Master's Thesis

Integrating Trust in the Worldwide Software Ecosystem: A Practical Tool for Enhanced Package Security

Angel Temelko
8221113

Supervised By:

Supervisor: Dr. Slinger Jansen
Co-Supervisor: Dr. Siamak Farshidi

Utrecht University

Faculty of Science
Master Computing Science
Utrecht University
The Netherlands
December 22, 2023

Abstract

The landscape of open-source software development is significantly enhanced by tools that enable developers to evaluate the trustworthiness of software packages. A recent initiative in this realm focuses on providing trust assessments for software packages, thereby bolstering the security and reliability of open-source communities. This initiative has led to the creation of a command-line tool, designed to integrate seamlessly with popular package management systems. The tool is particularly innovative in its approach, offering both pre-installation and post-installation analysis, along with policy-based evaluations and comprehensive package research capabilities. Feedback from the interview study involving 20 developers has been predominantly positive, though there are suggestions for improvement regarding the data sources used. This development marks a significant step towards integrating enhanced security measures into everyday open-source software practices.

The tool developed for this study can be accessed at:
`https://github.com/angeltemelko/TrustSECOjs`

Contents

Chapter 1

Introduction

A "software ecosystem" represents a collaborative and interconnected environment in which a variety of actors, both internal and external to an organization, engage to develop software systems[10]. Historically, companies primarily focused on developing proprietary software for their use. However, recent strategies demonstrate a shift towards greater openness. Today's businesses are actively seeking contributions from broader communities, encouraging them to enhance and refine their software. This approach not only sparks innovation by sharing enterprise software with external groups but also benefits from the expertise these communities bring. This mutually beneficial relationship offers shared advantages [8, 51, 33].

Open-source software (OSS) systems prioritize code transparency, allowing software stakeholders to access and share it. This model heavily relies on trust, with users expecting the community to provide reliable and safe software [33]. However, there is an inherent risk. In every community, a few might have malicious intentions. When examining platforms such as npm, the network of relationships can be likened to a complicated web. Given this interconnectedness, vulnerabilities in one area can quickly affect many others. For instance, an issue in a single software product can ripple to others that depend on it [77]. It is noteworthy that 40% of the open source software on npm has known vulnerabilities [77]. Similarly, in Java's open-source realm, about one-third are vulnerable [69].

This raises the question: **How safe are we within the open-source community?** The answer to this question is complex and multi-faceted, rather than a straightforward affirmation or negation. It is vital that we use software that has undergone checks by trusted organizations, in a manner similar to how Google reviews apps before permitting them on the Play Store. Google Play uses a mix of automated tools and human checks to maintain app and software engineer quality. Google Play Protect (GPP) defends users from possible threats by checking apps, rating their safety, and using ongoing device checks. This involves techniques such as machine learning and both dynamic and static assessments [22]. In contrast, package managers, e.g., npm, yarn, lack a strict review process. While people can flag issues, this might not be sufficient [77].

The issues discovered in Log4j clearly showed the security risks tied to open-source software. This flaw allowed harmful strings to be added to the popular Log4j library, enabling unauthorized remote code actions. The so-called Log4Shell problem was not only tough to address but also came with high costs [32, 29]. Events such as this underline the urgent need for tools that shield both users and software engineers from dangerous open-source software, highlighting the significance of proactive security measures in software development.

Most organizations' codebase is made up of 80% to 90% open-source software [60, 76]. This underscores the importance of having tools to guard against non-trustworthy open-source

software. Regular users, whether they are using a mobile app or an npm library, pose a consistent challenge. They often overlook security warnings, whether on command lines or phones, and do not treat them as urgent.

For instance, a large number of software engineers view updates and vulnerability fixes as extra tasks, not essentials. Surprisingly, 69% of software engineers are not aware that their codebases have security issues [45]. Similarly, about 64% of the general public feels they are not accountable for their smartphone's security [44].

TrustSECO is a system that allows users to obtain trust metrics for their third-party packages, thereby bolstering the trustworthiness of the SECO. Further details on TrustSECO will be elaborated in the subsequent sections. In light of this context, our objective is to enhance the functionality of the TrustSECO tool by integrating it within a software package manager. Specifically for this thesis, it will be integrated with npm package manager. Our objective is to send users timely alerts before the libraries are downloaded. We think that by taking a proactive approach, users will be more likely to act responsibly and avoid potential threats rather than having to deal with them after integration.

1.1 Problem Statement

TrustSECO has emerged as a beacon of trust in this complex landscape, providing a community-managed infrastructure to gauge the trustworthiness of various software packages and projects [33]. There is a clear gap even though this initiative has made significant progress in building trust within SECOs. The majority of JavaScript software engineers get their software from package managers, such as npm, and TrustSECO is currently not integrated with this system. Users may not be aware of potential risks if TrustSECO's trust ratings are difficult to see directly on these platforms or may need to take extra steps to access them.

Additionally, due to the dynamic nature of the software engineering industry, characterized by constant updates, patches, and new releases, trust metrics must be flexible, frequently updated, and simple to access. Without a successful integration, the larger digital ecosystem and particular users are in danger. A compromise in one package spreading to numerous connected systems can cause widespread vulnerabilities [77].

Intriguingly, while fast fixes are common, about 85.72% of patching releases for npm vulnerabilities come with unrelated changes, causing delays in their adoption by users. Only 21.28% of users update promptly [15]. Moreover, around 25% of npm dependencies and 40% of its releases fall behind, skipping crucial updates. These delays termed technical lags, often last between 7 to 9 months, with updates taking 12 to 22 days. Surprisingly, most releases start off with this delay, which usually deepens over time. Even updates touted as backwards-compatible often contribute to this delay, raising questions about their ease of adoption [21].

While the majority of packages in the npm package ecosystem primarily comprise JavaScript source files, some require additional steps during installation, such as configuring files or compiling code. npm permits packages to incorporate shell scripts to support these features, streamlining required tasks. But as the Twilio-npm package demonstrates, this adaptability can also be used for evil purposes. It may have appeared innocent, but it hid an install script that covertly started a reverse shell to a remote server [72]. The damage that such scripts can cause is significant, given that they frequently run with user-level permissions and occasionally even with administrative rights. This emphasizes the critical requirement for preventative measures to stop such untrustworthy packages from entering our projects.

Thus, addressing this integration challenge is crucial to strengthening the security architec-

ture of digital ecosystems as well as the user experience. It is critical to investigate techniques and strategies to improve the connection between TrustSECO and package managers to simplify the trust verification process and guarantee that users can make informed decisions about the software they use.

Our research, through a literature review will also explore how software engineers typically interact with package managers. Further, our research will analyze the existing security tools available and identify potential gaps. We will also gather feedback directly from users. Through conducting interviews, our objective is to ascertain whether users would perceive these security features as beneficial or if they would prefer their absence.

1.2 Thesis Layout

The remainder of the thesis is organized as follows. In **Chapter 2**, we elucidate our research design, elaborating on methodologies such as the Systematic Literature Review, Design Science, and Interview study while also justifying their inclusion. **Chapter 3** reviews existing literature on software trust systems. It contrasts TrustSECO with other notable trust tools in the npm ecosystem, investigates software engineers' security practices and behaviors, and examines their reliance on third-party libraries. This section also outlines the meticulous process of sourcing and categorizing relevant literature, underlining pivotal findings, and spotlighting research gaps. In **Chapter 4**, we delve into the creation of TrustSECO.js, highlighting crucial user experience design elements, major design components, and the logic underpinning them. **Chapter 5** details the systematic approach employed in organizing and conducting our research, specifically through an interview study. This chapter showcases the results obtained, captures user feedback from the interviews conducted, and offers an in-depth analysis of the findings. **Chapter 6** provides an exhaustive evaluation of our discoveries, aligning them with insights from our literature review, emphasizing implications for software ecosystems, and proposing future directions. Concluding the thesis, **Chapter 7** reviews the research objectives, summarizes the main conclusions, and recommends directions for future research.

Chapter 2

Research Approach

This research predominantly employs three methodologies: the Systematic Literature Review, Design Science, and Interview study. Each method provides a unique perspective and can address one or more of the research questions formulated. While some questions may benefit from a singular approach, others might require a combination of methods to gain a detailed understanding.

2.1 Research Questions

The primary research question for this thesis arises from the problem statement:

> **MRQ:** "How can package managers reinforce trust within the worldwide Software Ecosystem?"

To comprehensively address the MRQ and provide structured insights, it has been subdivided into four additional research questions:

> **RQ1:** What kind of trust reinforcement mechanisms exist in package ecosystems?
>
> **RQ2:** How effective are trust reinforcement mechanisms in package ecosystems?
>
> **RQ3:** How do software engineers perceive the balance between adding new features and ensuring security while using third-party packages?
>
> **RQ4:** How can third-party trust score tools be effectively integrated into a widely-used package manager through the development of a command-line interface?
>
> **RQ5:** How useful is the implemented tool from an expert's perspective?

2.2 Methodology Mapping to Research Questions

To provide clarity on which method addresses which research question, below is a representation of it:

The **Systematic Literature Review** provides foundational insights and reveals existing knowledge gaps. It is instrumental for the RQ1, RQ2, RQ3. **Design Science** focuses on the practical application and development of solutions, thus relevant for RQ4. Lastly, **Interview study** validate the practicality and efficacy of proposed solutions, making them critical for RQ5.

Research Questions	Systematic Literature Review	Design Science	Experiments
MRQ		X	X
RQ1	X		
RQ2	X		
RQ3	X		
RQ4		X	
RQ5			X

Table 2.1: Mapping of research methods to research questions.

2.3 Research Methods

In this study, we conducted a systematic literature review(SLR), design science, and Interview study, which will help us answer our research questions. Every research question can be answered through 1 or more of these methods.

2.4 Literature Review

We conducted a thorough SLR to better grasp the challenges and possible solutions associated with existing npm security tools. Our goal was to delve into documented experiences and findings. Specifically, we were keen to learn about the motivations behind choosing third-party packages, software engineers' responses to warning messages, and their overall understanding of security issues.

The main aim of this review was to pinpoint prevailing trends, methods, and concerns in trust tools for the present npm environment. Furthermore, we sought to understand the complexities of integrating SECO into platforms such as npm. By analyzing earlier studies, our intention was to spot any overlooked areas and steer our research to address them.

2.4.1 Understanding Systematic Literature Reviews

A group of researchers explains the nature and purpose of SLRs in the paper titled "Systematic literature reviews in software engineering – A tertiary study" [42]. According to them, SLRs serve to combine knowledge on a software engineering topic or research query in a manner that is unbiased, transparent, and reproducible. Such reviews are labeled as secondary studies, while the ones they assess are termed primary studies. There are primarily two types of SLRs:

- **Conventional SLRs**
- **Mapping Studies**

We are using the systematic review framework outlined by Kitchenham, Barbara in "Procedures for performing systematic reviews"[41].

2.4.2 Steps in the framework

- **Problem Formulation and Research Question:** It is important to define a specific problem that needs to be investigated in detail. The researcher identifies the topic and establishes its parameters in this initial stage of the research process. The problem must be

accurately described in order to generate a research question. The purpose of the study is to provide an answer to this straightforward and concise question. Usually, a single, larger research question is followed by several smaller ones.

- **Search Strategy:** To identify relevant academic papers for our study, we explored four esteemed libraries: IEEE Explore, Springer, ScienceDirect and ACM Digital Library. Our primary focus was on papers from 2017. By examining these initial papers, we developed specific search strings to streamline our literature search. Using these strings, we further extracted data and performed targeted searches in the libraries. After accumulating a substantial collection, we systematically organized the findings in a spreadsheet. Each paper was then screened for its relevance to ensure alignment with our research objectives.

- **Inclusion/Exclusion Criteria:** After screening, we use certain guidelines to pick papers for our study. We look at their relevance, studies that were not books or gray literature, and studies that were in English. It is essential to find a balance: if we are too relaxed in our choices, we might include lower-quality papers, but if we are too strict, our findings might not apply broadly.

- **Quality Assessment:** When examining studies, determining their quality is an important step. It is not just a matter of choosing which studies to include; you also need to consider how reliable and relevant they are. To do this, you must consider the type of research that was conducted, the methods used to gather the data, and the clarity of the conclusions reached. It is used to assess a paper's level of quality.

- **Data Extraction:** The step after the quality assessment is to extract the relevant dataset.

- **Analyzing and Synthesizing Data:** We then delved deeper into understanding the data we had gathered.

- **Results:** Presenting the findings of the SLR and answering our preliminary research questions.

2.5 Design Science

Hevner, March, and their team [31] say that Design Science has two main parts: the process of designing (how you design something) and the result (what you create at the end). Essentially, Design Science resembles a feedback cycle that connects understanding a problem to finding its solution. Through repeated cycles, the final product becomes better tailored to the problem.

Based on principles from Hevner & Chatterjee [30], the main idea is that Design Science helps create new solutions that can be applied in real-world settings. In fields such as information systems, these solutions could be in the form of models, methods, algorithms, or even usable software. The information system research framework can be seen in Figure 2.1.

Cycles of the framework:

1. **Relevance Cycle**: In this phase, the main problem is identified, and the standards for a good solution are set. The goal is to see how the finished product will help in real-world situations.

2. **Design Cycle**: Here, the focus shifts to building a solution (or prototype) aligned with the requirements from the Relevance Cycle. Feedback loops ensure continuous refinement, adapting the design based on evaluations.

3. **Rigor Cycle**: This final cycle evaluates the solution's efficacy, ensuring it meets the benchmarks set earlier. Researchers will assess the solution against set standards, often through experiments or other validation techniques.

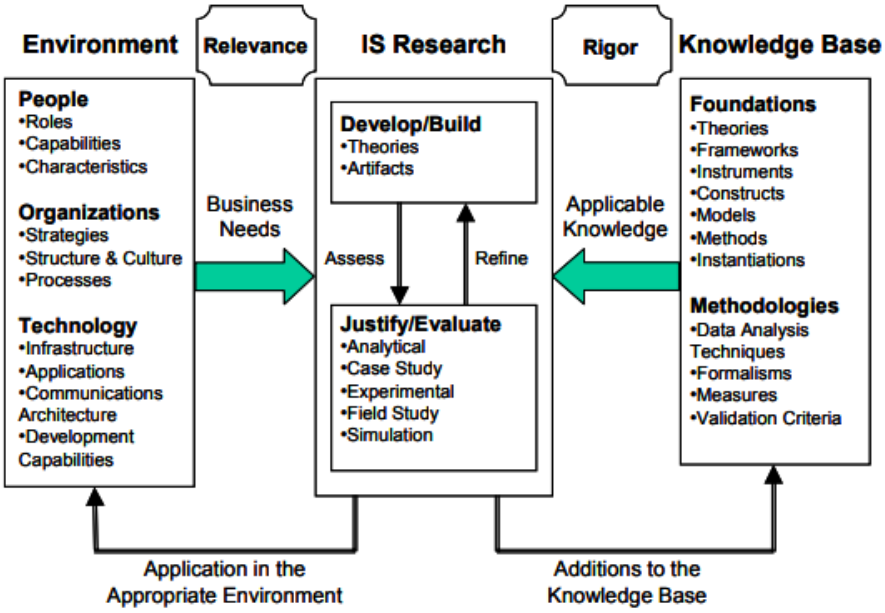

Figure 2.1: Information Systems Research Framework by [31].

2.6 Interview study

Survey research, as defined by [13], revolves around gathering data from a selected group by asking them a set of questions. In essence, it offers insights through the responses of the participants. In our study, this research approach will be instrumental in obtaining feedback on the npm integration. It will encompass in-depth interviews targeting JavaScript software engineers.

Taherdoost [66] highlights the detailed nature of conducting interviews, noting the importance of designing clear, open-ended questions that allow for in-depth responses. He points out the significance of carefully choosing participants and discusses the decision process regarding the format of the interview, whether it be semi-structured or fully structured. According to him, these factors are pivotal in determining the quality of an interview study.

The interview protocol for this thesis is detailed in Section 5.1. This protocol adheres to the principles established by Taherdoost for conducting interview studies, ensuring a comprehensive and methodologically sound approach.

Chapter 3

Literature review

In this section, we will discuss the outcomes of our thorough examination of previous research, which we conducted using a Systematic Literature Review. Our main goal was to locate significant studies that were relevant to our subject and provide the answers to the initial questions we posed. We looked at all of this research to better understand current tools and their effectiveness. Additionally, we wanted to determine whether software engineers were aware of the problems with these platforms.

3.1 Search Strategy

Our literature search strategy employed a mix of both automatic and manual methods. We initiated our search by identifying relevant keywords that align with our research objectives and the problems we aim to address. The core of our investigation lies in the incorporation of software into package managers, emphasizing the associated challenges and limitations.

Given the multitude of package managers for different programming languages, it was vital to specify our focus. We chose the Node Package Manager (npm) as our primary subject, as that is the tool we aim to integrate with.

We have also added “Node Package Manager” and "Node.js" to widen our search, capturing a more comprehensive view of this ecosystem. By including "software engineer and npm" and "software engineer and packages", we hope to gain insights into software engineers’ interactions with npm and packages in general, and also similar tools in the npm ecosystem.

“Best practices” serve to gather research on industry-recognized standards for npm and building CLI tools. Also to help us retrieve best practices in security for picking third-party packages.

In the context of npm packages, terms such as “security”, “trust systems”, “vulnerability scanner”, and “scanner implementation” are crucial for understanding the security protocols. By examining similar trust systems and vulnerability scanners, we can gain insights into the design and operation of these tools.

Considering the primary mode of interaction with the npm is through the terminal, “CLI” and “command line interface” were chosen. These keywords will provide insights into tools, extensions, and best practices software engineers should be aware of when operating within the command-line environment.

The phrase “open source” is crucial. It highlights the clear and community-focused approach of many packages in npm. Both TrustSECO’s open-source structure and our new tool show the importance of this approach in our study.

“Investigating”, “Empirical”, “Characteristics”, and “Analysis” are research-oriented terms aiming to narrow down our search to studies or reports that offer in-depth insights and findings on the subject matter.

The word “survey” helps us gather opinions, possibly revealing what the larger software engineer community thinks about npm and its use of third-party libraries. This is important for understanding their current needs and issues. It also helps us see how software engineers view security.

Lastly, terms such as "dependencies" and "trivial packages" highlight the details of handling package relationships and the challenges of using packages, whether it is trivial or non-trivial packagesdependencies.

npm Query:

```
(
    "npm"
    OR "Node Package Manager"
    OR "Node.js"
    OR "developer and npm"
    OR "developer and packages"
)
AND
(
    "best practices"
    OR "trust systems"
    OR "CLI"
    OR "command line interface"
    OR "integration"
    OR "security"
    OR "vulnerability scanner"
    OR "scanner implementation"
    OR "open source"
    OR "Investigating"
    OR "Empirical"
    OR "Characteristics"
    OR "Analysis"
    OR "developer"
    OR "survey"
    OR "dependencies"
    OR "trivial packages"
)
```

We carefully chose the following databases for our systematic literature review:

- **IEEE Explore**
- **Springer**
- **ScienceDirect**
- **ACM Digital Library**

These databases were picked for their trustworthiness, the range of their content, and their relevance to the topic of our study. We aimed to ensure a comprehensive and varied pool of literature for our review by utilizing the advantages of these platforms.

3.1.1 Search process

We obtained a total of 98 articles from the ACM Digital Library, 250 from IEEE Xplore, 374 from ScienceDirect, and 500 from Springer through our automated scholarly search. Thus, a total of 1,222 articles were produced by our primary sources.

3.1.2 Duplicate removal

We carried out a duplication removal process based on title and publication year to ensure the originality of the papers acquired and to gain a precise understanding of the volume of our acquisitions. 277 duplicates were found in the primary source dataset, and after removing them, we were left with a total of 945 papers.

Interestingly, as shown in Figure 3.1, there was a marked spike in publications during the years 2021 to 2023. This period seems to have been a breeding ground for discussions related to our topic. This also implies that we looked into the most recent trends in the field.

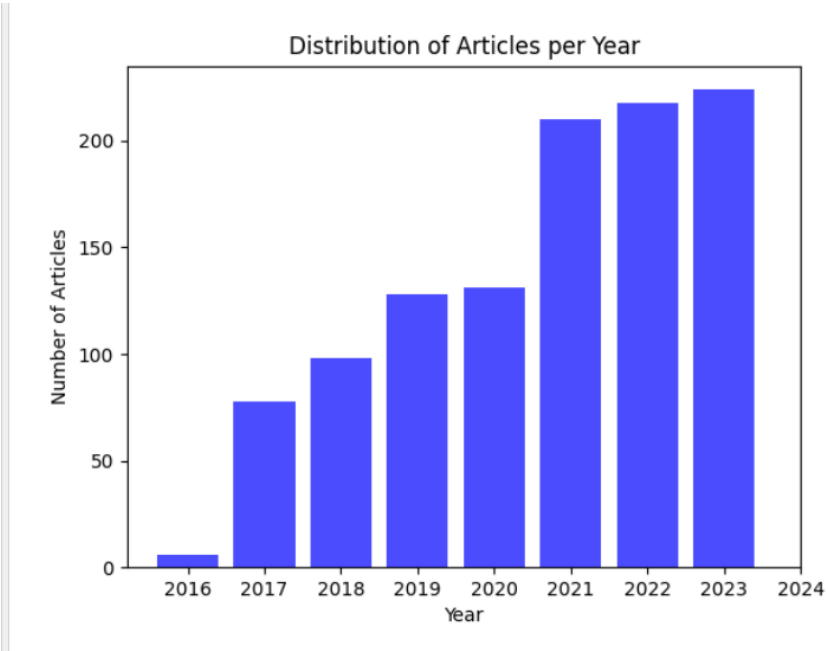

Figure 3.1: Yearly distribution of selected papers in primary source

3.2 Inclusion/Exclusion Criteria

To engage with each paper thoroughly would be neither efficient nor practical given the sizeable volume of more than 945 papers. Establishing strict inclusion and exclusion criteria is essential to speed up the selection process because false positives are a given in any scholarly search.

In the initial round of filtration, a thorough manual review was conducted to narrow down the pool of studies. The criteria for exclusion included:

- Papers that were irrelevant or outside the purview of our research goals.
- Studies that were books or gray literature.
- Studies that were not in English.

For each paper, the title and the abstract were carefully examined, and the relevance of each was determined based on its abstract, leading to its inclusion or exclusion accordingly. During this process, a significant number of false positives were identified, particularly within the Springer database. We speculate that these discrepancies arose due to the database's search mechanism and how it queries data. Nevertheless, through meticulous extraction of relevant papers, we managed to refine our list to 147 papers.

3.3 Quality Assessment

We conducted an evaluation of the publications we had incorporated in the next stage of the Systematic Literature Review (SLR). We only used journal articles and conference proceedings as our primary sources, which came from reputable academic libraries. Despite the reputation of our primary libraries, it remains paramount to ensure the quality of each chosen publication. Thus, our evaluation criteria emphasized several key attributes:

1. Addressing at least one research question.
2. Giving a clear statement of the study goal.
3. Articulating clear and coherent findings.
4. Presenting a well-defined problem statement.
5. Focus on Third-party libraries.
6. Focus on vulnerabilities in npm.
7. Focus on software engineer security behavior.

After conducting a thorough quality assessment process, which involved a manual review of the papers. We were left with a final count of 57 papers, from which we conducted the process of extracting data and subsequently analyzing and synthesizing the collected information.

The whole process of our search strategy can be seen in Figure 3.2

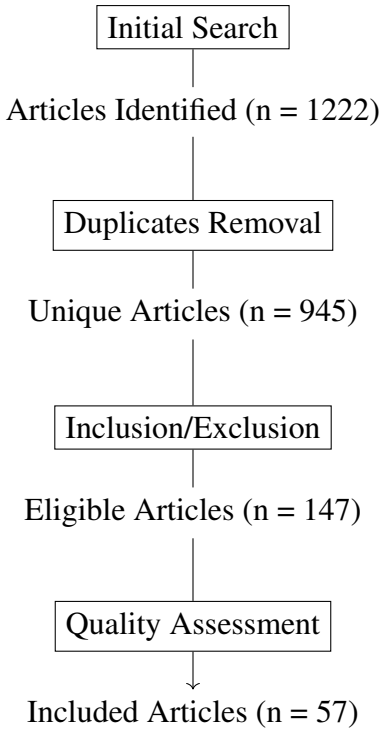

Figure 3.2: Flow Diagram representing the article selection process for the Systematic Literature Review.

