Gesellschaft für Informatik, Bonn, 2022.

[165] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pages 23–43. Springer, 2020.

[166] Marc Ohm and Charlene Stuke. Sok: Practical detection of software supply chain attacks. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, pages 1–11, 2023.

[167] Open Source Security Foundation (OpenSSF). Open source project security baseline (osps baseline). `https://baseline.openssf.org/`, February 2025. Current version (2025-02-25).

[168] OpenAI. Api rate limit advice. `https://help.openai.com/en/articles/6891753-api-rate-limit-advice`, 2024.

[169] OpenSSF. Open source security metrics. *https://metrics.openssf.org/*, 2021.

[170] OpenSSF. npm best practices guide. *https://github.com/ossf/package-manager-best-practices*, 2022.

[171] OpenSSF. Openssf scorecard (Website). `https://securityscorecards.dev`, 2023. [Online; Accessed: 30-Sept-2025].

[172] Ortelius Project. The ortelius openssf dashboard. `https://ortelius.io/blog/2024/12/21/the-ortelius-openssf-dashboard/`, December 2024. Accessed: 2025-06-12.

[173] Marco Ortu, Michele Marchesi, and Roberto Tonelli. Empirical analysis of affect of merged issues on github. In *2019 IEEE/ACM 4th International Workshop on Emotion Awareness in Software Engineering (SEmotion)*, pages 46–48. IEEE, 2019.

[174] OSI. Open source insight. `https://deps.dev/`, 2022.

[175] Ossillate-inc. packj. `https://packj.dev/`, 2022.

[176] OWASP. Software assurance maturity model v2. *https://drive.google.com/file/d/1ZWMk4dpS3zpXjE28wi4cf5Lq6TUjeA5x*, 2020.

[177] OWASP. Software component verification standard (scvs). *https://owasp-scvs.gitbook.io/scvs/*, 2020.

[178] Robert M O'brien. A caution regarding rules of thumb for variance inflation factors. *Quality & quantity*, 41:673–690, 2007.

[179] Harshil Parikh. Mttr: The most important security metric. `https://www.darkreading.com/cyberattacks-data-breaches/mttr-most-important-security-metric`, 2023.

[180] Marcus Pendleton, Richard Garcia-Lebron, Jin-Hee Cho, and Shouhuai Xu. A survey on systems security metrics. *ACM Computing Surveys (CSUR)*, 49(4):1–35, 2016.

[181] Brian Pfretzschner and Lotfi ben Othmane. Identification of dependency-based attacks on node. js. In *Proceedings of the 12th international conference on availability, reliability and security*, pages 1–6, 2017.

[182] Plextrac. Mttd and mttr in cybersecurity. `https://plextrac.com/blog/mttd-and-mttr-in-cybersecurity/`, 2023.

[183] G. Polasani and S. Rubin. Embedded malware in npm: Coa, rc, ua-parser. `https://fossa.com/blog/embedded-malware-npm-coa-rc-ua-parser/`, 2021.

[184] Badge Program. Cii best practices badge program. *https://bestpractices.coreinfrastructure.org/en*, 2021.

[185] Moumita Das Purba, Arpita Ghosh, Benjamin J Radford, and Bill Chu. Software vulnerability detection using large language models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 112–119. IEEE, 2023.

[186] PyPA. Python packaging advisory database. *https://github.com/pypa/advisory-database*, 2021.

[187] PyPI. Pypi api for package name. `https://pypi.org/simple/`, 2022.

[188] Chengwei Qin, Aston Zhang, Zhuosheng Zhang, Jiaao Chen, Michihiro Yasunaga, and Diyi Yang. Is chatgpt a general-purpose natural language processing task solver? *arXiv preprint arXiv:2302.06476*, 2023.

[189] Imranur Rahman, Nusrat Zahan, Stephen Magill, William Enck, and Laurie Williams. Characterizing dependency update practice of npm, pypi and cargo packages. *arXiv preprint arXiv:2403.17382*, 2024.

[190] Bruce Ratner. The correlation coefficient: Its values range between+ 1/- 1, or do they? *Journal of targeting, measurement and analysis for marketing*, 17(2):139–142, 2009.

[191] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 155–165, 2014.

[192] Elissa M Redmiles, Noel Warford, Amritha Jayanti, Aravind Koneru, Sean Kross, Miraida Morales, Rock Stevens, and Michelle L Mazurek. A comprehensive quality evaluation of security and privacy advice on the web. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 89–108, 2020.

[193] Robert W Reeder, Iulia Ion, and Sunny Consolvo. 152 simple steps to stay safe online: Security advice for non-tech-savvy users. *IEEE Security & Privacy*, 15(5):55–64, 2017.