3.4 Data Extraction

After completing our quality assessment, we dived into the papers to extract crucial information. This extraction procedure helped us answer some of our research questions. During this extraction, we focused primarily on two areas: Insight into Tools for Vulnerabilities and Trust Checking in the npm Ecosystem and software engineer Behavior and Dependency on Third-Party Libraries.

Insight into Tools for Vulnerabilities and Trust Checking in the npm Ecosystem: This served as the foundation for RQ1 and RQ2 responses. Our investigation aimed to comprehend the reasons behind and goals of these studies—what issues were they trying to address? We looked at their methodology choices to see if they used exploratory case studies, empirical studies, or other techniques. To gain a thorough understanding of the focus of these studies, a closer examination of the specific features they measured was also conducted. The focus then turned to their measurement metrics: Were they looking at code injections, transitive vulnerabilities, malicious packages, policy validation, or some other features? We tried to understand how their tools worked. To do this, we identified the methods they used, such as machine learning, static analysis, dynamic analysis, or some other technique. To assess the robustness and effectiveness of these tools, we also extracted their effectiveness metrics and results. Additionally, we made note of studies that explicitly stated their limitations because we thought it was important for our research. Finding out whether tools operated before or after installation—i.e., whether the package was evaluated before or after integration—was a crucial point of investigation. All of the data extracted can be seen in Table 3.1

Software engineer Behavior and Dependency on Third-Party Libraries: The second round of our investigation focused on how software engineers deal with security issues and how much they rely on third-party libraries. We looked at 11 studies on software engineer

Extraction	Description
Title	The title of the paper
Year	The year of the publication
Authors	The authors of the paper
Source	Which library was selected from
Keywords	Keywords of the publication
Goal of the study	What was the goal of the paper
Research method	The research method of the paper
Names of the tools	Name of the tool
Tool Description	How is the tool described
Tool purpose	What is the purpose of the tool
Features they measure	Specific features they measured
Measurement	Measurement metrics
Tool type/Approach	What approach they used for the tool
Effectiveness Metrics	How effective is the tool
Results	Results that are not contained in the Effectiveness metrics
Limitations	Limitations of the tool
Pre-install / Post-install	Is the tool checking packages before or after installation

Table 3.1: Summary of Extracted Data for existing tools

behaviour and reliance on outside libraries. Understanding the research methods used was a crucial component of our review since it gave us insights into the procedures used to examine software engineer behaviours and third-party library dependencies in light of the chosen study goals. The software engineer's behaviour and Third-party libraries extracted data reliance can be seen in Table 3.2 and Table 3.3 respectively.

Attribute	Description
Title	The title of the reviewed document or article.
RQ	The research questions posed in the document.
Year	The year in which the document was published.
Authors	The authors of the document.
Source	The source where the document was published.
Keywords	Keywords associated with the document or study.
Goal of the study	The main objective or purpose behind the study.
Methodology	The methodology used in the research or study.
Software engineers' Perceptions	Perceptions and practices of Software engineers as found in the study.
Actual Security Outcomes	The real security results or outcomes as reported in the study.
Factors Influencing software engineers	Factors that influence the software engineers' decisions or behaviors.
Recommendations/Best Practices	Recommended best practices or suggestions made in the document.
Most Common Perception	The most prevalent perceptions among software engineers as reported in the study.
% of software engineers (if available)	Percentage of software engineers that hold a certain perception or follow a practice.
Most Adopted Practice	The most commonly adopted practices by software engineers.
Key Awareness Factor(s)	Major factors or events that raise awareness among software engineers.

Table 3.2: Summary of Extracted Data for software engineers

Attribute	Description
Document Title	Title of the article.
RQ	Research questions posed.
Year	Publication year.
Authors	Authors of the articles.
Source	Publication source or journal.
Keywords	Associated keywords.
Goal of the study	Main objective of the articles.
Methodology	Research methodology used.
Reasons for Using Packages	Reasons software engineers use such packages.
Empirical Data on Packages	Data derived from analyzing packages.
Consequences of Using Packages	What are the consequences of using such packages
Packages Studied	Total count of packages.
Packages with Tests	Count of packages that have tests.
Packages with High Dependency	packages with a significant dependency count.
Reasons for Using Packages	Listed reasons for package adoption.
software engineers' Awareness Level	Awareness level of using packages.
software engineers' Adoption Rate	package adoption among software engineers.
packages in open source	Details packages in open-source.
Reasons for Avoiding Packages	Reasons software engineers avoid certain packages.
Recommendation Tools	Indication of tool requirements to recommend packages.

Table 3.3: Summary of Extracted Data on (Trivial) Packages

3.5 Results

Upon completing the data extraction, we started by analyzing the data and answering our research questions. The outcomes have been categorized into three significant sections: Trust Reinforcement Mechanisms in npm Ecosystems, software engineers' security practices and behaviours, and utilization of third-party libraries. In the subsequent subsections, a thorough discussion on each of these categories will be conducted separately.

3.5.1 Trust Reinforcement Mechanisms in npm Ecosystems

Methodology	Occurrences
Empirical Study	16
Design science	4
Case study	2
Exploratory study	1
Corpus analysis / Tool evaluation	1

Table 3.4: Occurrences of Research Methods in npm

There are 25 papers discussing tools for npm that can check for security-related features such as injections, vulnerabilities, and malicious code. Key information about these tools is presented in Table 3.7. To keep the analysis concise, only the essential fields have been included, while other fields, such as keywords, have been left out to avoid over-complicating the process. The majority of the papers, 16 to be exact, were empirical studies, as can be seen in Table 3.4. As shown in Table 3.5, most of them employed approaches for static and dynamic analysis to check for vulnerabilities. The tools primarily measured vulnerabilities, including taint-style, taint-flow, injection, and security issues. Others, such as Latch[72], took a policy-based approach and prevented users from installing particular packages based on their policies instead of looking for vulnerabilities. Latch is a system designed to mitigate risks associated with install-time software supply chain attacks in the npm ecosystem. Latch primarily focuses on mediating the install-time capabilities of npm packages through an innovative permission system. The tool flagged 100% of the tested malicious packages and maintainer policies, it is interesting that latch is a pre-installation tool, meaning that it will catch the problem before the user can install the application. 82% of tested potentially unwanted packages, 100% of tested malicious packages, and 1.5% of all npm packages are blocked by it.

Two further tools that are associated with permission systems are Demo[58] and Lightweight Permission System[24]. The Demo tool is employed to identify zero-day vulnerabilities inside third-party libraries that are not detectable by npm audit and Snyk. This is achieved by acquiring proper authorization and doing scans on the libraries subsequent to their installation. Static and dynamic analysis are employed to ascertain the necessary permissions for the seamless functioning of these libraries. The study highlights that the exclusive reliance on established tools such as Snyk test and npm audit may overlook possible vulnerabilities, particularly those that are not yet identified. The efficacy of their technology surpasses that of both Snyk and npm. The Lightweight Permission System is designed to provide sandboxes for node.js packages, namely those that do simple calculations, in order to prevent them from accessing security-sensitive resources. The tool has a dual purpose, functioning as both a pre-installation and post-installation mechanism. During the pre-installation phase, software engineers declare the rights required by their packages. Subsequently, the system enforces these permissions to

Tool Type / Approach	Occurrences
Static analysis	10
Dynamic analysis	6
Taint analysis	2
ML	2
Anomaly detection	1
Query (GitHub Advisory Database)	1
Build automation	1
Data analysis and modeling	1
Benchmark suite	1
CLI (not specifically mentioned)	1
Analytical tool	1
Pattern recognition	1
Hashing and content comparison	1
Graph-based approach	1
Blockchain and smart contracts (Ethereum)	1
Code-centric approach (not specifically mentioned)	1

Table 3.5: Occurrences of Tool Types and Approaches

guarantee that packages remain within their designated boundaries. Furthermore, installation prompts serve as reminders to software engineers regarding the declared permissions. The permission system that has been presented has the potential to significantly decrease the amount of work needed to review updates in the analyzed scenario. This reduction in effort can range from 6% to 52%.

Visualization tools, such as V-Achilies [37], leverage the GitHub advisory database to generate graphical visualizations of transitive dependencies. This means the tool not only visualizes the immediate dependencies of a project but also the dependencies of those dependencies. Notably, V-Achilies functions as a post-installation tool. The tool successfully detected vulnerabilities in four of the ten most highly regarded npm projects, namely sinopia, cnpmjs.org, windows-build-tools, and npx. The vulnerabilities exhibited a range of severity and were categorized based on their direct or transitive nature.

The npm-miner[12] is a tool that has similarities with V-Achilies, although it differs in its absence of visualization capabilities. The primary approach employed by this application for the crawling and analysis of npm JavaScript packages is static analysis. It is important to highlight that the primary data source utilized by the system is GitHub. In the present context, it is noteworthy to include npm-filter[6] as an additional tool of significance. While it utilizes GitHub data in a similar manner, its primary objective diverges as it focuses on extracting dynamic metadata from program execution. This includes characteristics, namely testing mechanisms, code coverage analysis, and performance metrics. npm-filter is a build automation tool that facilitates the installation, building, and testing of applications within a Docker[1] environment. This tool ensures a regulated and standardized setting for these tasks. It is noteworthy that npm-filter operates as a pre-installation tool, whereas npm-miner operates within a post-installation context.

Another static analysis tool is OpenSSF[74] seeks to enhance the security of open-source software (OSS). It functions as an automated tool specifically developed to examine the secu-

[1]https://www.docker.com/

rity status of software packages. OpenSSF utilizes GitHub metrics to evaluate the health of a package before installation. This evaluation is based on various factors such as code reviews, vulnerabilities, licenses, and other features offered by GitHub. The evaluation conducted by the Scorecard tool provided valuable insights into the security procedures throughout the NPM ecosystem. In particular, the documentation of licenses in their respective repositories was found to be present in just 68% of npm packages. In relation to permissions, the Token-Permission metric revealed that 84.4% of npm packages exhibited optimum file permissions inside their GitHub processes. Nevertheless, there were also apprehensions. According to the findings of the Scorecard, it was determined that around 15.6% of npm repositories were found to have *yaml* files that possessed write access rights. This discovery highlights a possible security vulnerability that may be exploited. Moreover, it is worth noting that a significant proportion of npm packages, specifically 30%, lacked a legal license inside their respective GitHub repositories. In relation to the implementation of security measures, it was found that a significant proportion of npm packages, namely 69%, did not consistently adhere to Code-Review procedures. Furthermore, a substantial majority of 86% of these packages showed that they were not subject to regular maintenance. Furthermore, it was observed that around 90% of npm packages exhibited a lack of implementation of default Branch-Protection and Security-Policy standards inside their repositories. The Scorecard tool utilizes recognized security metrics to assess the security posture of repositories within the npm ecosystem.

Within the domain of static analysis tools, LastJSMile[64], which draws inspiration from LastPyMile[67], has been developed to find code injections in malicious npm packages by discerning inconsistencies between the source code and the package. It has been observed that this tool exhibits a performance improvement of 20.7 times compared to the git-log technique. In the case of authentic npm packages, the tool's regulations resulted in several incorrect notifications. The following static analysis tool is FAST [39], designed to identify taint-style vulnerabilities in JavaScript packages, with a focus on achieving a balance between analysis scalability and accuracy. The tool identified a total of 182 zero-day vulnerabilities out of the 242 that were examined. The FAST tool had the most favourable rates of false positives (7.2%) and false negatives (5.1%) compared to the other tools that were evaluated.DAPP[40] is designed to automatically identify prototype pollution vulnerabilities inside Node.js modules. Additionally, it is capable of doing parallel analysis on all npm modules. The DAPP system conducted tests on around 75,000 modules, accounting for three-quarters of the total 100,000 modules. The tests revealed an error rate of approximately 26%. Each module was checked by DAPP in around 6 seconds. Nodest[55] is a feedback-driven static taint analysis tool that is designed to detect injection vulnerabilities in Node.js applications. It employs a feedback-driven approach to do static analysis and identify potential injection vulnerabilities inside the codebase. Nodest demonstrated a commendable level of precision by successfully identifying vulnerabilities in 22 out of 25 modules without any instances of false positives. The Jam[56] tool is utilized to generate accurate call graphs for Node.js in order to gain insights into potential security issues. The precision of Jam is found to be around 84.35% on average, while *js-callgraph* has an average precision of 58.64%. The memory rate of Jam is 98.62%, but the recall rate of *js-callgraph* is 48.16%. The call graphs generated by Jam exhibited a higher degree of accuracy compared to those created by *js-callgraph*. The duration of Jam's analysis varied, with *toucht* taking less than one second and *jwtnoneify* taking around 23 seconds, both *toucht* and *jwtnoneify* are nodejs packages on which tests were conducted. On the other hand, the performance of js-callgraph was significantly lower. The use of the modular method by Jam resulted in expedited analysis for all benchmarks, with completion times of less than one second seen in several instances. All of these tools employed static analysis techniques at some

stage and were categorized as post-installation tools.

During the literature study, two Machine Learning tools were identified in our research. The tool developed by K. Garrett et al [26]. This study presents a novel tool designed to identify malicious package updates within the npm ecosystem. The tool utilizes an analysis of security-relevant features to distinguish between benign and malicious updates. When subjected to testing using recent package updates, the detection model exhibited a significant 89% reduction in the need for manual review. This promising outcome suggests that the tool has the potential to streamline the verification process for software engineers by effectively flagging suspicious updates. Furthermore, the Amalfi[65] system incorporates classifiers, reproducibility checks, and clone detection techniques, enhancing its accuracy during the process of retraining. The results of the 10-fold cross-validation conducted on the basic corpus indicated that all of the models exhibited a notably high level of accuracy. However, the recall of the Naive Bayes and Support Vector Machine (SVM) classifiers was lower. This can be attributed to the fact that the dataset had a naturally low proportion of dangerous packages.

We already outlined the tools, Lightweight permission system[24], Demo[58], and Latch[72] functions in the context of dynamic analysis. We also have three more dynamic analysis tools. The first is NodeMedic[11] is a software tool that optimizes the operational procedures of the Node.js ecosystem, namely in the areas of triage, vulnerability verification, and the development of package drivers. NodeMedic is a software solution that specifically addresses the server-side dataflow vulnerabilities known as ACE (Asynchronous Code Execution) and ACI (Asynchronous Code Injection). The program revealed vulnerabilities in 40 out of 200 packages, representing a 20% occurrence rate. Additionally, 85% of the reported vulnerabilities were effectively exploited with tailored exploits. The second tool in question is NodeXP[57]. The main objective of the Python program is to automatically identify detection and explanations of Server-Side JavaScript Injection (SSJI) vulnerabilities in Node.js web applications, employing obfuscation techniques to bypass filters and defensive mechanisms. In contrast to other tools, NodeXP was the only tool that successfully identified all the Server-Side JavaScript Injection (SSJI) vulnerabilities present in the apps. The Buildwatch[59], the last tool utilized in the dynamic analysis approach, effectively identifies and mitigates security vulnerabilities arising from third-party dependencies inside the software supply chain. With the exclusion of buildwatch, a tool that is linked to continuous integration and continuous deployment and might be subject to argument as either a tool used after installation or before installation, the aforementioned dynamic tools are all categorized as post-install tools.

We also discovered benchmarking tools during the data extraction process, such as SecBench.js[7], an executable security benchmark tool for server-side javascript. Contains flaws from advisory databases that cover threat classes, for instance, code injection and path traversal. The benchmark contains 1,244 assertions, averaging 2.07 assertions per exploit.

There are two tools that serve as comprehensive alternatives to npm, effectively replacing or updating the whole npm. The first tool Maxnpm [62] serves as a substitute for npm and provides a configurable and efficient approach to resolving dependencies. Combines multiple objectives during installation, such as minimizing vulnerabilities and code size. The MAXnpm tool exhibits enhanced efficacy compared to the conventional npm approach, particularly in its ability to mitigate vulnerable dependencies by 30.51% and provide newer package solutions with an average improvement of 2.62%. The second tool is by D'mello et al. [23]. The solution makes use of smart contracts and Ethereum's blockchain technology. The technology decentralizes the administration of software packages. In addition to providing an immutable, transparent trace of software provenance, it seeks to avoid potential vulnerabilities discovered in centralized systems. Solidity-based smart contracts are used to manage and store data. They

operate in a decentralized environment, with package uploads taking place on peer-to-peer storage. The tool is intended to replace conventional package management systems completely.

In addition, we encountered analytical tools such as DepReveal[5], which were designed to examine and comprehend the effects of dependency vulnerabilities in Node.js applications. The fundamental objective of these tools was to enhance software engineers' consciousness about security vulnerabilities present in their dependencies. Out of a total of 200 packages, the system successfully found 40 packages (20%) that included vulnerabilities. Additionally, it was able to efficiently create exploits for 85% of the identified vulnerabilities. In addition, we discovered pattern recognition tools, such as the Vulnerability Detection Framework[43], which was employed to identify instances of prototype pollution and ReDoS. The precision rate for detecting prototype pollution was 92%, while for ReDoS it reached 97%.

Affogato [27] is a runtime Detection of Injection Attacks for Node.Js using dynamic grey-box taint analysis, it detects all vulnerable flows, meaning it has high recall, and produces no spurious flows, which translates to high precision. Specifically, it detected all vulnerable flows in the given benchmarks and produced no false positives. Poster [68] tool for detecting malicious code injections in software packages by comparing package repositories with source code repositories using hashing and content comparison approach. The poster needed 12 seconds for processing a source code repository, 0.04 seconds for scanning a suspected artifact, and 33 seconds median execution time for processing the source code repositories, with a 97% accuracy.

The Unwrapper[71] tool is a unique solution designed to tackle the problem of "shrinkwrapped clones" within the npm ecosystem. It effectively detects and identifies instances of shrinkwrapped clones, while also identifying similar npm packages that have been cloned. This tool is highly valuable as it helps to address the potential risks associated with package vulnerabilities. By identifying cloned packages, it becomes possible to recognize instances where vulnerabilities are replicated, thereby emphasizing the importance of utilizing such a tool. The precision of the Clone Detector was determined to be 94%, indicating that out of 100 samples, 94 were correctly identified as genuine positives while six were falsely identified as positives. Additionally, the recall of the Clone Detector was found to be 95.3%.

Plumber [70] is not a regular vulnerability checker, it focuses on the delay in propagation vulnerability fixes in the npm ecosystem, analysis of dependence structures, and package metadata. It provides repair tactics for vulnerable packages. The PLUMBER's efficacy is determined to be 79.8%, indicating that the proposed repair solutions provided by the tool line up with the actions taken by software engineers in the real-world context in over 80% of cases.

The last two tools that we are going to check in this Literature review are the DTReme [48] a graph-based approach tool and an extension of the Eclipse-Steady [16] tool which uses a code-center approach. DTreme deals with the npm ecosystem's vulnerabilities caused by third-party library dependencies. It seeks to correct errors in current approaches that do not take into consideration npm-specific dependency resolution criteria. The DVGraph insights serve as the foundation for the dependency tree-based vulnerability mitigation tool known as DTReme for npm packages. DTReme outperformed npm's official audit fix in handling vulnerabilities in 77 out of 262 projects.

The Eclipse-Steady[16] tool, initially developed for Java and Python programs, was expanded in this study to provide support for JavaScript. The primary objective of this tool is to detect and analyze open-source vulnerabilities inside Node.js applications by employing a comprehensive and code-focused methodology. The identification of occurrences when vulnerable code is repackaged or reused within Node.js applications constituted a fundamental aspect of the tool.

The landscape of tools designed to address npm package vulnerabilities reveals a notable trend. As delineated in Table3.6, the majority of these tools operate at a post-installation stage. Only a limited subset is designed for pre-installation checks, while a few ambitious solutions advocate for a comprehensive replacement of npm as the primary package management system.

Stage	**Occurrences**
Post-install	20
Pre-install	4
Total replacement	2
Pre-install / Post-install	1
Pre-install on CI/CD	1

Table 3.6: Occurrences of Installation Stages

Concerning the metrics used by these tools, there is a pronounced emphasis on assessing various forms of vulnerabilities. Prominently, vulnerabilities related to taint flows, transitive dependencies, and injection attacks are frequently measured. Beyond vulnerabilities, a minority of tools also focus on evaluating the quality and health of packages. Other areas of measurement encompass policy automation, dependency management, and avant-garde techniques and solutions. A comprehensive breakdown of these metrics is detailed in Table 3.8.