[194] Martin Ridout, John Hinde, and Clarice GB Demétrio. A score test for testing a zero-inflated poisson regression model against zero-inflated negative binomial alternatives. *Biometrics*, 57(1):219–223, 2001.

[195] Russ Cox. Timeline of the xz open source attack. `https://research.swtch.com/xz-timeline`, 2024.

[196] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*, pages 757–762. IEEE, 2018.

[197] RustSec. Rustsec advisory database. *https://github.com/golang/vulndb*, 2018.

[198] Johan Rydin. Evaluating open source network automation: A framework for security and maintainability. *https://www.diva-portal.org/smash/get/diva2:1971633/FULLTEXT01*, 2025.

[199] Antonino Sabetta and Michele Bezzi. A practical approach to the automatic classification of security-relevant commits. In *2018 IEEE International conference on software maintenance and evolution (ICSME)*, pages 579–582. IEEE, 2018.

[200] Niraj Salvi, Amal Thundiyil, Seema Supe, and Nataasha Raul. Evaluating security and community health metrics of foss repositories. *https://www.ijfmr.com/papers/2024/2/16980.pdf*.

[201] Rami Sammak, Anna Lena Rotthaler, Harshini Sri Ramulu, Dominik Wermke, and Yasemin Acar. Developers' approaches to software supply chain security: An interview study. In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 56–66, 2023.

[202] SANS. A sans 2021 survey: Rethinking the sec in devsecops: Security as code. *https://www.sans.org/webcasts/2021-survey-rethinking-sec-devsecops-security-code-118235/*, 2021.

[203] Natalie M Scala, Allison C Reilly, Paul L Goethals, and Michel Cukier. Risk and the five hard problems of cybersecurity. *Risk Analysis*, 39(10):2119–2126, 2019.

[204] Simone Scalco, Ranindya Paramitha, Duc-Ly Vu, and Fabio Massacci. On the feasibility of detecting injections in malicious npm packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–8, 2022.

[205] Joseph L Schafer and John W Graham. Missing data: our view of the state of the art. *Psychological methods*, 7(2):147, 2002.

[206] Scorecard. Security scorecards for open source projects. *https://github.com/ossf/scorecard*, 2021.

[207] Stewart Scott. Counting the costs. 2025.

[208] Stewart Scott. Counting the costs: A cybersecurity metrics framework for policy. Technical report, Atlantic Council, Cyber Statecraft Initiative, Washington, D.C., May 2025.

[209] Seaborn. seaborn-qqplot. *https://seaborn-qqplot.readthedocs.io/en/latest/*, 2022.

[210] Adriana Sejfia and Max Schäfer. Practical automated detection of malicious npm packages. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1681–1692, 2022.

[211] Ax Sharma. Dependency-confusion. `https://blog.sonatype.com/malicious-dependency-confusion-copycats-exfiltrate-bash-history-and-etc-shadow-files`, 2021.

[212] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18:25–59, 2013.

[213] Adam Shostack, Matthew Smith, Sam Weber, and Mary Ellen Zurko. Empirical evaluation of secure development processes (dagstuhl seminar 19231). *Dagstuhl Reports*, 9(6):1–25, 2019.

[214] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, and Yarin Gal. Ai models collapse when trained on recursively generated data. *Nature*, 631(8022):755–759, 2024.

[215] Sarah Ali Siddiqui, Chandra Thapa, Rayne Holland, Wei Shao, and Seyit Camtepe. Elevating software trust: Unveiling and quantifying the risk landscape. *arXiv preprint arXiv:2408.02876*, 2024.

[216] sklearn. Selectkbest. *https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html*, 2022.

[217] SLSA. Supply chain levels for software artifacts. *https://slsa.dev/*, 2022.

[218] Snyk. Snyk vulnerability database. *https://snyk.io/vuln*, 2021.

[219] Snyk. 10 github security best practices. `https://snyk.io/blog/ten-git-hub-security-best-practices/`, 2022.

[220] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653*, 2023.

[221] Socket. Dependency confusion, 2023.

[222] SocRadar. Smoothoperator supply chain attack targeting 3cx voip desktop client, Mar 2023. Last accessed December 2, 2023.

[223] SolarWinds. SolarWinds Security Advisory. https://www.solarwinds.com/sa-overview/securityadvisory, 2021.

[224] Sonatype. 2020 state of the software supply chain report. https://www.sonatype.com/resources/white-paper-state-of-the-software-supply-chain-2020, 2020.

[225] Sonatype. 2021 state of the software supply chain report. https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021, 2021.