The reproducability excel for the Table 3.7 can be accessed at this Excel link

Table 3.7: Summary of the key information of Reinforcement Mechanisms/Tools adopted in npm security checking

Name	Description	Measurement	Type/Approach	Results	Effectiveness Metrics	Limitations	Setup
NAN	Detect malicious package updates in Node.js/npm ecosystem.	Malicious package	Anomaly detection / ML	Demonstrated 89% reduction in manual review effort; identified eslint-scope attack.	89% reduction	preliminary evaluation, missing suspicious packages, untested scalability, no explanation mechanism, and no real-time dashboard for suspicious packages.	Post-install
V-Achilles	Detect vulnerabilities in npm projects, direct and transitive dependencies, introduces graph UI.	Transitive vulernabilities	vulnerability	Identified vulnerabilities in 4 top npm projects; direct and transitive dependencies.	NAN	NAN	Post-install
Latch	Tackle risks of install-time software supply chain compromise with Latch (Lightweight instAll-Time CHecker).	Policy vialation	Dynamic analysis	Blocks 1.5% npm, 82% undesirable, 100% malicious packages; 1.6% workflow impact.	100% of tested malicious packages	limitations in portability across operating systems, potential gaps for sophisticated adversaries.	Pre-install
npm-miner	Analyze npm JavaScript packages, data management layer, worker processes, and web application components.	Quality of its packages based on maintainability and security	Static analysis	Analyzed 2,000 GitHub packages; found 476K errors, 279K warnings.	NAN	NAN	Post-install
npm-filter	Automatically install, build, and test npm packages.	Automation	build automation	Running isolated in Docker can install, build, and test any application	NAN	Supports only GitHub packages.	Pre-install

Table 3.7 continued from previous page

Tool Name	Description	Measurement	Type/Approach	Results	Effectiveness Metrics	Limitations	Setup
OpenSSF	Automate monitoring of open-source software security health.	Health and security package	static analysis and automation tool.	Scorecard: 68% had licenses, 84.4% optimal permissions; some practices lacking.	NAN	Focuses on GitHub metrics, potential false positives for non-GitHub projects, does not account for empty repositories.	Pre-Install
Plumber	Identify and remedy software vulnerabilities in package ecosystems.	Vulnerabilities within packages,	data analysis	PLUMBER: 47.4% positive feedback from major npm projects.	nearly 80%	Applicability limited to npm packages, results may not generalize to other ecosystems, potential omissions in the dataset, manual inspection errors.	Post-install
NodeMedic	NAN	Vulnerabilities, Efficacy of Exploit Synthesis	Dynamic Analysis	NODEMEDIC: Detected vulnerabilities in 20% of 200 packages, 85% exploitable.	85% of the identified vulnerabilities.	focuses on ACE and ACI vulnerabilities, does not address others, limitations in automation and exploit synthesis.	Post-install
SecBench.js	SecBench.js benchmark suite with 600 JavaScript vulnerabilities across major threat classes.	Vulnerabilities	Benchmark suit	SECBENCH.JS: 1,244 assertions, average 2.07 assertions per exploit.	Exploit success via Oracle, 1,244 assertions, avg 2.07 per exploit.	NAN	post-install
Lightweight permission system	It is tailored for Node.js applications to minimize attack surface.	Permission system for sandboxes individual packages	dynamic analysis	Permission system reduces update review effort by 6%-52%.	6% to 52% reduction.	NAN	Pre-install / Post-install
Amalfi	Detect malicious npm packages automatically, addressing vast numbers and updates.	Malicious packages	ML	Models showed high precision; Naive Bayes and SVM had lower recall.	High precision. Low recall	biases in training data affect generalizability, sustainability issues with continuous retraining, and accuracy issues due to short inspection windows.	Post-install

Table 3.7 continued from previous page

Tool Name	Description	Measurement	Type/Approach	Results	Effectiveness Metrics	Limitations	Setup
Maxnpm	A complete, drop-in replacement for npm using PACSOLVE and Max-SMT solver.	vulenrabilities	Basic CLI - not mentioned	MAXnpm: Reduces vulnerable dependencies by 30.51%, slight solving time increase.	33% better audit, 2,62% new packages, 4,37% code reduction, 1.9% less duplication. Worse in tests and slower.	The selection may not represent the entire npm ecosystem, possible bugs in tool results, and evaluation criteria may not align with developer priorities.	Pre-install total replacement
Nodest	A feedback-driven static taint analysis tool for detecting injection vulnerabilities in Node.js apps.	zero-day injection vulnerabilities, taint flows	Static analysis using feedback-driven approach, taint analysis	Nodest: Detected vulnerabilities in 22/25 modules, no false positives.	find vulnerabilities in 22 out of 25 modules	Use GitHub only.	Post-install
DepReveal	A Node.js project analytical tool for dependency discoverability levels on GitHub.	vulnerabilities	Analytical tool	DepReveal analyzed dependency vulnerabilities, and provided insights.	Vulnerabilities detected: 40 out of 200 packages (20%) Exploits successfully crafted: 85% of the identified vulnerabilities.	Semgrep rules tailored to prototype pollution and ReDoS, may overlook other vulnerabilities, and struggle with obfuscated JavaScript code.	Post-install
Vulnerability detection framework	Experimental vulnerability detection framework with Semgrep and textual similarity methods.	vulnerable functions	Pater recognition/ Static analyisis	Detected 18K prototype pollution, 1.7K ReDoS; 92-97% precision.	Prototype pollution: 92% ReDoS: 97%. The precision rate is 98%. Vulnerable functions detected 94,5%	NAN	Post install
Affogato	A dynamic grey-box taint analysis tool combining black-box and white-box analysis.	injection vulnerabilities	taint analysis tool	Affogato: High precision/recall detecting injection vulnerabilities.	high recall, high precision).	NAN	Post-install
Demo	Conduct static and dynamic program analysis on server-side JavaScript third-party libraries.	NAN	Static/Dynamic analysis	Tool surpasses snyk test and npm audit in uncovering unknown vulnerabilities.	More effective than synk and audit	NAN	Post-install

Table 3.7 continued from previous page

Tool Name	Description	Measurement	Type/Approach	Results	Effectiveness Metrics	Limitations	Setup
Poster	Compare distributed software artifacts code with source code repositories.	code injection	hashing and content comparison approach	34 malicious artifacts with code discrepancies; 97% accurate to source.	12 seconds for processing a source code repository.0.04 seconds for scanning a suspected artifact.33 seconds median execution time for processing the source code repositories.	NAN	Post-install
DAPP	An automatic static analysis tool for prototype pollution vulnerabilities in Node.js modules.	prototype pollution vulnerability	static analysis	DAPP found 37 genuine prototype pollution vulnerabilities in 30K modules.	DAPP: 37 true positives, 38 false positives; 25.68% error rate; 6.17s avg. test time.	Significant rate of false positives and false negatives, results require manual verification.	Post-install
NodeXp	A Python tool for detecting and exploiting SSJI vulnerabilities in Node.js applications.	The Server Side JavaScript Injection (SSJI) vulnerabilities in	Dynamic Analysis	NodeXP discovered a 0-day vulnerability in SubleasingUIU app.	compared to other tools only NodeXP detected all the SSJI vulnerabilities in the applications.	NAN	post-install
DTReme Algortihm	A dependency tree-based vulnerability remediation tool for npm packages.	vulnerabilities in dependency trees	Graph-based approach (namely terms Graph Tree and DVGraph).	DTResolver 90.58% accurate in dependency resolution; outperforms npm-remote-ls.	vulnerability detection, DTResolver achieved coverage of 98.1% while npm-remote-ls had 97.7%.	Overlooks indirect dependency vulnerabilities, CVE mapping errors, missing dependencies, and high-risk version exclusions	post-install
Eclipse-Steady	Detect, assess, and mitigate vulnerabilities in open source dependencies.	vulnerabilities	code-centric approach - Not mentioned	NAN	Analysis covered 42 out of 65 apps; application constructs outnumber dependency constructs by a factor of three (75 vs. 26).	NAN	Post-install

Table 3.7 continued from previous page

Tool Name	Description	Measurement	Type/Approach	Results	Effectiveness Metrics	Limitations	Setup
NAN	Manage package interactions via smart contracts, with decentralized verification and a tree data structure.	decentralizing package management using blockchain technology and smart contracts.	Ethereum's blockchain and smart contracts	NAN	Metadata latency: 148 ms; Peak bandwidth: 650.48 Kbit/s; Other metadata: 396.32 Kbit/s.	Low gas can delay transactions; worker crashes from fund shortages; single failure point; slow node sync; unstable dynamic structure support.	Total replacement
LastJSMile	Detect code injections in npm packages by comparing source code with packaged versions, similar to LastPyMile for Python	injections in malicious npm packages	Static analysis	New approach is 89.4% faster than git-log; reduces false positives.	Tool speed: 20.7x faster than git-log; False positives: 90.2% for whole artifacts, 21.3% for "phantom" files; Recall: consistent.	Imbalances in the dataset, dependency on GitHub repos, changes in file hashes from code movement, and overlooking direct malicious code commits.	Post-install
FAST	Analyze JavaScript using unique abstract interpretation techniques for efficient vulnerability assessment.	Zero day vulenrabilities, taint-style vulnerabilities	Static analysis / taint style	AST detected 242 zero-day vulnerabilities; 21 CVEs issued.	182 out of 242 zero-day vulnerabilities. FAST-det had the lowest FP(7.2%) and FN(5.1%)	Balancing JavaScript's dynamic features with analytical scalability and accuracy is a key challenge, often leading to a trade-off between the two.	post-install
Jam	Build call graphs for JavaScript modules through summary composition.	Security vulnerabilities.	Static analysis	Jam's precision 84.35%, faster analysis than js-callgraph.	found all 8 vulnerabilities, yielding a 100% recall. reduced false positives by 81% compared to NPM audit. The precision is 61% compared to the 24% precision of NPM audit.	NAN	Post-install
BuildWatch	Analyze software dependencies dynamically.	Malicious packages,	Dynamic analysis	NAN	NAN	NAN	pre-install on CI/CD

Table 3.7 continued from previous page

Tool Name	Description	Measurement	Type/Approach	Results	Effectiveness Metrics	Limitations	Setup
Unwrapper	Detect duplicate NPM packages with a focus on speed and independence from the NPM database.	Clone packages, vulnerabilities in clone packages	file tree structures and file content comparisons.	10.4% of 6,000 NPM packages were clones; potential 178K cloned.	Clone Detector's precision is 94% (94 true positives out of 100 samples, with 6 false positives). Prefilter's recall is 95.3%.	NAN	post-install

Measurement Category	Occurrences
Vulnerability Types	20
- Malicious package	3
- Transitive vulnerabilities	1
- Vulnerabilities within packages	1
- zero-day injection vulnerabilities	1
- Injection vulnerabilities	3
- taint flows	1
- vulnerable functions	1
- prototype pollution vulnerability	1
- Server Side JavaScript Injection (SSJI)	1
- taint-style vulnerabilities	1
- security vulnerabilities	1
- injections in malicious NPM packages	1
- Impact of a vulnerability	1
Package Quality and Health	2
- Quality of packages	1
- Health and security package	1
Policy and Automation	2
- Policy violation	1
- Automation	1
Dependency Management	3
- Dependency relationships	1
- Evolution of dependencies	1
- vulnerabilities in dependency trees	1
Advanced Techniques and Solutions	3
- Efficacy of Exploit Synthesis	1
- Permission system for sandboxes	1
- Decentralizing package management (blockchain)	1

Table 3.8: Summary of Measurement Categories Assessed by Tools for Trust Reinforcement in the npm Ecosystem

3.5.2 Software engineers security practices and behavior

We have analyzed five papers, to understand the security best practices and how software engineers behave towards those securities and practices. 3 papers were Empirical studies, and two papers were Qualitative studies, as can be seen in Table 3.10. The data that we extracted can be seen in Table 3.9

Kabir et al. [38] conducted a study examining three optimal methodologies. In the first step, the user should employ the command "npm audit" to identify vulnerabilities present in library dependencies. Subsequently, these vulnerabilities can be addressed by utilizing the command "npm audit fix". Moving on to the second step, the user should conduct a thorough examination of the packages to identify any unused or duplicated ones. This can be accomplished by employing tools such as "depcheck". Once identified, the user should proceed to delete any redundant packages using the command "npm dedupe". Lastly, to ensure stability and consistency in library dependency versions, it is recommended to enforce the usage of the

Attribute	Description
Title	The title of the reviewed document or article.
RQ	The research questions posed in the document.
Year	The year in which the document was published.
Authors	The authors of the document.
Source	The source or journal where the document was published.
Keywords	Keywords associated with the document or study.
Goal of the study	The main objective or purpose behind the study.
Methodology	The methodology used in the research or study.
software engineers' Perceptions	Perceptions and practices of software engineers as found in the study.
Actual Security Outcomes	The real security results or outcomes as reported in the study.
Factors Influencing software engineers	Factors that influence the software engineers' decisions or behaviors.
Recommendations/Best Practices	Recommended best practices or suggestions made in the document.
Most Common Perception	The most prevalent perceptions among software engineers as reported in the study.
% of software engineers (if available)	Percentage of software engineers that hold a certain perception or follow a practice.
Most Adopted Practice	The most commonly adopted practices by software engineers.
Key Awareness Factor(s)	Major factors or events that raise awareness among software engineers.
% Aware (if available)	Percentage of software engineers who are aware of a certain factor or practice.
% Unaware (if available)	Percentage of software engineers who are not aware of a certain factor or practice.

Table 3.9: Summary of Extracted Data for software engineers

package-lock.json file. This file serves as a lock file and effectively pins the versions of library dependencies. It was discovered that a majority of software engineers do not adhere to best practice BP1, as evidenced by the identification of vulnerabilities in 55% of the projects examined through the utilization of npm audit. In the context of BP2, it has been observed that a significant proportion of projects have a worrisome abundance of duplicate instances. In the context of BP3, it was discovered that a mere 32% of the apps surveyed used the practice of explicitly specifying version numbers for their package dependencies. The researchers also aimed to investigate the underlying causes of the violation of these best practices. Their findings indicate that software engineers recognize the significance of security, yet they express scepticism towards npm-audit due to its high rate of false positives. Additionally, it was shown that software engineers often disregarded or misinterpreted alerts regarding duplicate dependencies. A considerable number of engineers did not assign significant importance to concerns related to duplicate dependencies. Certain engineers underlined the necessity of preserving distinct versions of packages, whilst others expressed doubts over the reliability of depcheck. Many software engineers failed to acknowledge the significance of lock files in ensuring con-

Methodology	Occurrences
Empirical Study	3
Qualitative Study	2

Table 3.10: Occurrences of Research Methods in software engineer Behavior

sistent builds, either due to a lack of understanding of the functioning of the locking mechanism or due to misunderstandings around it.

The study conducted by Zahan et al [75]. aimed to investigate the potential correlation between software security measures and the occurrence of vulnerabilities. The researchers utilized data obtained from the OpenSSF tool [74] in order to investigate the correlation between security practices and the number of vulnerabilities. The researchers incorporated a set of 15 established security practices identified in a prior study. Additionally, they collected data from a substantial number of packages, namely 767,389 npm packages and 191,158 PyPI packages. The vulnerability count was obtained from the OSV and Snyk databases. The researchers discovered that the practices of Security Policy, Maintenance, Code Review, and Branch Protection were identified as the most crucial measures for reducing vulnerabilities. It was discovered that packages possessing enhanced security measures frequently exhibited a higher frequency of reported concerns. This phenomenon may be attributed to the higher frequency of usage and more rigorous testing of these programs. Nevertheless, it remains uncertain if the enhancement of security measures directly leads to the discovery of more vulnerabilities, or if there are other contributing elements at play. Additional investigation is required to provide elucidation on this matter.

In a study conducted by Paschenko et al. [61], 25 software engineers were interviewed in a semi-structured manner to gain insight into their decision-making processes regarding the selection, management, and updating of software dependencies. From the interviews, they found out that software engineers often select third-party packages based on company policies, community support, and the library's core functionality, most interesting finding was that software engineers were mostly always focusing on functionality over security. This means a library's ability to perform its intended function is often valued more than its potential security vulnerabilities. When they updated the dependencies, their motivations were, security concerns, they were updating to mitigate vulnerabilities. However, an interesting fact is that software engineers were choosing stability over security, they were only updating it if they knew that it would not crash the application. They were also avoiding dependency updates due to fear of breaking changes. Organizational policies have a substantial impact on whether software engineers update dependencies, and lastly, software engineers find managing and updating dependencies challenging due to the high number of transitive dependencies, which are often difficult to control. They also discussed mitigating unfixed vulnerabilities, they found out that the first thing a software engineer does when faced with a weak dependency is assess the impact it will have on their project. While they wait for an official patch, some people might choose to disable the compromised functionality temporarily. When a vulnerability arises, knowledgeable people frequently take matters into their own hands, addressing the issue on their own and occasionally adding fixes to the original open-source repositories. It may be practical to switch to another library that offers comparable features in situations where patching the vulnerability is complicated or where the affected library is undersupported.

In their empirical study, Kula et al. [46] sought to investigate the behavior of software engineers in updating their library dependencies and their response to security advisories. This investigation was conducted through the examination of 4659 projects and the running of a

software engineer survey. The researchers discovered that a significant proportion of software engineers are neglecting to update their library requirements. Specifically, of the projects examined, a staggering 81.5% were found to have outdated dependencies. A significant proportion, namely 69%, of the software engineers who participated in the survey shown a lack of awareness of the vulnerabilities present in their software. However, when being notified about these vulnerabilities, they promptly undertook efforts to correct them. The process of updating these libraries is not simple. Such updates are a difficult choice due to the intricate web of library relationships, also known as "dependency hell." The costs and advantages of updating are frequently compared. These updates are often viewed by software engineers as extra work that is best completed when they have free time. The decision to update also depends on the software engineer's workload and how much weight their team or organization gives to such updates. Even when there are newer alternatives, software engineers favor more established, older libraries. The potential impact of the vulnerability and the role of the dependency in the project are frequently taken into consideration when deciding whether to react to a security advisory. It was claimed by some software engineers that the failure to update the vulnerable dependency was due to the inactivity of the project or due to its lack of criticality. Others believed that the vulnerable element's influence on the project's goals was minimal.

In many instances, software engineers tend to emphasize the usefulness of software over its security, especially when security measures have the potential to impact the product's operational capabilities. Ivory et al. [35] found that professionals tend to prioritize feature completion above security due to their demanding schedules and imminent deadlines. It has been shown that software engineers exhibit a prevalent optimism bias, wherein they possess a belief that they can efficiently identify and rectify security vulnerabilities. Indeed, a significant number of security vulnerabilities are typically overlooked. It is noteworthy that certain engineers acknowledge a lack of interest in or familiarity with security, hence rendering their programs more susceptible to vulnerabilities. Moreover, software engineers often rely on their existing expertise, implying that they may only possess an awareness of issues they have already experienced. A significant number of individuals choose heuristic-based coding, a method that may exhibit biases and inaccuracies. They often rely on their friends for guidance instead of consulting security experts, exposing themselves to the potential of receiving inaccurate information. Moreover, certain software engineers may opt to disregard a vulnerability if it poses a risk to the overall operation. Based on the findings of the study, it can be seen that in the absence of a specific motivation, software engineers commonly resort to employing heuristic methods as their default strategy. In many cases, prioritizing functionality is given greater importance than the abstract concept of security due to its tangible nature. Solo software engineers may exhibit a higher degree of idealism and prejudice as a result of limited exposure to contrasting viewpoints. Nevertheless, the prioritization of security by software engineers may be influenced by a company's attention to this aspect.

3.5.3 Third-party library usage

The papers that we found in our quality assessment round for Trivial Packages and Third-party library usage are 9. All of them are Empirical case studies, and a comprehensive breakdown of the research methods can be seen in Table 3.11. The data that we extracted can be seen in Table 3.3.

The initial study conducted by Abdalkareem R. et al. [4] aimed to gain insights into the utilization of trivial packages within Node.js applications. The researchers examined a total of 230,000 NPM packages and administered surveys to Node.js software engineers in order

Methodology	Occurrences
Empirical case study	8
Exploratory study + Empirical case study	1

Table 3.11: Occurrences of Research Methods in Thrid-party libraries

to gather information regarding the usage of third-party libraries and the concept of "trivial packages".Upon conducting an investigation, it was determined that a package is classified as trivial if it contains fewer than 35 lines of code and had a McCabe's cyclomatic complexity of less than 10. More than 50% of the trivial packages examined lacked test coverage, had fewer releases, and 43.7% had at least one dependency. Moving to the software engineers, the researchers discovered that the utilization of third-party libraries and trivial packages is mostly driven by their well-implemented and well-tested nature. These tools contribute to enhanced productivity, maintainability of code, improved readability, and reduced complexity within the application. There were instances in which individuals claimed that it enhanced their performance. It was determined that the software engineer community has a significant level of awareness on the benefits and challenges associated with utilizing trivial packages and third-party libraries.

Abdaklerem R, et al. [3] extended their previous work, included 501,001 packages, and also tested the PyPl together with npm. They found out that trivial package definitions are the same for JavaScript and Python. software engineers of JavaScript and Python see trivial packages differently. In contrast to Python software engineers, who see the usage of trivial packages as a negative practice in 70.3% of cases, just 23.9% of JavaScript software engineers thought it was harmful to use such packages. software engineers predominantly use trivial packages for their well-implemented and tested nature, with 54.6% of JavaScript and 54.1% of Python respondents citing this reason. 47.7% of JavaScript and 32.4% of Python software engineers believe these packages enhance productivity. A minority of respondents, 9.1% from JavaScript and 5.4% from Python held the belief that Well-maintained code is a significant factor. They identified several drawbacks, A significant 55.7% of JavaScript and 67.6% of Python software engineers face "dependency hell". Additionally, there is the risk of application breakage. As noted by 18.2% of JavaScript and 32.4% of Python software engineers. Performance can be impacted too, with 15.9% of JavaScript and 8.1% of Python software engineers citing slower build, run, and installation times. In some cases, instead of speeding up work, trivial packages can cause delays (12.5% JavaScript, 10.8% Python). While 9.1% of JavaScript software engineers mention potential missed learning opportunities, both communities are particularly concerned about security: 8.0% of JavaScript and a notable 13.5% of Python software engineers underscore the vulnerabilities trivial packages might bring.

The paper by M. A. R. Chowdhury et al. [17] also explored the reasons for adopting these trivial packages, they found out that most of the trivial packages are deeply integrated into projects, meaning if the packages are down it will affect the project. The overwhelming influence trivial packages had on both individual projects and the larger npm ecosystem surprised many software engineers. The information sparked reflection, and several software engineers hypothesized that the prevalence of trivial packages might be due to the void left by the lack of a solid standard library for JavaScript. Another interesting observation was that when considering whether to include dependencies, software engineers frequently gave priority to elements such as community acceptance and active project activity, frequently ignoring the triviality. They stressed that software engineers who use these packages should not undervalue their significance. Before incorporating these packages into their projects, they should make sure they

are regularly updated and carefully reviewed. Notably, these unimportant packages continue to be project dependencies after their initial inclusion as well. To reduce dependency risks, software engineers are urged to rigorously assess possible dependencies while taking into account their trivial nature and investigate refactoring or migration approaches.

In their study, Enrique Larios Vargas et al. [47] examined the perspective of software engineers in the process of selecting third-party libraries. A total of 16 software engineers were interviewed, and a survey was conducted with 115 software engineers. The examination encompassed the technical, human, and economic factors that software engineers consider when choosing third-party libraries. In relation to the technical aspect, the software engineers expressed their preference for libraries that are active and maintained long-term, they have regular updates, recent releases, and active contributions, and provide insights into library vitality. The quality elements that have an impact on a library include good documentation, usability, alignment with their architecture, good test coverage, no security vulnerabilities, and good performance. The most fundamental criterion was the extent to which the library fulfilled its project requirements. Another noteworthy observation is that software engineers who initiate projects from the beginning have greater autonomy in choosing libraries, but for ongoing projects, library selection is impacted by the necessity to conform with the existing software.

Xu et al [73] conducted two surveys exploring library reuse and re-implementation to discern why software engineers either substituted self-created code with an external library or vice versa. Their findings revealed that 69.6% of respondents agreed on the commonality of replacing self-implemented code with a library method, with 83.9% admitting to having done this in their practice. Only 39.3% believe that replacing a library method with a self-implemented code is common, but a larger 76.8% have done this in their practice. The reasons for replacing the library method they gave were Improving reliability by 25%, Development efficiency by 24%, Testability by 22%, and Maintainability by 20%. Their criteria for library selection were as follows: 22 out of 34 participants were in Library maintenance and testing, for Library reputation, there were nine participants, Code and documentation readability had four participants, and Stability had four participants. Library size/complexity had three participants, License compatibility had 3 participants and finally, Integration ease had 2 participants. The findings reveal that due to a lack of knowledge of the library or because the library's method was not yet introduced, software engineers frequently replaced their internally implemented methods with those from external libraries. They decide against using their own implementation when they later come across a well-maintained and tried library that meets their needs in the context of why software engineers Replace an External Library Method. With their Self-Implemented Code, they found out that 21% wanted to reduce dependency, 19% wanted to improve flexibility, 18% wanted a simpler solution, and the interesting fact was that only 3% wanted better security.

Mujahid et al [54] conducted a qualitative study on a survey of JavaScript software engineers to determine the qualities that JavaScript software engineers want in npm packages. They surveyed 118 JavaScript software engineers for this study. The researchers discovered that the primary factors influencing the choice of a library include documentation, which received a 93% preference rate, followed by download counts at 85%. The star count, rated on a 5-point scale, averaged at 4.0. Lastly, around 62% of software engineers considered vulnerabilities when making their selection. The parameters that exhibited some degree of significance were release dates, commit frequency, test code, license, dependent applications count, number of dependencies, closed issues count, and number of contributors. Another noteworthy consideration for software engineers when picking libraries was the level of support from prominent companies such as Facebook, Formidable Labs, and Infinite Red. Libraries backed by these

companies were perceived as more trustworthy and dependable. Positive reception is often observed for active community conversations and endorsements from reputable software engineers within the community.

In their study, Chen et al. [14] conducted a survey including 59 Javascript software engineers who actively create trivial npm packages. The objective of the study was to investigate the motivations behind software engineers' decision to publish such packages, as well as their perceptions of the negative repercussions associated with these products. Additionally, the researchers aimed to assess the extent to which these negative concerns may be alleviated. The results obtained from the study indicated that the advantages of publishing simple packages were as follows. The primary focus of this study is on the development of reusable components, which accounts for 64.41% of the overall emphasis. Additionally, the examination also encompasses the evaluation and documentation of these components, constituting 33.90% of the research concerns. The concept of separation of concerns, with a significance of 32.2%, is another crucial aspect that is explored. Furthermore, the study delves into the optimization of these components, which has an importance of 27.12%. Lastly, the study acknowledges the importance of contributing to the community and personal satisfaction, which accounts for 22.03%. When they were asked what the disadvantages were, the respondents predominantly indicated the need to manage various packages, the emergence of dependency conflicts, and the challenge of identifying suitable packages. They proposed grouping their packages, and it will save 13% of the number of dependencies in the ecosystem.

Lopez et al. [49] proposed several metrics for software engineers to consider when selecting libraries:

- **Popularity:** Number of client projects using the library.
- **Release Frequency:** Average interval between consecutive releases.
- **Issue Response And Closing Times:** Average times for issue responses and issue resolutions.
- **Recency:** The date of the latest library update or release.
- **Backwards Compatibility:** Average number of breaking changes per release.
- **Migration:** Frequency of library replacements in client projects.
- **Fault-proneness:** Measured by the number of bug fixes.
- **Performance & Security:** Efficiency of the library's code and its vulnerability.

In a subsequent study, Lopez et al. [20] conducted research aimed at examining the efficacy of software metrics in facilitating software engineers' decision-making process when selecting libraries. The majority of participants expressed that metric-based comparisons were beneficial, as shown by an average Likert scale rating of 3.85. Performance, Popularity, and Security were the top three metrics influencing software engineers' decisions. Their means were 4.08, 4.06, and 4.00, respectively. Other metrics, such as Issue Closing Time and Last Discussed on Stack Overflow were found to be less influential. The software engineers are in need of supplementary metrics pertaining to the usability of documentation and libraries. The significance of specific indicators differs across different areas. For instance, the importance of security varies across different areas.

3.6 Software Ecosystem

Before delving into TrustSECO's background and functionalities, it is essential to understand the concept of the software ecosystem: its definition, the various types of ecosystems, and their mining capabilities. SECOs are defined as *"are sets of actors that collaboratively serve a market for software and services, typically with an underlying technical platform"* [34, 36]. The roles within the software ecosystem include [34]:

- **End-user**: An individual who uses the software to improve their productivity.
- **End-user Organization**: A group of end-users using the software to support their organization's objectives.
- **Software Engineer**: Professionals who develop and maintain software products.
- **Software Producing Organization**: An entity that employs software engineers to develop and maintain software aimed at broad adoption.
- **Package Maintainers**: Software engineers responsible for the upkeep of software packages, often working independently and utilizing platforms like GitHub for storage and maintenance.

Furthermore, they describe the exchanges within the software ecosystem, which consist of:

- **Software Product**: *"A prepared collection of software elements or a service powered by software, complete with supporting materials, made available for a particular market."*
- **Component**: *"An independent module with specified connectors and requirements that can be assembled and implemented on its own."*
- **Library**: A compilation of functions tailored for specific tasks, such as React, which provides functions that facilitate writing client-side code.
- **Package**: A collection that includes software, libraries, and metadata detailing the name, version, and dependencies of the software.

Lastly, the authors elucidate the following ecosystem services in Figure 3.3.

- **Ecosystem Services**: Services that enhance existing software, such as Integrated Development Environments (IDEs) that assist in code writing and debugging.
- **Package Manager**: A tool for the automated management and updating of software packages on a computer system.
- **Package Repository**: A storage server where software packages are hosted and made accessible.

More researchers provide an exploration of the SECO. According to Mens and De Roover, SECO can be viewed from various perspectives, including ecological, economic, technical, and social, each perspective offers a unique context and structure [52]. This understanding is from Manikas[50] that encapsulates all these views, describing a software ecosystem as: *"the interactions of a set of actors on top of a common technological platform, resulting in*

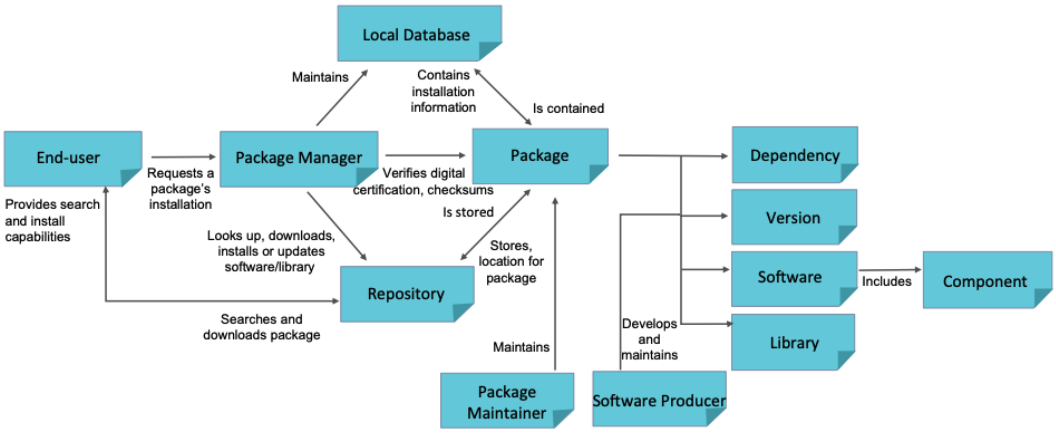

Figure 3.3: Structure of the software package ecosystem[34].

a number of software solutions or services. Each actor has specific motivations or business models and is connected to other actors and the ecosystem through symbiotic relationships. The technological platform is structured to allow the involvement and contribution of the different actors". Additionally, they further categorize software ecosystems into six primary types:

- **Digital Platforms:** These are defined as platforms where the product is owned by a company, such as Apple's App Store, JetBrains' IntelliJ IDEA, or Microsoft's VSCode. They allow third-party apps or plugins to integrate, with contributors being the software engineers of these third-party libraries and their respective users.
- **Social Coding Platforms:** These platforms primarily host software projects. They serve as repositories where software engineers can version, store, and maintain their codebases. Examples include GitHub, GitLab, and BitBucket. In this context, the components are the software projects, and the contributors are the software project software engineers.
- **Component-based Software Ecosystems:** This concept is centered around the reuse of software components, enabling the efficient utilization of previously developed software, thereby saving time and reducing costs. Initially proposed by McIlroy, this idea did not gain much traction during its inception. However, with the advent of cloud computing and the emergence of OSS, it has witnessed significant success. A prime example of such ecosystems is the realm of third-party libraries, which is the focal point of this thesis. Each ecosystem is accompanied by its own package managers and package registries where the software packages are stored. For instance, Python utilizes PyPi, JVM languages use Maven, JavaScript employs npm, and .NET relies on NuGet. The core components of these ecosystems are the interdependent software packages, while the contributors encompass both the consumers and producers of software packages and libraries.
- **Software Automation Ecosystems:** These ecosystems are centred around automating various facets of software management, development, and deployment. Examples include:
  - *Containerization:* Software engineers package components into containers, ensuring uniform behavior across different environments.

  - *Management:* Tools such as Bicep and Terraform, which use Infrastructure as Code (IaC), help automate infrastructure management tasks.
  - *DevOps and CI/CD:* Continuous deployment and delivery tools streamline and automate workflows during deployments. Here, the components are container images and CI/CD pipelines, and the contributors are DevOps professionals.

- **Communication-oriented Ecosystems:** Unlike technical ecosystems, these primarily focus on platforms facilitating communication within software communities. Examples include modern communication platforms such as Slack and Discord, and discussion forums, namely Stack Overflow where software engineers seek and offer advice. The components in these ecosystems are emails, messages, posts, and questions, while the contributors encompass software engineers, end-users, and researchers.

- **OSS Communities:** OSS communities are decentralized groups that maintain open and transparent projects. While they offer the benefits of transparency and openness, challenges such as delayed updates and unmaintained components persist, often because many contributors volunteer their time unpaid. However, initiatives are emerging to financially support these contributors. Prominent OSS communities include the Apache Software Foundation and the Linux Foundation.

The above classification of SECOs reveals their various roles in the industry. Together, these ecosystems form a complex environment that is critical for understanding modern software development, particularly with regard to third-party libraries.

3.7 The npm Ecosystem

In this thesis, we delve into the realm of Component-based software ecosystems, with a specific focus on the npm ecosystem. This selection is due to a discernible gap in the literature and the intended design of a tool tailored for this ecosystem. Npm, an acronym for Node Package Manager, is a package manager for the Node.js platform. According to the official npm documentation, the current count of packages is 2,565,715.

Package managers, as defined by Hisman et al. [53], are *"programs that map relations between files and packages (which correspond to sets of files), and between packages (dependencies), facilitating users in maintaining their systems at the package level rather than dealing with individual files"*. In essence, package managers simplify the task of packaging files and code, promoting reusability within our codebases. This is the core functionality of npm, which aids developers in packaging reusable code to be used within the Node.js environment [2].

npm is recognized as the world's largest software registry. The npm ecosystem comprises three primary components:

- Website: Utilized for searching and examining packages.

- CLI (Command Line Interface): Employed by developers to manage packages, including retrieving and uploading them to the registry with commands such as **npm install** and **npm publish**.

- Registry: The database that contains all the JavaScript packages.

We can understand the roles within the npm ecosystem by referencing the model presented by Hou and Jansen, as shown in Figure 3.3. The primary users of npm are software engineers and organizations that utilize the packages available through npm's Command Line Interface (CLI). Npm itself acts as the package manager, overseeing a registry that stores packages and facilitates their retrieval and installation upon user request. Additionally, npm maintains a local database that aids in managing the metadata of installed packages. The packages in the npm registry are maintained and kept up-to-date by software engineers. The package can also have dependencies in the npm, other dependent packages which are called transitive dependencies, the npm automatically detects and installs all the packages.

They also explain the trust factors in the software package managers, the factors that contribute towards the trust are dependency hell, security vulnerability, and package prevention. In npm there is no such solution to dependency hell, when you install a package it will give you all the dependencies, npm is not strong enough with prevention as explained in [77], most of the high vulnerabilities and preventions are done when a user reports it and they can take down the package, also regarding the vulnerabilities, there can be packages with vulnerabilities and the only thing that npm is doing is giving you with warning using npm audit, and an ability to fix with npm audit fix, however, it does not detect all the vulnerabilities and it does not fix all of them.

Compared to other Software Ecosystems (SECOs), npm is distinct as it is open-source, allowing everyone to contribute and view the code. This contrasts with many other SECOs that are closed-source. Another unique aspect of npm is its centralization around a single repository, the npm registry. Governance in npm is community-driven, which sets it apart from others. Additionally, npm excels in dependency management, offering robust features in this area. Furthermore, the ease of versioning and publishing in npm is notable, making it more user-friendly compared to other SECOs.

3.8 Background on TrustSECO

TrustSECO is an innovative system designed to help users confidently choose and install software based on its reliability. Its main spotlight is on tracking and assessing different versions of software packages. The system gathers insights from a wide array of users, including regular software users, software-making companies, and even the individuals behind the creation of these software packages. They all come together to pool information that provides insight into the software's past performance, any known issues, and its general trustworthiness. This gathered data is housed in an online ledger that serves as the foundation of TrustSECO. using a specially created scoring mechanism, this ledger actively assists in determining how much a user can trust a specific software package. By ensuring users have all the information they require about software before deciding to use it, TrustSECO essentially aims to make the digital world safer and more dependable [33]. In the following subsection, we explain and dive into each part of the TrustSECO software.

3.8.1 Distributed Ledger

The Distributed Ledger was designed to decentralize the database, allowing the community to create and access it. The community ensures the system remains private and secure to prevent the entry of unchecked data, employing "spider jobs" that come with a variable fee.

This ledger lets both users and software providers input data. Based on the information provided, the ledger adjusts the score: factors such as the number of GitHub stars can boost the

score, while the presence of harmful bugs might reduce it.

3.8.2 Spider

The spider is in charge of collecting the information or trust facts via API calls or web crawling. It pulls data from several sources, including Stack Overflow, Libraries.io, GitHub, and CVE to collect the data.

3.8.3 Portal

The portal of the project is created in Vue.js[2]. The main purpose of the portal is to provide UI for the user, so they can spider and/or retrieve/store data on the distributed database.

3.8.4 Trust Score Calculation

Through the use of a *trust score*, TrustSECO has developed a system to evaluate the trustworthiness of software packages. This rating has several properties:

1. **Multi-dimensionality**: It assesses the software package as well as its particular version and the software engineer community that supports it.
2. **Transparency**: The score calculation is replicable given the same data, promoting community consensus.
3. **Combinatorial**: The evaluation of package combinations can be done by combining various scores.
4. **Numeric**: Designed to quickly determine whether a piece of software is reliable.

According to TrustSECO, software trust has many facets and is influenced by both the software and additional factors. They have a number of properties that they are calculating based on TrustSECO's scoring. Starting with **Packages + Versions**, because trust is variable, different versions of the same package might have different trust ratings. Bug resolution times, user feedback, known vulnerabilities, and confidence in the contributing software engineers are all factors that affect this. Secondly, the sole **Package Managers**, they are potentially unsafe. TrustSECO's assessment criteria encompass Usage frequency. Any malicious, outdated, or broken dependencies. Reputation and popularity. Compliance with security standards. Recent compromises of the package manager. Another part of the calculation involves the **Software Engineers**, who are critical to the collection of the Trust facts because they are providing information such as activity duration on platforms such as Github, star ratings, and negative experience. Last, it is the **Software Organizations**, they are considering the following factors: Organizational support and popularity.

The steps for Trust Score calculation are as follows:

1. A user submits a package name and version.
2. Relevant trust facts for that package are retrieved.
3. Each trust fact's data points are counted and stored in an array.

[2]https://vuejs.org/

4. Each data point, converted to an integer, is multiplied by its weight. This value is then divided by the number of data points for that trust fact to obtain an average.

5. The final score is adjusted to fit within a 0-100 range

The model of the TrustSECO can be seen in Figure 3.4

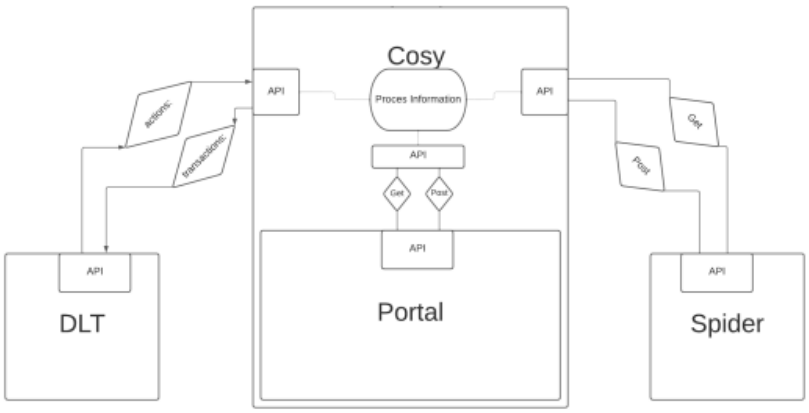

Figure 3.4: Architecture for Integration of TrustSECO with npm [9].

The implementation details and source code of TrustSECO are available for review on GitHub at this link.

3.9 Grey Literature Review

In the systematic literature review (SLR), we identified various tools for trust reinforcement mechanisms within the npm ecosystem. A notable gap was observed in the literature regarding the development of such tools. Given that our tool is intended to be command-line based, a decision influenced by the common interaction of developers with npm through CLI, we explored this direction further. The npm ecosystem primarily comprises three components: the website, CLI, and the registry. While the website facilitates library exploration and the registry serves as a storage medium, the CLI remains the most interactive element for software engineers. Hence, our focus is on developing a CLI wrapper tool over npm. However, existing literature lacks detailed insights into the development methodologies, libraries used, and best practices followed in the creation of such tools. Consequently, we turned to grey literature to bridge this knowledge gap.

For adherence to best practices in CLI tool development, we referred to sources such as Prasad et al. [63], Jeff D [19], official Heroku documentation [1], Gnu Standards for Command Line Interfaces [28], and Czapski [18].

Czapski emphasizes the development of command-line interfaces (CLIs) across various programming languages, highlighting key guidelines for optimal CLI design. These guidelines include ensuring command clarity, producing easily interpretable outputs, and enhancing command discoverability.

Jeff presents a structured approach with 12 critical factors for building effective CLIs that they introduced in Heroku. These encompass the necessity of comprehensive help features, a

preference for flags over arguments, inclusion of version control, attention to input and output streams, robust error handling, user-friendly interfaces, interactive prompts, the use of tabular data representations, optimizing for performance, fostering a community for contributions, and clarity in subcommand structures.

The official GNU documentation advocates for simplicity and user-friendliness in CLI design. It recommends the implementation of both short and long-named options, such as '-o' and '–options', to cater to user preferences. A fundamental requirement is the inclusion of '–version' and '–help' flags in all CLIs. Additionally, it stresses the importance of maintaining consistency in commands, flags, and arguments throughout the program.

Prasad's guide offers a thorough framework for creating effective command-line interface (CLI) tools. His methodology focuses on a 'human-first' design, prioritizing user experience in CLI development. He emphasizes the critical role of comprehensive documentation and readily accessible help functions to guide users. The guide also delves into the importance of designing intuitive and clear outputs, ensuring that users can easily understand and interact with the CLI.

Error handling is another key aspect highlighted by Prasad, underscoring the need for CLIs to manage and report errors effectively. He discusses the optimal use of arguments and flags to enhance user command control, as well as the significance of CLI interactivity for a more engaging user experience. Subcommands are addressed, with a focus on their organization and clarity.

Furthermore, Prasad underscores the importance of robustness and future-proofing in CLI tools. This includes considerations for scalability and adaptability in response to evolving user needs and technology advancements. Lastly, he provides insights into effective naming conventions, ensuring that command and option names are intuitive and self-explanatory, thereby enhancing overall usability.

Our exploration in the npm registry and various articles led us to identify the following libraries that aid in CLI tool development:

- **yargs:** 90,783,468 Weekly Downloads, 7 dependencies, last release: 7 months ago, Health Analysis: 91/100.
- **oclif:** 124,840 Weekly Downloads, 18 dependencies, last release: 1 month ago, Health Analysis: 86/100.
- **minimist:** Weekly Downloads: 56,176,805, Dependencies: 0, last release: 9 months ago, Health Analysis: 88/100.
- **meow:** 18,440,305 Weekly Downloads, Dependencies: 0, last release: 3 months ago, Health Analysis: 91/100.
- **inquirer:** Weekly Downloads: 28,800,689, Dependencies: 15, last release: 6 days ago, Health Analysis: 97/100.
- **vorpal:** 34,767 Weekly Downloads, Dependencies: 10, last release: unknown, Health Analysis: 53/100.
- **commander.js:** Weekly Downloads: 134,435,650, Dependencies: 0, last release: 2 days ago, Health Analysis: 93/100.

The Health Analysis is done with Snyk. Among these seven npm libraries, oclif is classified as the most comprehensive framework. However, its popularity is moderate, it has numerous

dependencies, and its health rating is below 90. In contrast, yargs, minimist, meow, and commander.js are more lightweight and provide essential features for CLI development. Inquirer is a utility for interactive prompts. Vorpal, however, is no longer supported and has a significantly low health rating.

In the evaluated npm libraries, commander.js stands out as a notable example, striking an optimal balance between being lightweight and feature-rich. It emerges as the most popular among its counterparts, characterized by its extensive maintenance. Remarkably, commander.js operates with zero dependencies and has achieved a health score of 93, further solidifying its position as a preferred choice in this domain.

3.10 Discussion

With the SLR and all the data that we gathered, we can answer our research questions. We have 25 papers to study to answer RQ1 and RQ2, and 15 papers for RQ3, the data extracted can be seen in Table 3.12

Research questions	Papers
RQ1	29
RQ2	29
RQ3	14

Table 3.12: Research questions with number of papers

RQ1: What kind of trust reinforcement mechanisms exist in package ecosystems?

Answer: Addressing this question is complex. Initially, we must define what constitutes trust in the software ecosystem. According to Hou et al.[9], it is the end-user's willingness to take risks, grounded in their belief that the system providers will be dependable. Based on this definition, trust in package ecosystems involves the end-user trusting that both the package provider and creator will be reliable. However, trust cannot always be assured, especially in less stringent open-source systems like npm. While npm strives to enforce security and dependability, vulnerabilities can still be introduced. As noted in subsection 3.5.1, there are numerous tools enhancing trustworthiness in these ecosystems. These include various approaches like dynamic and static analysis, build automation, sandboxing, and machine learning techniques to identify vulnerabilities. These tools focus on different aspects, such as transitive vulnerabilities, code injections, zero-day vulnerabilities, clone packages, malicious code, and permission systems, each with its specific use cases. The npm ecosystem performs checks for potential security issues, but some risks may still bypass these measures. Notably, most tools do not prevent the installation of a package but rather operate as post-installation tools, scanning the codebase or requiring manual package checks, as shown in Table 3.6. We identified a gap in our study: the need for a tool that safeguards users before they install a package, allowing them to assess its trustworthiness.

RQ2: How effective are trust reinforcement mechanisms in package ecosystems?

Answer: The evaluation of trust reinforcement mechanisms in package ecosystems, as reported in the literature, generally indicates positive outcomes. Tools analyzed in various studies demonstrated high precision in detecting issues and security vulnerabilities in libraries. However, there's a variance in the reported effectiveness. Some studies did not disclose effectiveness metrics, while those that did revealed differing perceptions of effectiveness. This variation is

often attributed to factors like the occurrence of false positives or negatives, performance measurements, and the number of errors identified.

In terms of capabilities, certain tools were effective in identifying basic errors, whereas others were advanced enough to detect zero-day vulnerabilities. The diversity in techniques used by these tools complicates the task of establishing a uniform standard for comparing their effectiveness. Additionally, the literature often points to limitations such as high rates of false positives or negatives, reliance on specific platforms like GitHub, and constraints unique to particular tools. These factors collectively contribute to the challenge of assessing the overall efficacy of trust reinforcement mechanisms in package ecosystems.

RQ3:How do software engineers perceive the balance between adding new features and ensuring security in their npm packages?

Answer: In Subsection 3.5.2 and Subsection 3.5.3, we examined the literature related to the utilization of third-party libraries and the security awareness of software engineers. Our analysis revealed that software engineers demonstrate an increased level of security consciousness when selecting third-party libraries, regardless of whether they are complex libraries or simple packages. Nevertheless, while selecting libraries or attempting to upgrade them, the emphasis is consistently placed on prioritizing the library's usefulness or the stability of the system rather than its security. Security vulnerabilities are not their first priority. When selecting a library, individuals tend to value indicators such as documentation quality, download numbers, and star counts. A significant proportion of software engineers are unaware of the vulnerabilities present in their libraries, and many of them neglect to update these libraries due to perceiving it as a burdensome task.

Based on our examination of Research Questions 1, 2, and 3, it is apparent that a deficiency exists within the current body of literature. To begin with, it is worth noting that there is currently a lack of available tools that offer proactive warnings or safeguards prior to the installation process. Furthermore, it is evident that software engineers acknowledge the need of security; nevertheless, it is not frequently prioritized within their top three factors when making package selections. It is believed that the implementation of this tool will serve as a preventive measure against the installation of potentially harmful third-party libraries. This is due to the observation that software engineers generally lack the initiative to thoroughly assess security vulnerabilities when selecting packages, as well as the limited effort put into updating existing packages. By "preventing" software engineers from initiating installations altogether, it is anticipated that a significant portion of the challenges currently encountered by software engineers can be mitigated.

Chapter 4

Design Science

In this chapter, we describe the creation of our artifact: **TrustSECO.js**. This CLI-based tool is developed to address gaps highlighted in our earlier literature review. Its primary function is to offer both pre-installation and post-installation features, with a particular focus on the former. This ensures users avoid adding unreliable or harmful software to their projects. A pre-installation tool or feature provides a safeguard for software engineers, ensuring safety and compatibility before installing any package or software in a project or on a machine. On the other hand, a post-installation tool or feature is used for conducting thorough checks and assessments after the software or package has been installed.

4.1 Conceptualization of TrustSECO.js

TrustSECO.js serves two main purposes: first, it protects developers from questionable and vulnerable software. Second, it allows for an analysis after software installation. If developers have already used several third-party libraries, our tool offers two methods to check the trustworthiness of these packages. The main method involves scanning the entire codebase, specifically the *package.json* file, to identify dependencies. These are then checked against the distributed ledger of TrustSECO. This method has two paths: one that looks only at primary dependencies and another that examines deeper transitive dependencies. Based on our literature study, vulnerabilities in one dependency can affect its connected elements. Thus, understanding these deeper connections helps developers decide whether they should switch libraries or update versions.

4.1.1 Core Functionalities and Post-installation Analysis

TrustSECO.js serves a dual purpose: it protects developers from untrustworthy software and allows post-installation scrutiny. For projects with pre-existing third-party libraries, the tool offers two distinct methods for trustworthiness evaluation: scanning the codebase to list dependencies and offering both primary and deeper transitive dependency analyses.