[226] Sonatype. 2022 state of the software supply chain report. *https://www.sonatype.com/state-of-the-software-supply-chain/introduction*, 2022.

[227] Sonatype. 700% average increase in open source supply chain attacks. https://www.sonatype.com/press-releases/sonatype-finds-700-average-increase-in-open-source-supply-chain-attacks, 2022.

[228] Sonatype. 2019 stateof the software supply chain. https://www.sonatype.com/hubfs/SSC/2019%20SSC/SON_SSSC-Report-2019_jun16-DRAFT.pdf, 2023.

[229] Sonatype. State of the software supply chain. https://www.sonatype.com/state-of-the-software-supply-chain/introduction, 2024. Accessed: February 24, 2025.

[230] Murugiah Souppaya, Karen Scarfone, and Donna Dodson. Secure software development framework (ssdf) version 1.1. *NIST Special Publication*, 800:218, 2022.

[231] Synopsys. What is software supply chain security? https://www.synopsys.com/glossary/what-is-software-supply-chain-security.html, 2023.

[232] Synopsys. 2024 Open Source Security and Risk Analysis Report. https://www.blackduck.com/resources/analyst-reports/open-source-security-risk-analysis.html, 2025.

[233] Liran Tal. Snyk uncovers malicious code activities in open source supply chain security on the npm registry. https://snyk.io/blog/npm-security-malicious-code-in-oss-npm-packages/, 2021.

[234] Liran Tal and Juan Picado. 10 npm security best practices. https://snyk.io/blog/ten-npm-security-best-practices/, 2019.

[235] Matthew Taylor, Ruturaj K Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. Spellbound: Defending against package typosquatting. *arXiv preprint arXiv:2003.03471*, 2020.

[236] The Integrity360 Team. The adverline breach and the emerging risk of using third-party vendors. https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/, 2019.

[237] The White House. Executive order on strengthening and promoting innovation in the nation's cybersecurity, January 16 2025. Accessed: 2025-02-18.

[238] DC The White House. Executive order on improving the nation's cybersecurity. https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity, 2021.

[239] Christopher Thompson and David Wagner. A large-scale study of modern code review and security in open source projects. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 83–92, 2017.

[240] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[241] Dirk Thorleuchter and Dirk Van den Poel. Weak signal identification with semantic web mining. *Expert Systems with Applications*, 40(12):4978–4985, 2013.

[242] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1393–1410, 2019.

[243] Bill Toulas. Npm ecosystem at risk from "manifest confusion" attacks, 2023.

[244] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proceedings of the 40th international conference on software engineering*, pages 511–522, 2018.

[245] Ruturaj K Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. Security issues in language-based sofware ecosystems. *arXiv preprint arXiv:1903.02613*, 2019.

[246] Steven J. Vaughan-Nichols. Solarwinds, the world's biggest security failure and open source's better answer. `https://thenewstack.io/solarwinds-the-worlds-biggest-security-failure-and-open-sources-better-answer/`, 2020.

[247] Elisa Velarde. Nexus intelligence insights: Sonatype-2020-0003 - npm malicious package 1337qq-js. `https://blog.sonatype.com/sonatype-2020-0003-npm-malicious-package-1337qq-js`, 2020.

[248] John Viega and Gary R McGraw. *Building secure software: How to avoid security problems the right way, portable documents*. Pearson Education, 2001.

[249] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. Lastpymile: identifying the discrepancy between sources and packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 780–792, 2021.

[250] Duc-Ly Vu, Zachary Newman, and John Speed Meyers. Bad snakes: Understanding and improving python package index malware scanning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 499–511. IEEE, 2023.

[251] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. Towards using source code repositories to identify software supply chain attacks. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 2093–2095, 2020.

[252] vulert. Malicious Code In Eslint-Plugin-Unicorn-Ts-2. `https://vulert.com/vuln-db/npm-eslint-plugin-unicorn-ts-2-124511`, 2024. Last accessed: Feb 28, 2024.

[253] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. Detecting" 0-day" vulnerability: An empirical study of secret security patch in oss. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 485–492. IEEE, 2019.

[254] Xinyuan Wang. On the feasibility of detecting software supply chain attacks. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*, pages 458–463. IEEE, 2021.

[255] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[256] Zhi Wei et al. Nonparametric pathway-based regression models for analysis of genomic data. *Biostatistics*, 8(2):265–284, 2007.

[257] Charles Weir, Sammy Migues, Mike Ware, and Laurie Williams. Infiltrating security into development: exploring the world's largest software security study. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1326–1336, 2021.

[258] Dominik Wermke et al. Committed to trust: A qualitative study on security & trust in open source software projects. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1572–1572. IEEE Computer Society, 2022.

[259] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[260] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In *Generative AI for Effective Software Development*, pages 71–108. Springer, 2024.

[261] Dumidu Wijayasekara, Milos Manic, Jason L Wright, and Miles McQueen. Mining bug databases for unidentified software vulnerabilities. In *2012 5th International conference on human system interactions*, pages 89–96. IEEE, 2012.

[262] Laurie Williams, Giacomo Benedetti, Sivana Hamer, Ranindya Paramitha, Imranur Rahman, Mahzabin Tamanna, Greg Tystahl, Nusrat Zahan, Patrick Morrison, Yasemin Acar, et al. Research directions in software supply chain security. *ACM Transactions on Software Engineering and Methodology*, 2024.

[263] Laurie Williams, Sammy Migues, Jamie Boote, and Ben Hutchison. Proactive Software Supply Chain Risk Management Framework (P-SSCRM) Version 1, 2024. Accessed: February 21, 2025.

[264] Zekun Wu, Seonglae Cho, Umar Mohammed, Cristian Munoz, Kleyton Costa, Xin Guan, Theo King, Ze Wang, Emre Kazim, and Adriano Koshiyama. Libvulnwatch: A deep assessment agent system and leaderboard for uncovering hidden vulnerabilities in open-source ai libraries. *arXiv preprint arXiv:2505.08842*, 2025.

[265] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. Spain: security patch analysis for binaries towards understanding the pain and pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 462–472. IEEE, 2017.

[266] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Bing Yin, and Xia Hu. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *arXiv preprint arXiv:2304.13712*, 2023.

[267] Qinyuan Ye, Maxamed Axmed, Reid Pryzant, and Fereshte Khani. Prompt engineering a prompt engineer. *arXiv preprint arXiv:2311.05661*, 2023.

[268] Jiaxin Yu, Peng Liang, Yujia Fu, Amjed Tahir, Mojtaba Shahin, Chong Wang, and Yangxiao Cai. Security code review by llms: A deep dive into responses. *arXiv preprint arXiv:2401.16310*, 2024.

[269] Nusrat Zahan. Impact of the research study- "what are weak links in the npm supply chain?". `https://www.nzahan.net/research-project`, 2022.

[270] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. Leveraging large language models to detect npm malicious packages. *arXiv preprint arXiv:2403.12196*, 2024.

[271] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. Malwarebench: Malware samples are not enough. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 728–732, 2024.

[272] Nusrat Zahan, Parth Kanakiya, Brian Hambleton, Shohanuzzaman Shohan, and Laurie Williams. Openssf scorecard: On the path toward ecosystem-wide automated security metrics. *IEEE Security & Privacy*, 2023.

[273] Nusrat Zahan, Elizabeth Lin, Mahzabin Tamanna, William Enck, and Laurie Williams. Software bills of materials are required. are we there yet? *IEEE Security & Privacy*, 21(2):82–88, 2023.

[274] Nusrat Zahan, Shohanuzzaman Shohan, Dan Harris, and Laurie Williams. Do software security practices yield fewer vulnerabilities? In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 292–303. IEEE, 2023.

[275] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. What are weak links in the npm supply chain? In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 331–340. IEEE, 2022.

[276] Jerrold H Zar. Spearman rank correlation. *Encyclopedia of biostatistics*, 7, 2005.

[277] Eric Zelikman, Yuhuai Wu, and Noah D Goodman. Star: Self-taught reasoner. *arXiv preprint arXiv:2203.14465*, 2022.

[278] Chenrui Zhang, Lin Liu, Jinpeng Wang, Chuyuan Wang, Xiao Sun, Hongyu Wang, and Mingchen Cai. Prefer: Prompt ensemble learning via feedback-reflect-refine. *arXiv preprint arXiv:2308.12033*, 2023.

[279] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.

[280] Qihuang Zhong, Liang Ding, Juhua Liu, Bo Du, and Dacheng Tao. Can chatgpt understand too? a comparative study on chatgpt and fine-tuned bert. *arXiv preprint arXiv:2302.10198*, 2023.

[281] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. What the fork: a study of inefficient and efficient forking practices in social coding. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 350–361, 2019.

[282] Yaqin Zhou and Asankhaya Sharma. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, pages 914–919, 2017.

[283] Henry Zhu. eslint-scope attack. `https://gist.github.com/hzoo/51cb84afdc50b14bffa6c6dc49826b3e`, 2018.

[284] Thomas Zimmermann et al. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540, 2008.

[285] Zimmermann, Markus, et al. Small world with high risks: A study of security threats in the npm ecosystem. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 995–1010, 2019.