4.1.2 Additional Features: Reporting and Policy-Based Access

To cater to various developer and organizational needs, **TrustSECO.js** can produce CSV reports of dependencies. The exploration of transitive dependencies without installing the specific library is also facilitated. Furthermore, drawing from our literature insights, a policy-based access feature is planned, allowing entities to set specific installation policies.

In essence, **TrustSECO.js** seeks to furnish developers with a holistic view of their software libraries. The subsequent sections will delve deeper into the design details of **TrustSECO.js**.

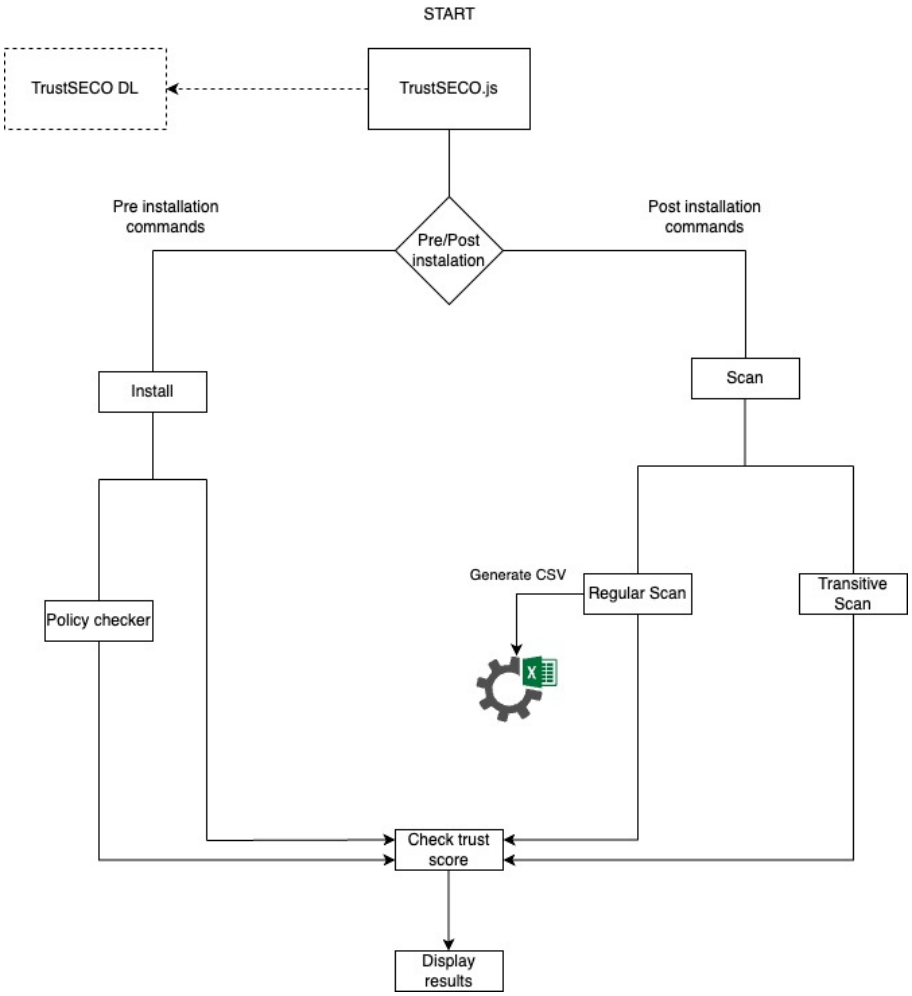

Figure 4.1: Meta model of the initial design.

Meta-model can be seen in Figure 4.1

4.2 Design Rationale

During our literature review, we noticed that many tools focused on what they achieved, how they operated, and the results they delivered. Still, there was a lack of detail on how they were developed, the challenges faced, or the specific tools used. This observation led us to explore grey literature to better understand how to develop our NPM CLI integration.

We primarily turned to content written by experienced developers because of their hands-on tool-building insights. Medium articles[1] and the official node.js documentation were particularly informative.

According to the Node.js documentation, it is possible to create CLI tools using built-in modules by processing command-line arguments provided to the Node.js script *using process.argv*, which allows access to user inputs. We can also use *child_process*. This method

[1] `https://medium.com`

lets us run external programs within our Node.js application [25]. However, this manual approach can be quite time-consuming and requires a lot of manual work, especially when handling different command-line inputs, where we have to parse and validate the command-line arguments.

Instead of spending too much time on detailed low-level tasks, we looked for existing libraries that could help speed up our work. Third-party library helps us abstract the repetitive tasks of processing the command-line arguments, it makes the process more efficient and less error-prone.

Our search in the npm registry brought us to several promising CLI libraries explained in SLR, namely the Grey literature section:

- yargs[2]
- oclif[3]
- minimist[4]
- meow[5]
- inquirer[6]
- vorpal[7]
- commander.js[8]

Many of these libraries depended on others. Following the advice from Prasad et al. [63], we aimed for libraries with few dependencies to reduce potential risks. We wanted a simple tool but one that also had all the functions we needed, regarding this part, most of the libraries that we looked at the base and necessary features required to build a CLI tool. From the libraries listed, commander, minimist, and meow had no dependencies. When we checked their trust scores, we found that inquirer, yargs, meow, and commander all had substantially high scores, the scores were obtained from the Snyk database. Given that TrustSECO is known to occasionally produce false positives, we aimed to ensure the accuracy of our library selection by cross-referencing with this more reliable source. Ultimately, we chose commander.js because it is widely used, has a high trust score, an active user community, regular updates, and has the highest download count.

TrustSECO.js tool was built using TypeScript[9], chosen over JavaScript for better type-checking. Since the tool is for npm, using its native language was sensible.

4.3 Detailed Design

This section provides an in-depth look at the design of our Command Line Interface (CLI) tool, breaking down the architecture of commands, the method of output delivery, and the design of

[2] `https://www.npmjs.com/package/yargs`
[3] `https://www.npmjs.com/package/oclif`
[4] `https://www.npmjs.com/package/minimist`
[5] `https://www.npmjs.com/package/meow`
[6] `https://www.npmjs.com/package/inquirer`
[7] `https://www.npmjs.com/package/vorpal`
[8] `https://www.npmjs.com/package/commander`
[9] `https://www.typescriptlang.org/`

the user interface. It offers a thorough exploration of the thought process behind each feature, the libraries chosen for implementation, and the reasons behind these choices.

Our CLI tool is developed with a focus on human-first interaction, meaning its primary function is to engage with users to carry out various tasks. To craft a user-friendly experience for developers, we followed guidelines from several well-known sources, including Prasad et al. [63], Jeff D [19], the official Heroku documentation [1], Gnu Standards for Command Line Interfaces [28], and Czapski [18].

Commander.js played a crucial role in the tool's development, providing high-level functionalities that made the setup of our CLI tool straightforward. The library's built-in commands created a structured way of defining the commands that trigger specific logic. We started with an initial version of our tool, supplemented with a detailed description of its purpose, and then introduced four main commands—info, install, scan, and view-tree. Each of these commands will be introduced in more detail in the following sections. The commands are associated with specific keywords and a concise explanation, as represented in Figure 4.2, helping users utilize the tool effectively without constantly referring to external documentation.

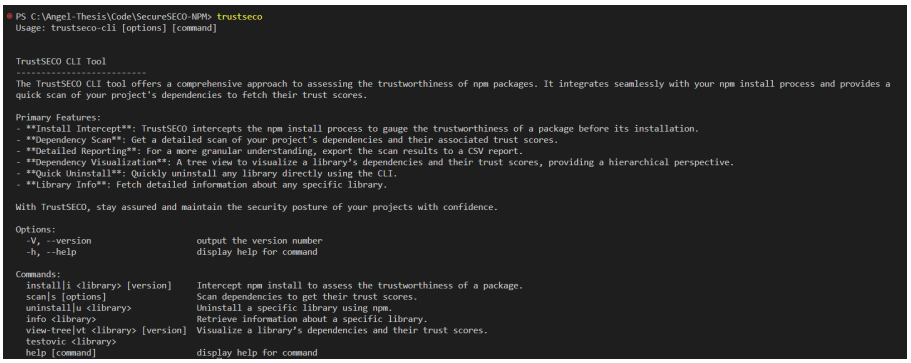

Figure 4.2: Comprehensive help information

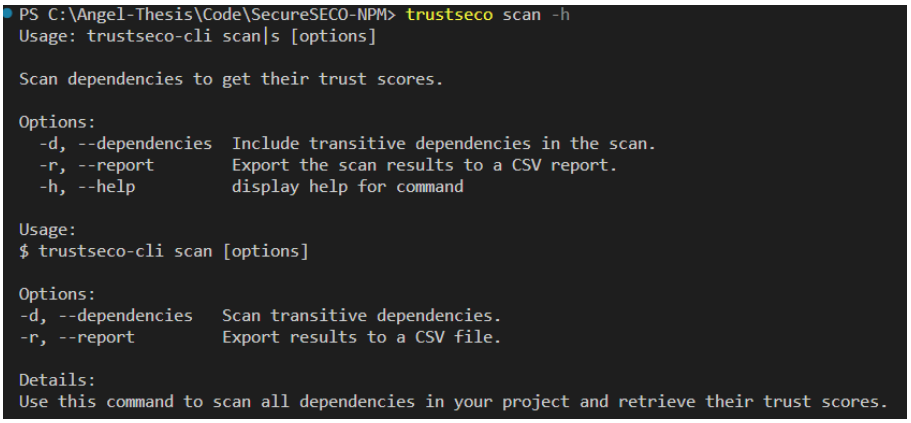

Figure 4.3: Comprehensive help information of Scan command

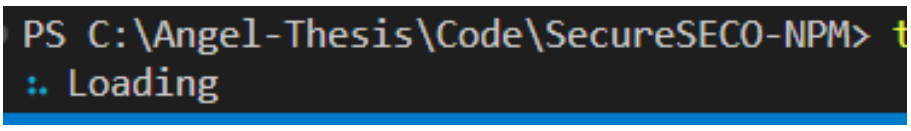

Figure 4.4: Loading Process

One of the key principles in our development process was to extend commands through the use of flags, avoiding the introduction of additional arguments or separate commands. This approach resulted in enhancing the scan command with flags such as -dependencies and -report, allowing users to include transitive dependencies in the scan and export the results to a CSV report, respectively. For user convenience, shorter versions of the flags (-d and -r) were also made available.

Once the basic structure of the CLI tool was in place, we focused on creating detailed help sections for each command. Thanks to Commander.js's inherent help flag, users can easily access general information and a list of available commands, enhancing their experience as shown in Figure 4.3.

4.3.1 Install Command

The install command is a fundamental element of the CLI tool, acting as the main mechanism for developers to incorporate third-party libraries. It mirrors npm's functionality, allowing developers to specify a library name and, if needed, its version. Detailed information is available through the -h or –help flag, as illustrated in Figure 4.6.

A significant feature of the install command is the policy checker. It allows for the incorporation of custom policies by adding a policy.{env}.json file to the project's root directory. With a Permissive Approach, the policy checker functions smoothly when there are no restrictions, operating similarly to the standard npm. The flexibility in the allowed and blocked properties guides the accessibility to libraries, with warnings presented for any policy breach. The mandatory presence of a policy json file in the root directory and the non-restrictive nature of this feature are essential, ensuring developers have the final say, as depicted in Figure 4.8 and Figure 4.7.

After the policy verification step, the trust score and package details are fetched from the ledger and displayed in a user-friendly ASCII table format. If the trust score is below a set threshold, a conspicuous yellow warning is issued, seeking user confirmation to proceed. This process, detailed in Figure 4.5, highlights the intricate nature of the install command in our tool, significantly contributing to secure and efficient software development. For each array-like structured data retrieved from the ledger, an effort is made to display it in a separate table. This approach aims to enhance the intuitive understanding and user experience for the developer.

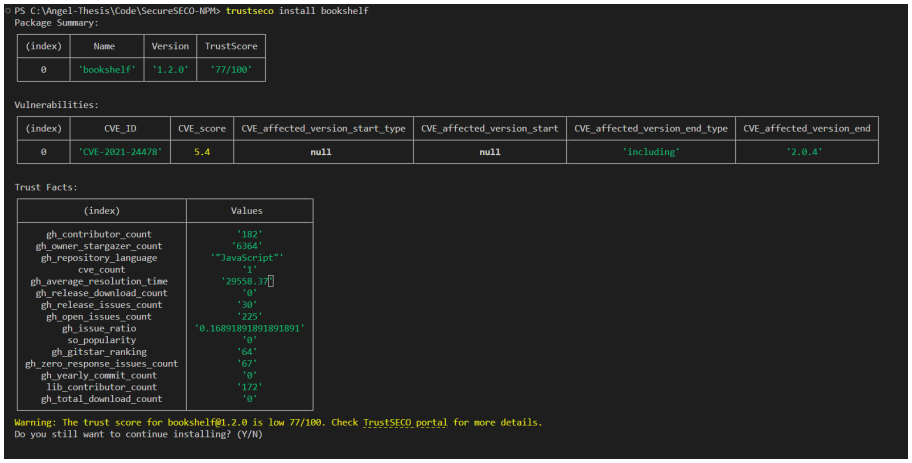

Figure 4.5: Install Intercept Process

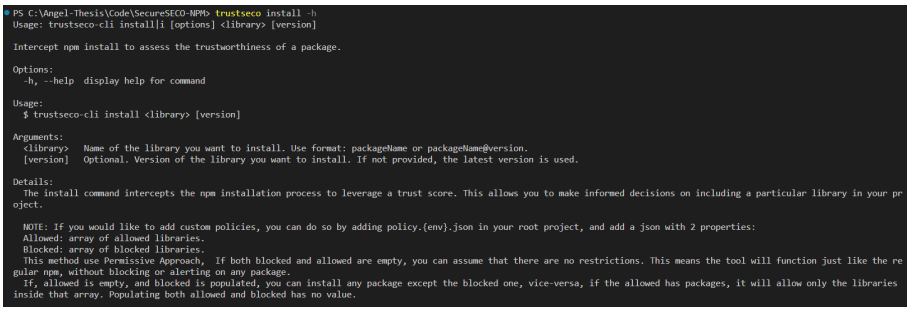

Figure 4.6: Install command help

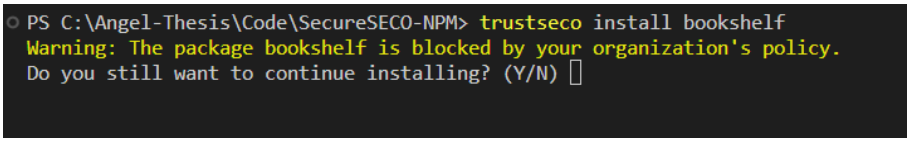

Figure 4.7: Blocked Library Alert

4.3.2 Scan Command

The second command incorporated in our tool is the scan command. During the development phase and the subsequent literature review, our tool was envisioned with the primary objective of establishing a pre-installation mechanism to prevent users from incorporating malicious packages. Although our tool was originally designed for pre-installation, it possessed the essential features enabling it to function as a post-installation tool as well. This capability is particularly beneficial for projects that are already built, allowing them to scan their existing installed packages. Similar to the install command, developers can employ a help flag to review the description and functionality of the command, as illustrated in Figure 4.3.

The core function of this command is to scrutinize the *packages.json* file, extract all the dependencies, perform a scan over the distributed ledger of TrustSECO, and display the results in the terminal. Any trust score falling below the predetermined threshold will be highlighted in yellow to attract the developer's attention. The scan command is also equipped with two additional features: the ability to scan dependencies and generate reports. We have enhanced the tool to scan not only the primary dependency but also include transitive dependencies. By executing the flag -d or –dependencies, we leveraged the npm ls command to retrieve all transitive dependencies and present a tree-like visualization of the trust scores, as depicted in Figure 4.10. In the transitive dependency scan, we relied on npm to list dependencies. While the **npm ls** command provides an extensive tree of all dependencies, our CLI needed to traverse each package to obtain its trust score. Given the potentially vast number of packages, this process involves significant recursion. To optimize data fetching, we employed memoization and caching portions of the tree. If a trust score for a package was previously calculated, we retrieved it from the cached data.

The final feature of the scan command is report generation. For every scan or transitive scan, the flag -r or –report can be added to generate a CSV file named trust_scores_report.csv in the project's main root. This feature facilitates better reporting for developers and aids in the future improvement of their untrusted dependencies, with an example of the reporting illustrated in Figure 4.11.

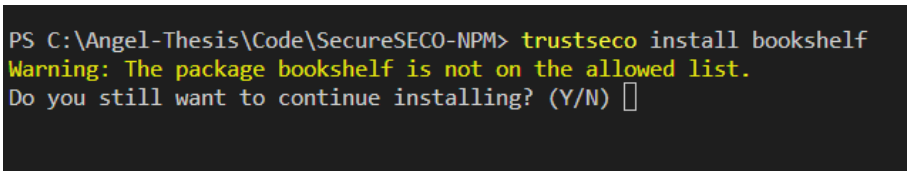

Figure 4.8: Allowed Libraries

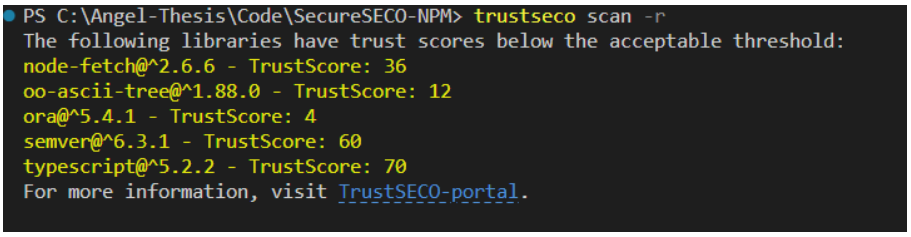

Figure 4.9: Dependency Scan with Report Export

4.3.3 View-Tree Command

The view-tree command serves as an augmentation to our pre-installation strategy. It is particularly useful when there is a need to inspect transitive dependencies without downloading the app or navigating to the portal. By employing the view-tree command, users can visualize the transitive dependencies along with the trust scores, as demonstrated in Figure 4.12.

4.3.4 Info Command

Lastly, the info command serves as an alternative approach to the install option. It accommodates developers who may prefer to review detailed information and the trust score of a package prior to installation. This command, as exemplified in Figure 4.13, enables such functionality, providing users with insights into package details without necessitating installation.

Figure 4.10: Transitive Dependency Scan

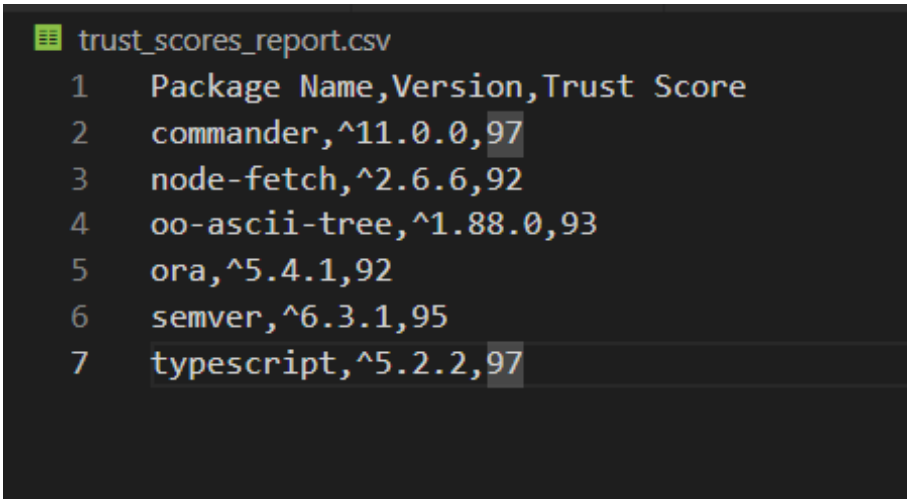

Figure 4.11: CSV report

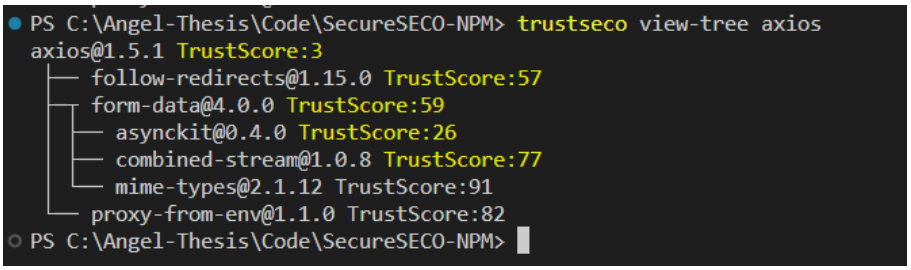

Figure 4.12: Dependency Visualization Tree

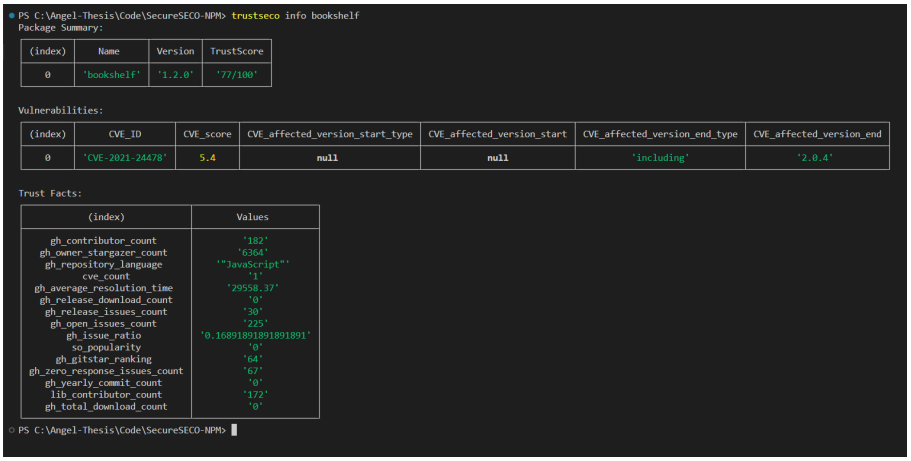

Figure 4.13: Library Information Retrieval

4.4 Preliminary Testing

For the examination of the TrustSECO.js tool, two distinct testing methodologies were adopted. The first method involved the utilization of the Jest[10] testing framework, a widely acknowledged and industry-recommended tool, for conducting automatic unit tests. Unit testing is a foundational approach in software engineering, aimed at validating the smallest components of an application, thereby enabling developers to identify and rectify issues at an early stage.

The second method was manual testing, which entailed executing each command on a variety of third-party packages. The combination of both automatic and manual testing strategies proved invaluable in identifying and resolving logical and critical bugs within the tool, thereby enhancing its reliability and efficiency. Nevertheless, it is relevant to note that the tool's performance is contingent on third-party API calls, such as requests to the TrustSECO Distributed Ledger and npm APIs. Addressing potential bottlenecks related to these dependencies extends beyond the purview of this study, as it necessitates optimizations at the backend of TrustSECO.

The importance of speed in the operability of any CLI tool is underscored by Jeff D. [19], emphasizing that while promptness is crucial, certain allowances must be made for tools performing significant tasks. The TrustSECO.js tool aligns with this observation, as it undertakes substantial tasks, notably the API fetching of each transitive dependency. Consequently, variations in performance were observed, with some components exhibiting expedited responses, while others necessitated extended durations.

A summary of the testing results for each command and flag is as follows:

- `install`: 2790ms with verions. 4414ms without version
- `info`: 1500ms
- `scan`: 4000ms
- `scan -d`: 137s
- `scan -r`: 4000ms
- `view-tree`: 19s

The test was conducted using the Bookshelf library to examine the installation process, utilizing the *MeasureCommand*[11] PowerShell command to gauge the execution speed. The selection of Bookshelf as a testing library was based on its profile of having few dependencies, numerous unresolved issues, known vulnerabilities, and a lack of releases in the past three years. This made it an ideal candidate for evaluating the trust score and assessing the performance of the library.

The tests were done on a Windows 11 laptop with the following specs:

- Intel Core i7-1185G7 @ 3.00GHz
- 32GB RAM

[10] `https://jestjs.io/`

[11] `https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/measure-command?view=powershell-7.3`

The results show that the install function is notably slower, nearly twice as slow as a standard npm install, when the version is not specified. This slowdown is due to the need for two HTTP calls, one for fetching data and another for obtaining the latest version from the npm registry. On the other hand, a significant speed improvement is observed when the version is specified by the user. The time taken is close to that of npm install, with a small additional delay of 300 milliseconds in our setup, which is deemed reasonable.

The performance of the 'info' command is also commendable, executing in a relatively quick 1500 milliseconds. This speed is appreciated, especially given the necessity to initialize the JavaScript compiler, a step known to take a significant amount of time. However, the evaluation of other commands such as 'scan' and 'transitive dependencies' faced a challenge due to issues in the backend system that generate trust scores, preventing their interaction with our packages. To work around this, a mock delay of 300 milliseconds was introduced to simulate the fetching of trust scores and trust facts. The choice of a 300-millisecond delay was based on the observed time difference between npm install and our install process, assumed to be the data retrieval time. It is important to note that this setup is experimental, and the findings should be viewed cautiously. A more thorough assessment is planned upon the stabilization of the backend system in a production setting, which will provide a clearer understanding of the process.

The 'scan' command was tested on a collection of 12 different packages namely axios, bookshelf, chalk, commander, express, lodash, moment, node-fetch, oo-ascii-tree, ora, semver, and typescript. The scanning process took a considerable duration of approximately 137 seconds, largely due to the numerous transitive dependencies associated with most packages. However, Bookshelf, with fewer dependencies, displayed a significantly faster scan time of 19 seconds, indicating a relationship between the number of dependencies and the time required for the scanning process.

4.4.1 Dependencies used and additional UX

In the project, we utilized several dependencies to facilitate the development process. We aimed to minimize the number of dependencies for our tool, prioritizing those with the fewest transitive dependencies. Additionally, this subsection will discuss various UI/UX improvements implemented in the tool.

Three additional dependencies were incorporated:

- **oo-ascii-tree**[12]: Enables tree-structured visualization for dependency scans.
- **semver**[13]: Aids in clean versioning of dependencies for ledger communication.
- **ora**[14]: Enhances UI via an interactive loading spinner.

Each command provides detailed feedback, as previously discussed. A loading indicator, illustrated in Figure 4.4, informs the user of ongoing processes. For lengthier processes such as scans, additional information regarding the expected duration is provided, improving user engagement. Users also have the option to use aliases for the lengthy *trustseco*, they can utilize the shorter version *ts*, and also for each command they can use their respective aliases which can be seen in Figure 4.2

[12] `https://www.npmjs.com/package/oo-ascii-tree`
[13] `https://www.npmjs.com/package/semver`
[14] `https://www.npmjs.com/package/ora`

4.5 Design Limitations

The major design constraint, as discussed earlier, is the developmental stage of the distributed ledger, which does not yet provide users with a complete experience akin to a production environment. This limits our ability to optimize our tool fully. Concerning the tool's features, a notable bottleneck is observed in the **view-tree** function while checking transitive dependencies, as it requires frequent calls to the npm API to retrieve dependencies. This area might benefit from future optimizations or feature enhancements to reduce these API calls. However, it is crucial to mention that we have made strides to improve the performance by implementing a top-down memorization approach for the view-tree to limit recalculating the same tree pieces for a particular package.

Chapter 5

Interview study

In this interview study, the primary objective is to conduct a series of interviews with software engineers actively using or have been using Javascript and Node.js in their routine software development activities. The intention behind these interviews is to gain valuable insights into the software engineers' attitudes and practices regarding the use of third-party libraries. This involves obtaining an understanding of the frequency with which libraries are used and the factors software engineers deem important when selecting such packages.

A central goal of this part is to assess the practicality and utility of the introduced CLI tool from the participants' viewpoint.

A key aspect of our investigation will be gathering general feedback and evaluating the usability and user experience of TrustSECO.js. The interview study will compare the tool with others that participants might have used previously, aiming to understand the perceived effectiveness of our tool. A crucial component of our research will focus on the performance and reliability of TrustSECO.js, identifying any inconsistencies or issues encountered by the users.

Furthermore, the study seeks software engineers' suggestions and insights for future developments and improvements, aiming to refine and enhance the tool based on real-world user feedback and experiences.

5.1 Methodology

The methodology for conducting our interviews was based on the principles outlined by Taherdoost [66], which provides a comprehensive guide to organizing effective research interviews. According to Taherdoost, interviews can be conducted either remotely or face-to-face, adopting one of three formats: structured, semi-structured, or unstructured.

A pivotal aspect of the interview process is designing the questions. In qualitative studies, creating open-ended questions is vital to allow participants the freedom to express their viewpoints thoroughly. Another vital component is the selection of participants. It is imperative to choose individuals whose insights are appropriate to the study, ensuring the gathered information is relevant. Taherdoost recommends a participant range of 10 to 50 individuals.

The interviewer's skills significantly influence the success of the interview. Essential abilities include active listening, effective conversation management, patience, timing, flexibility in phrasing, avoiding biases, and maintaining confidence throughout the interview process.

Additionally, managing the interview session is crucial. Consideration of the interview's duration, as well as the impact of the site and location, is essential. The interviewer must be well-prepared and present information clearly to the interviewee, ensuring an understanding of

the interview's purpose and structure. Consideration should also be given to how the interview will be recorded.

Maintaining focus, revisiting specific questions, posing a final question, expressing appreciation to participants, and collecting feedback are all important elements of conducting an interview. To enhance the quality of the interview, it is advisable to avoid common mistakes such as disregarding participants' emotions, failing to seek additional information, causing interruptions, and relying exclusively on close-ended questions.

5.1.1 Interview participants

A total of 20 developers were interviewed for the interview study, a number considered sufficient to gather important information for our study.

The participant demographic was intentionally focused on industry professionals possessing a minimum of two years of experience in the field. This criterion was established to avoid the inclusion of students lacking professional experience, as their participation could potentially introduce inaccuracies in the findings due to their limited exposure to the professional industry.

Most participants were found through my own connections, built over three years of working in this field. These relationships, formed through direct contacts and recommendations, helped in identifying willing participants for the study.

5.1.2 Interview Design

During the design phase of our interview process, a concentrated effort was made to address our fifth research question (RQ5). The formulation of the interview questions was meticulously aligned with the research question, ensuring relevance and coherence. We adopted a semi-structured approach, a widely recognized methodology in scientific research, which allows for a combination of predetermined questions and the flexibility for the interviewer to seek clarification through additional inquiries.

The decision was made to conduct interviews through a hybrid model, incorporating both in-person and remote interactions. In-person interviews were favored due to the direct contact they facilitate with the interviewees; however, to accommodate developers unable to meet in person, remote interviews were also integrated.

The interview questions were systematically divided into nine sections. The initial section, Introduction and Context, aimed to gather insights into the developer's behavior towards third-party libraries. The subsequent section was designed to obtain general feedback regarding our tool. The third and fourth sections were concentrated on exploring the usability, functionality, and features of our CLI. Following this, we sought to understand the performance and reliability of the tool and glean insights into how it compares with existing tools used by the developers.

Additionally, developers were prompted to share ideas for future developments and improvements. Closing inquiries were made to assess the likelihood of the developers utilizing our tool in the future and recommending it to their peers. The comprehensive list of questions is detailed in Appendix A.

Through this methodical approach, we anticipate covering all facets of the usefulness of our implementation of TrustSECO.js, thereby garnering valuable results.

5.1.3 Interview Process

The initial design had each interview lasting approximately 60 minutes. While this duration was suitable for some developers, others found it too lengthy. After conducting four interviews, we opted to record a video in which we explained and demonstrated the tool. This video was sent to participants prior to their interviews, effectively reducing the meeting time to 30 minutes. Throughout each session, data was diligently collected either through notes or recordings, with interviewee responses documented in OneNote. The transcripts of OneNote can be accessed in Appendix A. The recordings were done using Microsoft Teams. At the end of the interview, they were also asked if they would prefer a summarized version of the document, which included the questions, sent to their email.

5.1.4 Ethical & Privacy Considerations

In accordance with university guidelines pertaining to ethics and privacy, every participant interviewed for this study provided their consent through a written consent form, referenced in Appendix B. Measures were implemented to maintain the confidentiality and anonymity of the participants, ensuring that any sensitive data acquired would not be disclosed in our study, with the exception of the participants' positions and years of experience in development.

Sensitive information such as names and surnames were solely utilized by the thesis author for the purpose of organizing and tracking the interviews conducted. Participants were afforded the right to pose any questions, withdraw from the interview at any time, and request the deletion of their data at any time. Furthermore, this study has been categorized as low-risk, with the university determining that no additional ethical review or privacy assessment was required. The Ethics and Privacy Quick Scan results can be accessed in Appendix B.

5.2 Results

In this section we will provide the results obtained by our interview, the section is subdivided into multiple subsections, each representing a subpart of our questions. We will both give explanations and provide visualization for some of the data that we have gathered.

5.2.1 Introduction and Context

As discussed in the previous subsections, our goal was to interview software engineers with a significant amount of experience. The respondents had experience ranging from 2 to 29 years. Notably, a substantial portion of the developers, specifically 60% of the respondents, had over 6 years of experience. These findings are illustrated in Figure 5.1. This high percentage of seasoned professionals suggests that the insights derived from our study are particularly valuable, given that the feedback is from developers who have spent a considerable amount of time in the industry.

The majority of our respondents, comprising 70% (14 out of 20 developers), identified as Full-Stack developers. This means they have experience working on both the UI and server sides of systems, providing them with a comprehensive understanding when discussing npm packages. It is worth noting that 15% (3 out of 20 developers) of the respondents were Cloud Architects, further diversifying the expertise and perspectives gathered in our study. The results are illustrated in Figure 5.2.

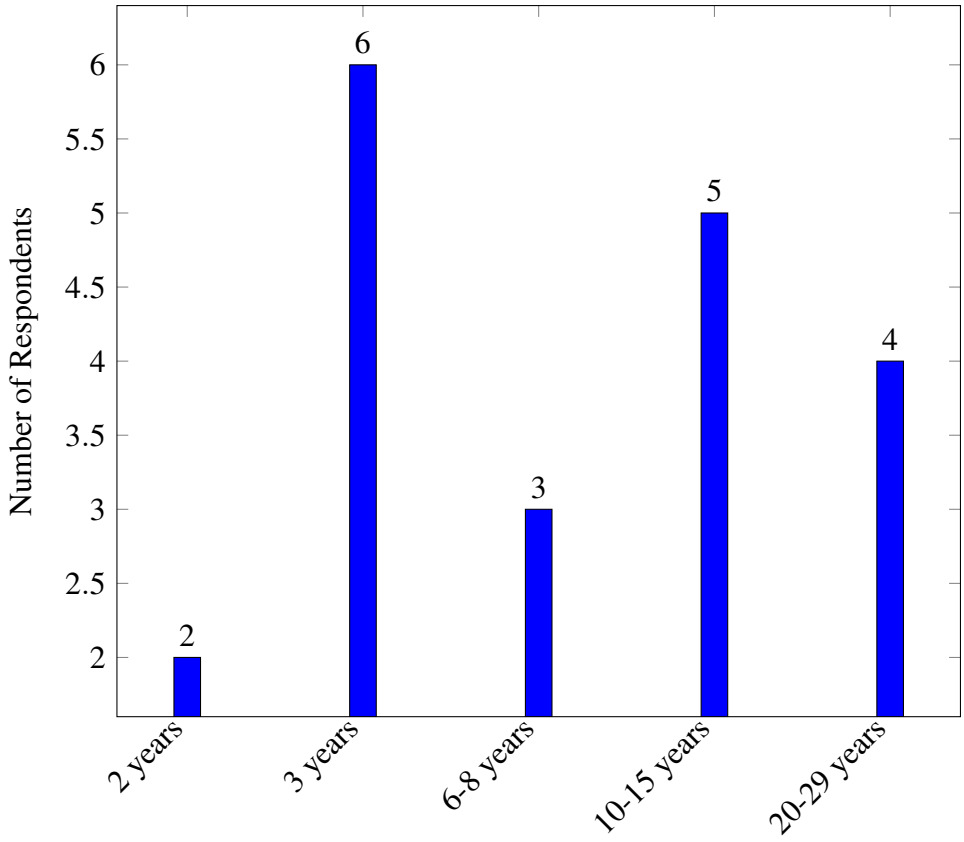

Figure 5.1: Distribution of Years of Experience among Respondents.

When developers were asked about the frequency with which they use third-party libraries in their projects, rating on a scale from 1 to 5 (where 1 indicates never using them and 5 indicates daily use), the results were quite revealing. Notably, none of the developers refrained from relying on third-party libraries. In fact, a significant majority, 60% (12 out of 20 developers), indicated a frequency level of 4, suggesting that they frequently integrate third-party libraries into their work. Furthermore, a whopping 80% (16 out of 20 developers) reported a score greater than or equal to 4. This underscores the ubiquity of third-party library usage in daily development tasks. These findings resonate with the literature review, further emphasizing that developers heavily rely on third-party libraries. Such a trend highlights the need for a secure open-source ecosystem, reinforcing the relevance and necessity of our tool, TrustSECO.js. The distribution of responses is depicted in Figure 5.3.

When the developers were asked about their considerations in choosing a third-party library, their responses were as follows: 60% (12 out of 20 developers) considered 'Popularity' as a major factor; 'Maintenance Activity' was a concern for 55% (11 out of 20 developers); 'Functionality' was 30%(6 out of 20 developers) 'Known Vulnerabilities' influenced 25% (5 out of 20 developers) of the respondents; 'Community Support' was considered by 20% (4 out of 20 developers); 'Licensing' was specifically mentioned by two developers, accounting for 10% (2 out of 20 developers); 'Source Trustworthiness' was significant for 20% (4 out of 20 developers); 'Size and Efficiency' was pointed out by three developers, which translates to 15% (3 out of 20 developers); and 'Peer Experiences' affected the decisions of 10% (2 out of 20 developers). Our tool encompasses most of these factors. All the data can be seen in Figure 5.4.

When asked about their practices in assessing the security of third-party libraries, a majority

of the respondents admitted they do not conduct independent checks and typically rely on feedback from the npm install process. Some base their decisions on feedback from open-source communities, while a small amount utilize tools such as Snyk or other prebuilt CI tools. Interestingly, developers from smaller companies often reported a lack of formal policies regarding third-party library usage; these companies tend to trust their developers' judgment in selecting appropriate libraries. In contrast, those employed by enterprise-level organizations or within the financial sector indicated stringent regulations and policies. Such developers have access to a pre-approved list of libraries, and any deviation requires following a specific protocol for approval.

Software engineers' responses on question *Does your company or team have a policy regarding the use of third-party libraries? If yes, how does it influence your choices? And how is the company enforcing those policies?* :

- ID-9 *Yes, we have, and of course, it influences what kind of library do I pick. We need to discuss it before we use it. We have to get an agreement between team members*
- ID-13 *Our company doesn't have a strict policy on third-party libraries. We typically discuss within our team and decide collectively. The company and clients trust us to select the best available solutions.*
- ID- 16 *At Company, we have a stringent policy against using third-party libraries. If there's an exception, a rigorous review process is initiated through our support team. Developers are also required to have certain certifications before they can install third-party libraries.*
- ID-19 *Our company doesn't have a strict policy on third-party libraries. Mostly, they trust developers to make informed choices.*
- ID-7 *Our company does have a policy concerning the use of third-party libraries. The primary emphasis of our policy isn't necessarily on security but rather on preventing application bloat and ensuring optimal performance. In essence, we're advised not to mindlessly add a multitude of packages, especially those that aren't genuinely beneficial or necessary for our projects. While we've had occasional security concerns, the guideline has been to exercise caution, especially with lesser-known or inactive packages, and apply common sense. However, during my time at a financial company, there was a more stringent approach. Given the nature of their operations, security was of great importance. They provided an approved list of libraries and any deviation from that required explicit approval from security personnel. It was a more structured and strict enforcement of third-party library use policies. Starting a new project, the financial institution has a list of approved packages. If someone wants to use a different package, they can't just decide on their own. They need to discuss it with the team. If the team agrees, they still need permission from higher-ups. It's a detailed process to make sure we only use the best and safest packages.*

The demographic characteristics of the interview participants are presented in Table 5.1. Regarding company size, responses varied, with some participants providing approximate figures. Notably, a number of respondents indicated the size of their client organizations instead of their direct employers, reflecting the consultancy nature of their companies.

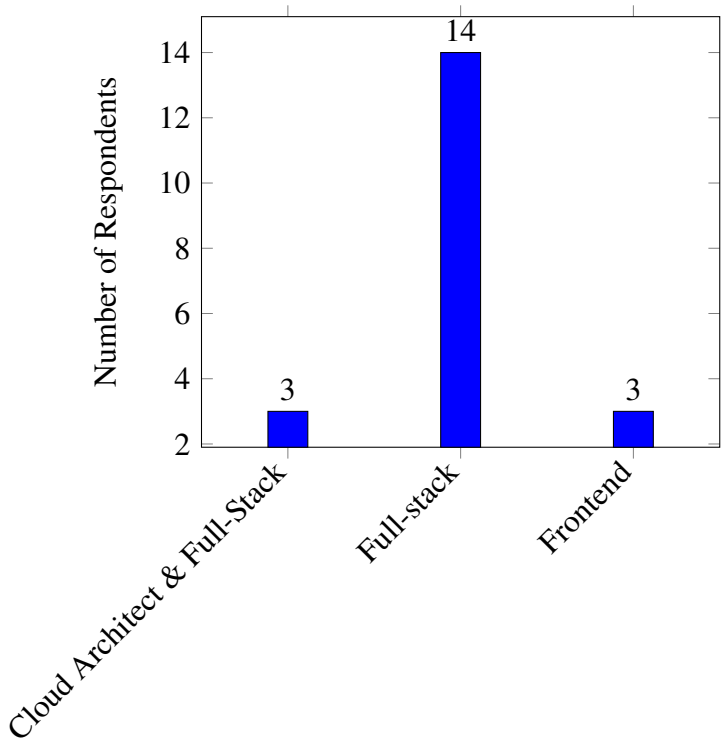

Figure 5.2: Distribution of Developer Roles among Respondents.

5.2.2 General Feedback

After using the CLI, developers consistently expressed positive feedback. They saw its potential and highlighted various ways it could benefit them and their colleagues. One of the main advantages they identified was the tool's ability to streamline research and enhance security, preventing potentially risky decisions. Many appreciated the CLI's user-friendly nature, especially because it resembled familiar npm commands. Several even contemplated integrating it into their existing projects.

However, there were concerns. Some developers pointed out the need for comprehensive and accurate information in the backend system, emphasizing that a tool lacking key data might be less effective. There were also questions about the trustworthiness and metrics behind the "TrustSECO". Even though the primary focus of this study was the integration of the npm CLI tool, the developers still felt the need to voice their thoughts on these interconnected issues.

Software engineer's responses for question: *What are your initial thoughts after using the CLI tool?*

- ID-2: *"I think ease of use is one of the most important aspects of this tool. It has well-defined general parameters and intuitive command names, such as 'I' for installation. When writing a CLI, ensuring it's user-friendly and provides helpful guidance is crucial. I appreciate the tree view feature, which offers extensive information about where each element fits within the overall stack of dependencies. "*

- ID-3: *"I find it very good and interesting. Similar to the Apple Store, which has stringent security measures where apps are thoroughly scanned and examined before being made public, I think npm is somewhat less strict. I haven't given it much thought before because I usually rely on the popularity and maintenance activity of a package, trusting it based on a gut feeling. Based on the SLR, this is almost the first time a CLI tool has been*

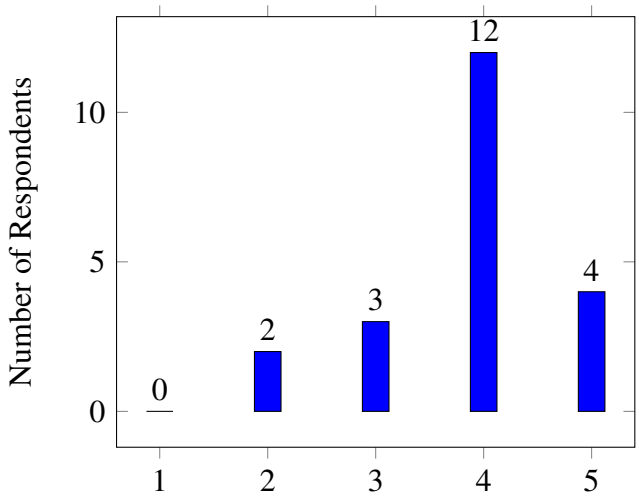

Figure 5.3: Frequency of Using Third-party Libraries on a scale 1-5.

developed specifically for npm to enhance trust in packages. Even with NuGet, the C# .NET alternative, I'm not aware of any system that scans for trust scores. In contrast, even though the Microsoft Store states that downloading something like "winget install Google Chrome" is the user's responsibility, I believe this new CLI tool is brilliant. Often, the discussion around these packages is based on hearsay, so having a tool to verify trustworthiness is extremely valuable."

- ID-4: *"It looks good, but its effectiveness largely depends on the database (ledger) behind it. If the database is well-structured and reliable, then the tool will be truly usable."*
- ID-5 *"After understanding its purpose, I realized how handy this tool could be for projects. It not only provides insights about the trust score but also about dependencies. Often, it's not clear what libraries a package relies on. This tool makes that information very clear and transparent."*
- ID-6 *"I think the CLI tool you showed is quite interesting. The feature to allow and disallow specific packages stands out to me. It's essential for users to see safety, security, and trustworthiness scores for packages, as not everyone might consider these factors, though they should. Especially for organizations that prioritize security, this tool could be highly beneficial. However, I would like to understand better how these ratings are determined and who decides the scores. Knowing the scoring system's details will help better assessing its reliability."*
- ID-20 *"I think it's fine. An extra warning before I do something stupid. "*

When we asked developers about their favorite features, 'transitive dependencies' was the top choice. Out of 20 developers, 15 pointed to this feature. Among them, 11 chose it on its own, and four paired it with other features. Next in line was the 'regular scan' feature, picked by three developers, but always with other features. The 'info' feature was noted by two developers: once by itself and once with 'regular scan'. Both 'reporting' and 'info' were picked twice. 'Reporting' was combined with 'transitive scan', while 'info' appeared both on its own and with 'regular scan'. Finally, 'view-tree', 'install', and 'policy' each got a single mention, with 'view-tree' linked with 'transitive scan. The visualization can be seen in Figure 5.6

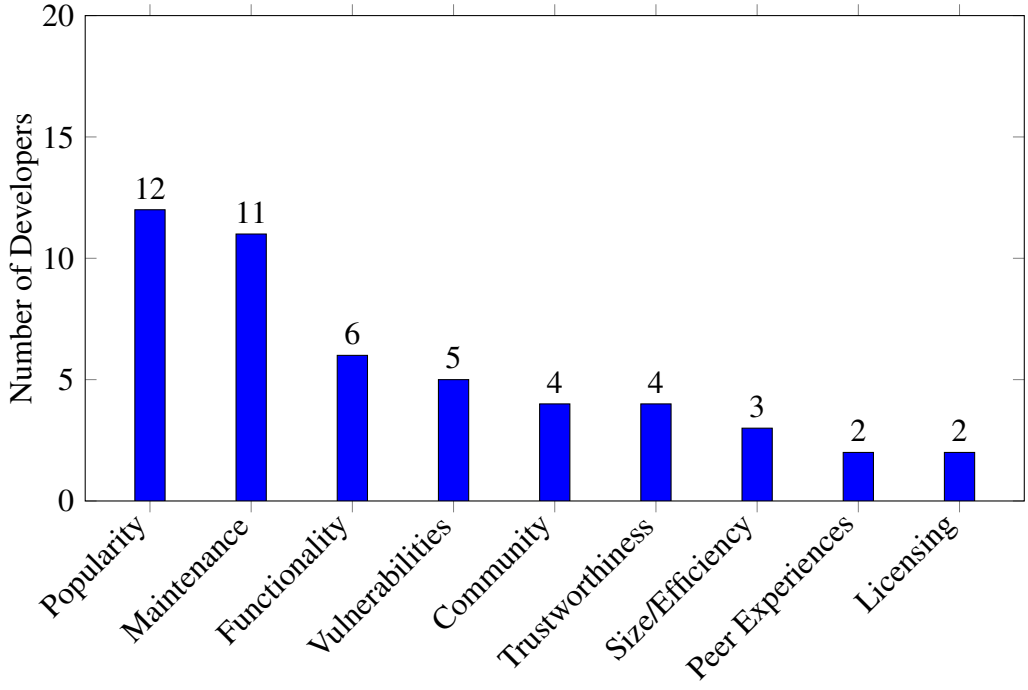

Figure 5.4: Factors Considered by Developers When Selecting Third-party Libraries.

When gathering feedback about potential additions or changes to the CLI tool, the responses from developers were diverse. Some felt that the current version of the tool met their needs well. Another group mentioned they would need more regular interaction with the tool in their daily tasks before giving more specific feedback. They felt that more use might uncover other areas for improvement.

Nevertheless, a number of developers shared specific ideas for the tool. These suggestions included:

1. Integrating the tool to work directly with npm install rather than operate as a separate entity. ID-2
2. Enabling automatic policy blocking for libraries that fall below a certain trust threshold. ID-12
3. Introducing HTML-based reporting. ID-6, ID-10, ID-15
4. Implementing more visually distinct alerts for libraries with notably low trust scores ID-2, ID-16
5. Making tabled information more intuitive and less technical. ID-3, ID-4
6. Allowing users to set trust levels for different libraries individually. ID-19

However, it is crucial to remember that many of these developers had only a brief experience with the tool. Their feedback was based on initial reactions, not extensive usage.

Interviewee ID	Years of Experience	Role	Company Size (Client)	Use of 3rd-Party Libraries (1-5)
1	3	Full-Stack	15-20	3
2	23	Full-Stack	30,000	5
3	20	Full-Stack	300	5
4	15	Full-Stack	40.000	4
5	29	Cloud Architect / Full-Stack	350	5
6	10	Cloud Architect / Full-Stack	20	4
7	15	Full-Stack	300	4
8	3	Front-End	20	4
9	3	Full-Stack	80	2
10	10	Full-Stack	250	4
11	6	Full-Stack	10	2
12	3	Full-Stack	20	4
13	3	Front-End	200	3
14	3	Front-End	200	4
15	11	Full-Stack	400	4
16	8	Full-Stack	90.000	3
17	2	Full-Stack	10	4
18	25	Full-Stack	20.000	4
19	6	Full-Stack	400	4
20	2	Full-Stack	130	5

Table 5.1: Demographic Information of Software Developers

5.2.3 Usability

All respondents unanimously agreed that the tool was straightforward to use, install, and set up. This consensus was consistent even though the tool is not available in the npm registry. Developers found the process of cloning and running the tool hassle-free. Furthermore, they universally appreciated the CLI's intuitiveness, emphasizing that no flags or commands were confusing. The comprehensive documentation provided, especially through the help flag, received commendations for its clarity and informativeness.

In terms of understanding the tool's output, an overwhelming 95%(19 out of 20 developers) confirmed that the results were clear and easily interpretable. However, there were a few areas of feedback. Some developers pointed out that table entries occasionally overflowed due to the length of the names. Others felt that the data naming was not as user-friendly as it could be, and the values lacked context. We anticipated these comments due to design limitations we previously addressed. The backend of TrustSECO, particularly the distributed ledger, could not supply more user-friendly information. We acknowledged the possibility of rectifying this on the frontend, but we prioritized backend improvements for TrustSECO DL as a future enhancement. A single developer felt the information was presented as an overwhelming block of data.

To sum up the usability section, graphical representations are absent since the majority of the feedback indicated near-complete satisfaction.

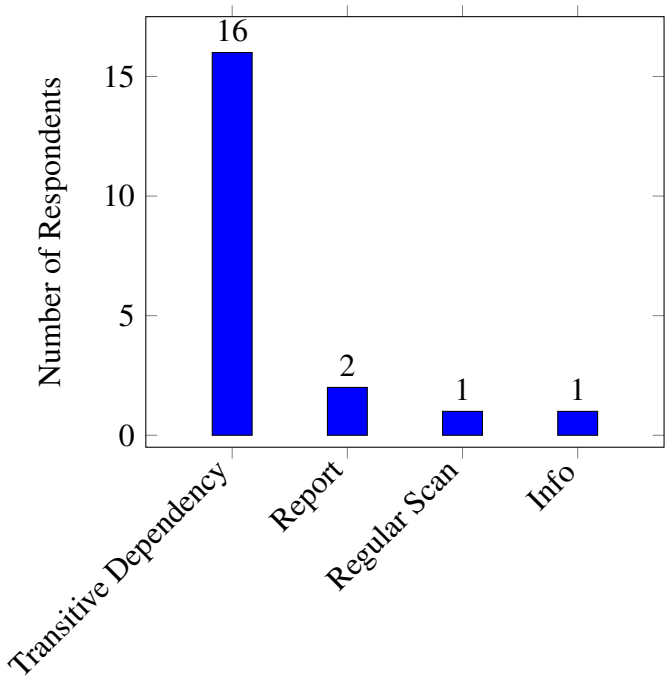

Figure 5.5: General Feedback on the Tool.

5.2.4 Functionality and Features

When the developers were asked if the trust score feature was useful to them, before they installed an up and if changed their mind, overwhelmingly positive feedback emerged from the surveyed developers. Specifically, 19 out of 20 developers believed that this feature improved their understanding of the packages they were considering. They felt it promoted more careful decision-making.

ID-2 response: *"Definitely, because it provides a much better understanding. Otherwise, I would have to visit the npm homepage or their GitHub, which is where most information typically is. I'd try to read through it, scan it, and decide if it looks good. I often go through the code, but if you get a trust score back that says it's 90% or something similar, it's already a big help. It makes it easier to decide, 'OK, yeah, we can use this.' Then, maybe after some time using it, we can do some additional checks."*

ID-5 response: *It can surely help. I've been thinking about at what point I would decide not to install a package. The trust score should make me pause and consider for a few seconds longer before installing, especially from a security standpoint. Vulnerabilities and other risks need to be more prominently highlighted. A low trust score should be a significant deterrent, emphasizing the potential risks. I believe there should be a clear explanation as to why a package should not be installed. It's not enough for it to be marked in red; there needs to be a concise yet informative explanation, in a few sentences or words, outlining the specific reasons for avoiding it.*

Interestingly, 2 out of these 20 developers said that before this feature, they had not considered the trustworthiness of specific libraries. Now, they have begun to see the importance of being cautious before using external libraries, especially in larger projects that depend on many libraries. Some mentioned that using standard commands, such as "npm install", might sometimes cause them to miss details about a package. The trust score acts as a guide to prevent such oversights

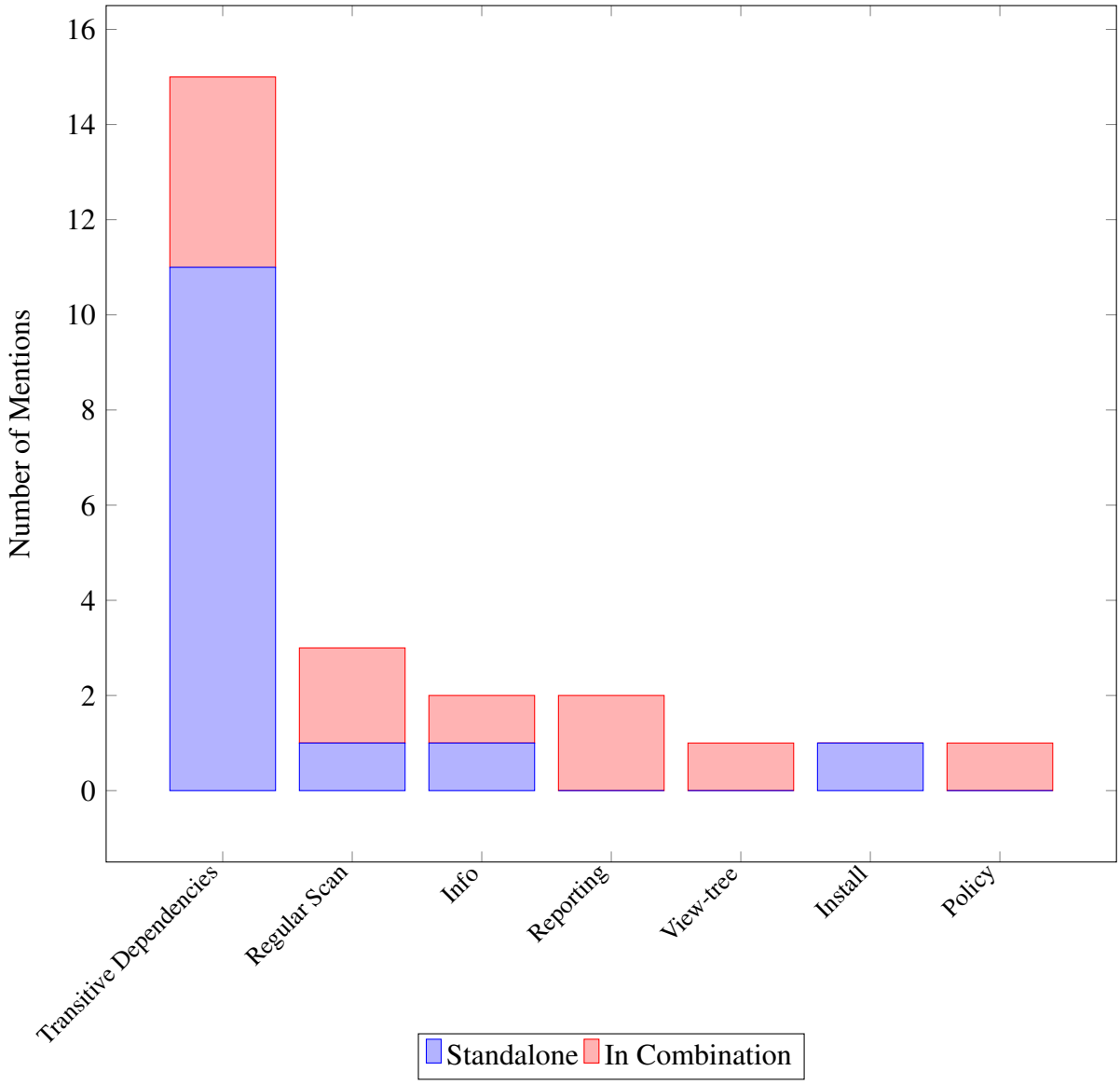

Figure 5.6: Distribution of mentioned features by developers.

ID-12 response *Prior to being introduced to the tool, I hadn't given much thought to the concept of 'trust' in packages. However, after seeing the trust score feature, I've come to realize its significance. It's undeniably useful, not just for me but for everyone. It emphasizes the importance of caution when deciding which packages to integrate into one's project.*

ID-14 response *Initially, I hadn't considered this, as I typically navigate to websites to search, read, and gather information. However, having all of that directly on the CLI is incredibly beneficial for developers. I believe many organizations will find great value in using this tool.*

However, there were concerns. Some developers felt the trust score could be unfair to new packages. Just because a package is new and has a low trust score does not mean it has problems. There were also doubts about how the score is calculated. For example, a library that does not have frequent updates might still be reliable. It might just be a small tool that works well and does not need constant changes. Also, using metrics such as the number of GitHub stars might not always reflect the true value of a library.

ID-10 Response: *The trust score feature is quite insightful. If the tool flags a library with serious vulnerabilities, I'd definitely be hesitant to use it. However, it's interesting to note that a new package might not have vulnerabilities but could still have a low trust score due to its novelty and fewer downloads. This certainly adds another layer to the decision-making process when selecting packages.*

ID-20 software engineer suggested a change: *if a main library uses another library with a low trust score, then the main library's score should not be higher than this sub-library. This would address situations where a main library seems trustworthy but depends on a less reliable sub-library. Additionally, I find some of the metrics are questionable - although these aren't part of the CLI but the service itself. The yearly commit count was given as an example. A library that is working perfectly doesn't need anything committed, so that would drop to close to 0 lowering the trust score for no reason. Another one is the contributor count, I can't see any reason why the number of people working on it would matter.*

To sum up, while the trust score was appreciated for helping developers make better choices, some areas need further consideration to ensure the scores are both fair and helpful.

All developers unanimously agreed that the message provided in the CLI was both clear and actionable.

When inquired about the utility of the transitive scan, an overwhelming majority found it valuable. Specifically, 19 out of 20 developers recognized its benefits, with 15 out of 20 even considering it the most beneficial feature of the tool. One particular response from Developer 20 stood out: "I think checking transitive dependencies should not even be a question. Check everything I am installing every time please." This sentiment underscores the importance and success of the transitive scan feature. This feedback affirms that our choice to incorporate this feature was both well-founded and well-received by the users. On the contrary, only one developer found it to be somewhat beneficial.

When asked about the value of additional package information (such as GitHub stars and maintainability), a large portion of developers responded positively. Specifically, 12 out of 20 found the feature "very valuable," often citing the time-saving convenience of the CLI tool compared to manual GitHub repository browsing.

However, there were variations in feedback. 4 out of 20 felt certain data points, such as GitHub stars, first release date, and contributor count, were redundant or not particularly useful. Additionally, one developer was confused about data presentation, specifically regarding package age.

Lastly, a group of 3 developers, while understanding the feature's broader appeal, person-

ally preferred accessing information directly from GitHub.

All of the developers said that they found the scan feature to be useful to them. when they were asked if the feature is beneficial for them

When prompted about the utility of the report feature, the developers offered varied feedback. A majority, with 12 out of 20, regarded it as a highly useful tool for reporting purposes. A smaller subset, 3 out of 20, expressed interest in using the feature but suggested enhancements, such as customizable headers and the option to export to HTML.

Two developers acknowledged the general usefulness of the tool but felt it would not serve their specific needs, suggesting that others might find it more beneficial. Lastly, another 3 out of 20 saw potential in the feature but believed its real value would come into play in larger projects with an extensive list of dependencies.

When asked about the usefulness of the policy feature, developers shared mixed yet predominantly positive responses. A clear majority, with 14 out of 20, found the feature beneficial, especially in larger organizational settings where there are stringent guidelines on package usage. The ability to easily set and adjust these rules directly was seen as an advantage. They also mentioned that this feature could be integrated into DevOps pipelines to ensure rules are consistently followed.

Some developers highlighted the feature's role in controlling projects and ensuring software safety. They felt it was particularly good at preventing less experienced developers from using potentially risky software parts.

However, not everyone was in agreement. 3 out of 20 developers were either neutral or had reservations about the feature. They felt it might not be as relevant to their own work, but could see its value for others. The last 3 developers acknowledged the feature's potential without diving deep into its impact on their specific tasks.

When asked what you think about the info command, an overwhelming majority (19 out of 20 developers) deemed it beneficial. They emphasized its significant assistance during their package library research and appreciated the convenience of quick lookups. However, despite their recognition of its utility, there was unanimous feedback highlighting concerns over data presentation. This was consistent with previous usability feedback where developers mentioned that the data naming could be more intuitive, and the presented values required clearer context.

5.2.5 Performance

When asked about the tool's performance, the majority of developers indicated that they did not observe any significant performance issues. Some of the respondents reasoned that performance was not a primary concern for them, given that they will use the tool once in a while. For them, the benefits of ensuring trustworthiness outweighed any potential performance drawbacks. Those who did observe performance hiccups primarily noted them during the transitive dependency scans. Nevertheless, these respondents felt that such delays were anticipated due to the recursive nature of fetching all the packages. Some even pointed out that manually obtaining all the transitive dependencies would be a far more time-consuming task. Additionally, positive feedback was received about the loading spinner; respondents found it beneficial as it assured them the tool was actively working and had not stalled. Overall there were no substantial concerns regarding the tool's performance.

Some of the software engineers responses:

- ID-1 *"Because there's no hurry, there's no millisecond requirement using this tool when you're developing software. I don't see the performance as an issue, so I would say it is good"*

- ID-3 *"When I use a CLI, there are moments where nothing appears on the screen, leaving me unsure if it's frozen or processing. Having visual feedback, such as a loading indicator, is helpful. Overall, the feature seems engaging and practical. The performance is also good."*
- ID-5: *"The performance was good because it deals with transitive dependencies using a recursive approach. We have to look at each dependency one by one, so the speed seems fine. Doing this by hand would take three days, so waiting a few minutes is no big deal."*
- ID-8: *"Since the basic installation command is not slow, I don't think it will matter much to me. Scanning the transitive dependencies might take more time, but on the other hand, it's a useful feature to have."*
- ID-20: *"I don't care about the package installation process that much. Even if it was significantly slower, I just wait for it 1 time and never again."*

5.2.6 Comparison with Existing Tools/Practices

A significant majority of 85%(17 out of 20 developers) mentioned that they primarily rely on manual methods to determine the trustworthiness of npm packages. These manual checks include visiting the npm website, and GitHub repositories, checking download numbers, and reading community forums, and reviews.15%(3 out of 20 developers) developers cited the use of other tools, such as Dependabot and "rong core tools", but noted distinctions between these tools and TrustSECO.js. The results can be seen in Figure 5.7.

Around 45% (9 out of 20 developers) mentioned that the tool would likely supplement their current practices or tools. They viewed the tool as an additional layer of assurance, complementing other methods or tools they currently use. Approximately 40% (8 out of 20 developers) indicated that the tool would replace their current methods. 3 out of 20 developers expressed conditional sentiments. Their adoption of the tool would depend on its availability in the npm registry, its database stability, or directives from higher-ups. One developer emphasized the need for long-term use before deciding on the tool's permanence in their workflow.

5.2.7 Future Development and Improvement

In discussing potential developments and improvements for the tool, developers provided a variety of suggestions. A prominent idea was the integration of the tool with Visual Studio Code. Many emphasized their preference for this, noting that they spend a significant amount of time within the VS Code environment. This would allow them to run commands seamlessly without having to switch interfaces.

Another frequently mentioned enhancement was integration with DevOps pipelines, enabling real-time checks and validations. This would streamline the development process by ensuring that the code aligns with organizational policies before deployment.

HTML reporting was also a common suggestion. Several developers expressed a preference for this feature, emphasizing that it would enable more precise data visualization. For instance, a transitive dependency, which might be complex when viewed in the terminal, could be more lucidly visualized in an HTML report.

Other developers expressed a desire for a Chrome extension. This extension would, upon visiting a GitHub page, automatically fetch data to provide the trust score and relevant information about the repository. Furthermore, there was an innovative proposal to directly integrate

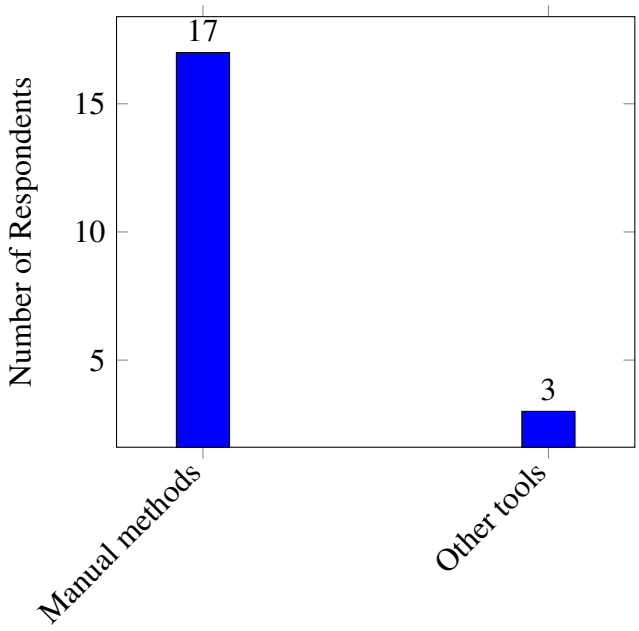

Figure 5.7: Methods to Check the Trustworthiness of npm Packages.

a trust score with npm, potentially intercepting the npm install command. This would allow users to gauge the trustworthiness of packages during installation.

Another noteworthy suggestion was the assignment of verification badges or trust scores to established corporations or reputable repositories. Such badges would give users immediate insight into the credibility of a package.

While there were many suggestions, a segment of the developers felt that the tool did not need any immediate enhancements or future developments.

5.2.8 Closing Thoughts

When surveyed about their potential utilization of the tool, a substantial 70%(14 out of 20 developers) of developers expressed a definite willingness to use it. Within those positive responses, a subset of 15%(3 out of 20 developers) had specific reservations or conditions for adoption, encompassing concerns about licensing permissions, the quality of the backend, or the availability of integrations, such as with Visual Studio Code. A smaller fraction, 5%(1 out of 20 developers), mentioned they would be inclined to use the tool if it underwent further enhancements. On the more cautious end, one developer (5% of respondents) stated they would consider deploying the tool only if it became a company mandate, while another (another 5%) would be open to it upon receiving approval from their supervisor. A representation can be seen in Figure 5.8

Regarding referrals, there was unanimous agreement among developers about recommending the tool to peers or colleagues. While some showed unconditional support, others would suggest it if a colleague articulated a need for such a tool. Furthermore, all participants expressed enthusiasm for future versions and updates of the tool, keen to see its evolution in developmental or production environments.

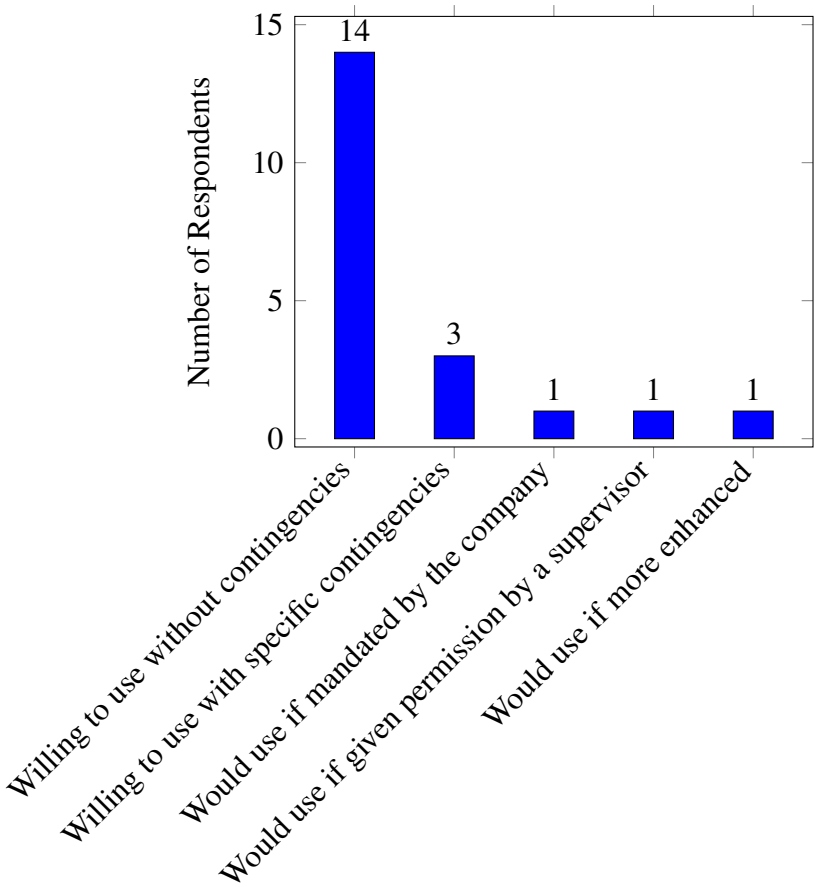

Figure 5.8: Intent to Use the Tool in the Future.

Chapter 6

Discussion

In this study, our primary aim is to identify gaps in the literature concerning security software tools within the npm ecosystem. This was achieved through a systematic literature review (SLR) of existing academic papers. A key component of our research involved examining how developers strike a balance between the functionality and security of npm packages and what the gaps in the current security tools in the academic world. To bridge the gap identified in existing literature, as observed with tools like DapReveal [5], Nodest [55], and NodeXp[57] and all the other tools observed in the SLR, we introduced TrustSECO.js. This tool is designed to enhance the safety of developers by ensuring secure tool installation practices

Our systematic literature review has made a significant contribution by providing a foundation for future research in the realm of security tools. TrustSECO.js not only fills the identified gap but also serves as a baseline for researchers or developers interested in designing similar tools with the best practices that we got from the grey literature like Czapski [18], Prasad et al.[63], the GNU standards[28] and Jeff [19]. The Design Science methodology employed in our research outlines the consideration of both functionality and user experience when creating such tools. Drawing from various guidelines, we developed TrustSECO.js with a focus on its potential impact and usability.

The empirical data from our interview study suggest that developers welcomed our tool. This positive reception reinforces the effectiveness of our design science approach. Moreover, our findings contribute to the literature by offering a detailed process and guidelines for developing command-line interface tools specific to the npm or JavaScript ecosystem.

Our diverse pool of respondents all shared an appreciation for third-party libraries, whether for professional or personal projects. A recurring trend we identified was a general oversight by developers regarding the security of these libraries. Interestingly, many were swayed by factors such as popularity, maintenance, and functionality.

On the topic of library policies, we observed variance based on company size and sector, with larger companies, especially those in the financial sector, exhibiting stricter controls over third-party library usage. Feedback on TrustSECO.js was predominantly positive, with the transitive dependency scanner feature being particularly well-received, which was somewhat unexpected.

There were, of course, areas of improvement highlighted by the users. Some users desired enhanced reporting features, while others provided feedback on aesthetics. Usability, however, was universally commended. This positive reception indicates the successful implementation of our design science principles, offering an intuitive tool with clear and actionable guidance. While some users felt the detailed technical data was excessive, this was an integral aspect of the distributed ledger system.

The concept of a 'trust score' was enlightening for many developers. Previously, many, including myself, installed npm packages without significant security considerations. This observation was especially true for developers not associated with enterprise-level or financial sector companies.

In terms of tool performance, most users were more concerned with user experience than raw speed. Many praised the spinner feature, recognizing it as a valuable real-time feedback mechanism. While performance was not a major concern, the majority were willing to sacrifice some speed in favor of enhanced security and trustworthiness.

When considering integration with existing workflows, many developers believed TrustSECO.js would enhance their productivity. They expressed interest in broader platform integration and eagerly anticipated its stable production release. Compared to existing literature, our approach of incorporating real-world developer feedback sets our tool apart, lending it increased credibility.

In conclusion, TrustSECO.js has shown significant promise as a minimum viable product (MVP). This research provides a foundation for further development or for other researchers wishing to expand upon this work.

6.1 Research questions

At the close of our study, we have effectively tackled all the research questions we began with. Importantly, the answers to three out of these five questions were already covered in our Literature Review. To provide a clear overview, we will list each research question and its answer:

RQ1: What kind of trust reinforcement mechanisms exist in package ecosystems?

Answer: Addressing this question is challenging. As discussed in subsection 3.5.1, various tools adopt different methodologies, ranging from dynamic and static analyses to sandboxing and the application of Machine Learning techniques for vulnerability detection. These tools can be integrated into CI/CD pipelines and assess a diverse array of threats, including transitive vulnerabilities, code injections, zero-day vulnerabilities, clone packages, malicious code, and permission systems. Each tool serves distinct use cases. A notable observation is the lack of pre-installation safeguards among these tools. Predominantly, as presented in Table 3.6, they operate post-installation, either by scanning the existing codebase or necessitating manual package input for checks. The gap we identified in the study is the need for a tool to protect users before they even install a package, ensuring its trustworthiness. While many tools simply check for vulnerabilities, they do not provide a trustworthiness score such as TrustSECO does. These represent the two primary gaps we have observed.

RQ2: How effective are trust reinforcement mechanisms in package ecosystems?

Answer: The evaluation of trust reinforcement mechanisms in package ecosystems, as reported in the literature, generally indicates positive outcomes. Tools analyzed in various studies demonstrated high precision in detecting issues and security vulnerabilities in libraries. However, there's a variance in the reported effectiveness. Some studies did not disclose effectiveness metrics, while those that did revealed differing perceptions of effectiveness. This variation is often attributed to factors like the occurrence of false positives or negatives, performance measurements, and the number of errors identified.

In terms of capabilities, certain tools were effective in identifying basic errors, whereas others were advanced enough to detect zero-day vulnerabilities. The diversity in techniques used by these tools complicates the task of establishing a uniform standard for comparing their

effectiveness. Additionally, the literature often points to limitations such as high rates of false positives or negatives, reliance on specific platforms like GitHub, and constraints unique to particular tools. These factors collectively contribute to the challenge of assessing the overall efficacy of trust reinforcement mechanisms in package ecosystems.

RQ3: How do software engineers perceive the balance between adding new features and ensuring security while using third-party packages?

Answer: In Subsection 3.5.2 and Subsection 3.5.3, we examined the literature related to the utilization of third-party libraries and the security awareness of developers. Our analysis revealed that developers demonstrate an increased level of security consciousness when selecting third-party libraries, regardless of whether they are complex libraries or simple packages. Nevertheless, while selecting libraries or attempting to upgrade them, the emphasis is consistently placed on prioritizing the library's usefulness or the stability of the system rather than its security. Security vulnerabilities are not their first priority. When selecting a library, individuals tend to value indicators such as documentation quality, download numbers, and star counts. A significant proportion of developers are unaware of the vulnerabilities present in their libraries, and many of them neglect to update these libraries due to perceiving it as a burdensome task.

RQ4: How can third-party trust score tools be effectively integrated into a widely-used package manager through the development of a command-line interface?

Answer: In Chapter 4, we detailed the process of designing the npm tool. We employed commander.js, a streamlined third-party library, to develop the core commands and flags for our CLI tool. To fuse TrustSECO with the npm environment, we leveraged API calls to retrieve necessary data from the distributed ledger. Adhering to best practices for usability and intuitiveness, we successfully crafted a CLI tool that seamlessly interfaces with TrustSECO.

RQ5: How useful is the implemented tool from an expert's perspective?

Answer: Experts largely viewed the CLI tool favorably. While not all saw a direct application for their individual needs, they recognized its potential value within the broader open-source landscape. Many found the tool intuitive with a commendable user experience. Different functionalities resonated with different experts, indicating its versatility. The overarching sentiment was positive, both in terms of its usability and its contribution to the community. Many expressed a willingness to use it if the tool was made available to them and recommended it to colleagues. Performance-wise, it met expert expectations.

MRQ: How can package managers reinforce trust within the worldwide Software Ecosystem?

Answer: In addressing our main research question, our findings shed light on the integration potential of TrustSECO with package managers such as npm. As delineated in the results, TrustSECO's integration can be streamlined by leveraging best practices and adopting optimal toolsets. The need for an effective CLI tool, which addresses a clear gap in the domain, was verified by our research. Even though several tools exist both in literature and the market, the distinct value of providing trust scores to developers was underscored in our interview study. As evidenced by developer feedback, such scores were not just beneficial but crucial in reshaping developer perspectives on trust features in SECO. This, in turn, contributes to enhanced productivity by saving them significant time and fostering more informed decision-making.

6.2 Limitations - Threats of validity

As with all research endeavours, this study has its limitations and biases, which we have acknowledged throughout the thesis.

1. **Backend Limitations of TrustSECO:** The most significant limitation stems from TrustSECO's backend. We were unable to utilize real data for testing transitive dependencies and the scanning capabilities of the tool. This limitation critically impacted our ability to validate the tool's effectiveness comprehensively.

2. **Trust Score Ambiguity:** The derivation of the trust score is another area of concern. Assigning trust scores based on certain GitHub features might disadvantage new or smaller packages, especially those that fulfill specific functions and do not require frequent updates. It is crucial for potential users to understand that the trust score should be interpreted with caution. Its accuracy is not guaranteed, especially given that data retrieval often encounters errors.

3. **Deployment Constraints:** At this juncture, the tool cannot be deployed to the npm registry due to the unavailability of TrustSECO's Distributed Ledger (DL). Even if available, the DL has existing issues that need addressing before it can be deemed stable. The information retrieved from the DL is not always user-friendly or human-readable.

4. **Tool Dependencies:** Our tool heavily relies on npm for information on transitive dependencies. Furthermore, it operates under the limitation that the DL only recognizes GitHub package names, which can be different from npm names. This differentiation can impact the tool's efficiency and accuracy.

5. **Initial Phase and MVP Status:** It is vital to understand that this tool is in its initial phase. It is best described as a Minimum Viable Product (MVP) - a prototype that showcases the potential of a fully developed version. Based on the feedback we have gathered, there is a clear path to refining the tool into something highly beneficial for developers.

6. **Potential Biases:** Biases might have been introduced during the research phase. While effort was put into identifying npm scanner tools for vulnerabilities or security checks, some might have been overlooked. This oversight, especially concerning gray literature or open-source tools not reported in academic publications, could introduce bias. Feedback on the tool's effectiveness might also be skewed. For instance, if we predominantly gather feedback from a specific subset of professionals, such as software engineers employed at prominent tech companies (represented by the acronym FAANG, which stands for Facebook, Amazon, Apple, Netflix, and Google), the results could differ from a more diverse sample. The framing of our interview questions could also introduce bias: more negative phrasing could elicit different responses.

In conclusion, this tool remains in its developmental stage, created by a single researcher with limited resources and time. When reviewing or considering the tool for application, it is essential to set realistic expectations.

Chapter 7

Conclusion

In this thesis, we conducted a thorough literature review on npm security tools, developer security practices and behavior, and the use of third-party libraries. Our examination of various tools revealed a significant gap: the lack of pre-installation safeguards for npm packages. This was evident when compared to the predominantly post-installation focus of tools such as Plumber [70], Affogato [27], and NodeXp [57], as well as others discussed in section 3.5.1. This finding is particularly crucial considering that developers often prioritize functionality over security [38, 75, 61, 35].

Recognizing this gap, we introduced the TrustSECO.js CLI tool, leveraging the TrustSECO distributed ledger to access information and trust scores for diverse packages. While primarily designed for pre-installation processes, our extensive review of existing academic and industry research inspired the implementation of additional features. For instance, inspired by Ferreira et al. [24], who designed a permission-based system to protect against malicious updates, we incorporated a similar feature. This allows for a permission-based approach in our tool, enabling companies to enforce policies that restrict the installation of certain libraries. Furthermore, Jarukitpipat et al.'s study [37] on the risks associated with transitive dependencies led us to add transitive scanning capabilities, enhancing our tool's security measures.

Following the tool's development, we engaged 20 developers in interviews, probing into their practices and seeking feedback on our tool. The response was overwhelmingly positive. Developers recognized its potential and deemed it a promising venture. However, no software is without its challenges. Being the inaugural version of our tool, it had its share of critiques. A significant portion of this feedback revolved around the displayed data, which is intrinsically tied to the TrustSECO DLT, a factor beyond the scope of our study.

In conclusion, this thesis journey has been a rewarding one. We successfully designed a preliminary version of a tool aimed at seamlessly integrating TrustSECO with the npm ecosystem. The overarching goal was to enhance the safety of the open-source community, and we believe we have made strides in that direction. While the tool is still evolving and the feedback has illuminated areas of improvement, especially concerning TrustSECO, the positive reception from developers underscores its potential. We delve deeper into recommendations for future iterations in the subsequent section.

7.1 Contributions

This research makes pivotal strides in understanding and enhancing software trustworthiness and security within software ecosystems (SECOs). The primary contributions are:

1. **Uncovering software engineer Practices in npm Ecosystem**

   We present a comprehensive analysis of software engineer habits within the npm ecosystem, revealing:

   - Motivations behind the adoption of third-party packages.
   - Trends related to package updates' frequency and immediacy.
   - software engineer reactions to vulnerabilities and their influence on decision-making.
   - software engineer awareness of security.

   This analysis addresses the challenge of identifying best practices among developers regarding the use of third-party libraries. Understanding these trends helps in shaping the tool we have developed.

2. **Vulnerability tools analysis**

   Additionally, a comprehensive examination of existing tools within the literature was conducted, assessing their effectiveness, which will serve as a motivating factor for the integration of our prototype.

   This effort identifies gaps in the current landscape of npm tools, guiding us in creating a unique tool that fills these gaps.

3. **Enhanced TrustSECO Prototype**

   Our research has pioneered the development of a prototype that augments TrustSECO's capabilities. This innovative prototype called *TrustSeco.js* seamlessly integrates with pivotal platforms such as npm, fortifying the trust layer within SECOs.

   This contribution addresses the challenge of creating a practical tool that fills the identified gaps in the existing npm ecosystem.

4. **In-depth Feedback Analysis**

   We present an extensive analysis of feedback from software engineers concerning:

   - Adoption rates of the TrustSECO.js CLI tool.
   - Effectiveness of TrustSECO.js

   This analysis helps determine the effectiveness and reception of our tool among professionals, ensuring its relevance and utility in the field.

7.2 Recommendations for Future Work

From the feedback gathered from developers during the interview study in Section 5.2 especially in Subsection 5.2.7, we have identified several areas for potential improvement and exploration. While some of these ideas might step beyond the current study's immediate realm, they provide valuable insights for future endeavors. Key areas of focus include:

1. **Improving the TrustSECO Distributed Ledger:** A foundational element of our tool, the TrustSECO Distributed Ledger, needs further enhancement. By setting it up on a dedicated server with daily updates, we can ensure users consistently receive the latest package details, optimizing the tool's effectiveness.

2. **Introducing HTML Reporting:** A common suggestion was the addition of an HTML-based reporting feature. This would produce a user-friendly HTML document, presenting clear and interactive data views or dependency outlines.

3. **Broadening Tool Accessibility**: It is worth considering creating versions of our tool for browsers, software development platforms, and DevOps systems. This could make the tool more integral to developers' daily tasks.

4. **Adding Trust badges**: To avoid biases in our tool's outputs, we could incorporate badges or signs that show validations from well-respected sources or major library maintainers.

5. **Integration Across Multiple Package Managers:** The tool should be made available across diverse programming language ecosystems and not just limited to npm users. It would be beneficial to design the tool such that users can specify which package manager they intend to use during installation.

By addressing these points, we can position future versions of our tool and research to resonate more deeply and effectively within the developer community.

Appendix A: Interview Questions

Before the interview began, I introduced myself and explained the research I was conducting and the purpose of this interview. Although I had previously communicated this information in the email invitation, I felt it was important to reiterate it. Following the introduction, I asked for their permission to record the meeting.

Afterwards, we began with the questions below:

1. **Introduction and Context**
    - 1.1. Introduce myself, what research I am doing, and what is the goal of this interview.
    - 1.2. How many years of experience do you have in software development?
    - 1.3. Would you classify yourself as a frontend, backend, or full-stack developer?
    - 1.4. Number of employees in the company?
    - 1.5. How frequently do you use third-party libraries in your projects? Please rate on a scale from 1 to 5, where 1 means you never use them, and 5 means you use them every day.
    - 1.6. What factors are most important to you when deciding whether to use a third-party library? (e.g., popularity, maintenance activity, known vulnerabilities, etc.)
    - 1.7. Do you check the security of a third-party library before using it, if yes, how?
    - 1.8. Does your company or team have a policy regarding the use of third-party libraries? If yes, how does it influence your choices? And how is the company enforcing those policies?
2. **General Feedback**
    - 2.1. What are your initial thoughts after using the CLI tool?
    - 2.2. Were there any features that stood out to you?
    - 2.3. Were there any features that you felt were missing or could be added?
3. **Usability**
    - 3.1. Was the tool easy to install and set up?
    - 3.2. Did you find the command-line interface intuitive? Were there any commands or flags that were confusing?
    - 3.3. Was the output information clear and easily understandable?
4. **Functionality and Features**
    - 4.1. How useful did you find the trust score feature? Did it change how you felt about installing a package?
    - 4.2. Was the warning message clear and actionable?
    - 4.3. How did you feel about the tool checking transitive dependencies? Was this feature beneficial for you?
    - 4.4. How valuable was the additional package information (such as GitHub stars, maintainability, etc.)?

4.5. What do you think about the full scan of the dependencies? Was the output helpful?

4.6. What do you think about the report feature, was it useful?

4.7. What do you think about the policy feature, was it useful?

4.8. What do you think about the info command, was it useful?

5. **Performance**

5.1. Did you notice any performance issues when using the tool? For instance, did it significantly slow down the package installation process?

6. **Comparison with Existing Tools/Practices**

6.1. Are there other tools or manual methods you use to check the trustworthiness of npm packages? How does this tool compare?

6.2. Would this tool replace or supplement any current tools or practices you use?

7. **Future Development and Improvement**

7.1. Would you like integrations with any other tools or platforms?

8. **Closing Thoughts**

8.1. Will you use this tool?

8.2. Would you recommend this tool to your colleagues or peers?

8.3. Would you be interested in future updates or iterations of the tool?

8.4. Any additional comments or feedback that we have not covered?

8.5. Can we contact you if we have any further questions?

Link of the interview questions and answers for reproducibility: One note

Appendix B: Interview Consent Form

Universiteit Utrecht

DECLARATION OF CONSENT for participation in:

Trust in the Worldwide Software Ecosystem: Integrating TrustSECO in NPM

I hereby confirm:

- that I have been satisfactorily informed about the study through the information letter;
- that I have been given the opportunity to ask questions about the study and that any questions I asked have been satisfactorily answered;
- that I have had the opportunity to carefully consider participation in this study;
- that I voluntarily consent to participating.
- that I am free to withdraw at any time, without giving a reason, without my medical care or legal rights being affected.
- that participation involves participating in an interview to validate, evaluate and elicit new requirements of a design science research artifact.
- that my personal data, which links me to the research data, will be kept securely in accordance with data protection guidelines, and only available to the immediate research team.
- that the research data, which will be anonymised (not linked to me), may be shared with others.
- that I am free to contact any of the people involved in the research to seek further clarification and information.
- that under freedom of information legalization I am entitled to access the information I have provided at any time

Signature: _____ Date: ___ / ___ / ____

To be completed by the researcher carrying out the study:

I declare that I have explained to the above-mentioned participant what participation in the study entails.

Name: Angel Temelko

Signature: _____

Date: ___ / ___ / ____, _____

Appendix C: Ethics and Privacy from the University

Ethics and Privacy Quick Scan pdf link

Bibliography

[1] Cli style guide by heroku. https://devcenter.heroku.com/articles/cli-style-guide, 2023. Accessed: 9/20/2023.

[2] npm Documentation. `https://docs.npmjs.com/`, 2023. Accessed: 2023-11-06.

[3] R. Abdalkareem, V. Oda, S. Mujahid, et al. On the impact of using trivial packages: an empirical case study on npm and pypi. *Empirical Software Engineering*, 25:1168–1204, 2020.

[4] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 385–395, New York, NY, USA, 2017. Association for Computing Machinery.

[5] Mahmoud Alfadel, Diego Elias Costa, Emad Shihab, and Bram Adams. On the discoverability of npm vulnerabilities in node.js projects. *ACM Trans. Softw. Eng. Methodol.*, 32(4), may 2023.

[6] Ellen Arteca and Alexi Turcotte. Npm-filter: Automating the mining of dynamic information from npm packages. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 304–308, New York, NY, USA, 2022. Association for Computing Machinery.

[7] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staicu. Secbench.js: An executable security benchmark suite for server-side javascript. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1059–1070, 2023.

[8] Vittorio Dal Bianco, Varvana Myllärniemi, Marko Komssi, and Mikko Raatikainen. The role of platform boundary resources in software ecosystems: A case study. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 11–20, 2014.

[9] G.A. Blanken, Y. Bok, A.N. van der Goot, V.A. Hoffman, G.J.J. Jansen, M.T. Loo, J.A. Schenkel, D.S. van der Spek, A. Vakili, A.W. Verhoef, and B.D. Wout. Trustseco computing science bachelor's thesis, 6 2022.

[10] Vasilis Boucharas, Slinger Jansen, and Sjaak Brinkkemper. Formalizing software ecosystem modeling. In *Proceedings of the 1st International Workshop on Open Component Ecosystems*, IWOCE '09, page 41–50, New York, NY, USA, 2009. Association for Computing Machinery.

[11] Darion Cassel, Wai Tuck Wong, and Limin Jia. Nodemedic: End-to-end analysis of node.js vulnerabilities with provenance graphs. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, pages 1101–1127, 2023.

[12] Kyriakos Chatzidimitriou, Michail Papamichail, Themistoklis Diamantopoulos, Michail Tsapanos, and Andreas Symeonidis. npm-miner: An infrastructure for measuring the quality of the npm registry. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 42–45, 2018.

[13] James Check and Russell K Schutt. *Research methods in education*. SAGE Publications, 2012.

[14] X. Chen, R. Abdalkareem, S. Mujahid, et al. Helping or not helping? why and how trivial packages impact the npm ecosystem. *Empirical Software Engineering*, 26(27), 2021.

[15] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Softw. Engg.*, 26(3), may 2021.

[16] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Code-based vulnerability detection in node.js applications: How far are we? In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1199–1203, 2020.

[17] Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, and Bram Adams. On the untriviality of trivial packages: An empirical study of npm javascript packages. *IEEE Transactions on Software Engineering*, 48(8):2695–2708, 2022.

[18] Adam Czapski. Guidelines for creating your own cli tool. `https://medium.com/jit-team/guidelines-for-creating-your-own-cli-tool-c95d4af62919`, 2022. Accessed: 9/20/2023.

[19] Jeff D. 12 factor cli apps. https://medium.com/@jdxcode/12-factor-cli-apps-dd3c227a0e46, 2023. Accessed: 9/20/2023.

[20] Fernando López de la Mora and Sarah Nadi. An empirical study of metric-based comparisons of software libraries. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'18, page 22–31, New York, NY, USA, 2018. Association for Computing Machinery.

[21] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the evolution of technical lag in the npm package dependency network. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 404–414, 2018.

[22] Google Developers. Cloud-based protections, 2023. Accessed: 2023-09-03.

[23] Gavin D'mello and Horacio González-Vélez. Distributed software dependency management using blockchain. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 132–139, 2019.

[24] 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, 2021.

[25] Node.js Foundation. *Node.js Documentation*. Node.js Foundation, 2023. Accessed: 2023-09-28.

[26] 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, 2019.

[27] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. Affogato: Runtime detection of injection attacks for node.js. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA '18, page 94–99, New York, NY, USA, 2018. Association for Computing Machinery.

[28] GNU. 4 program behavior for all programs. https://www.gnu.org/prep/standards/html_node/Program-Behavior.html, 2023. Accessed: 9/20/2023.

[29] Hritik Gupta, Alka Chaudhary, and Anil Kumar. Identification and analysis of log4j vulnerability. In *2022 11th International Conference on System Modeling & Advancement in Research Trends (SMART)*, pages 1580–1583, 2022.

[30] Alan Hevner and Samir Chatterjee. *Design Science Research in Information Systems*, pages 9–22. Springer US, Boston, MA, 2010.

[31] Alan R. Hevner, Salvatore T. March, S. Ram, and J. Park. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[32] Raphael Hiesgen, Marcin Nawrocki, Thomas C Schmidt, and Matthias Wählisch. The race to the vulnerable: Measuring the log4j shell incident. *arXiv preprint arXiv:2205.02544*, 2022.

[33] Fang Hou, Siamak Farshidi, and Slinger Jansen. Trustseco: A distributed infrastructure for providing trust in the software ecosystem. In Artem Polyvyanyy and Stefanie Rinderle-Ma, editors, *Advanced Information Systems Engineering Workshops*, pages 121–133, Cham, 2021. Springer International Publishing.

[34] Fang Hou and Slinger Jansen. A systematic literature review on trust in the software ecosystem. *Empirical Software Engineering*, 28(1):8, 2022.

[35] Matthew Ivory, Miriam Sturdee, John Towse, Mark Levine, and Bashar Nuseibeh. Can you hear the roar of software security? how responsibility, optimism and risk shape developers' security perceptions. page 41, New York, NY, USA, 2023. ACM.

[36] Slinger Jansen, Michael A Cusumano, and Sjaak Brinkkemper. *Software ecosystems: analyzing and managing business networks in the software industry*. Edward Elgar Publishing, 2013.

[37] Vipawan Jarukitpipat, Klinton Chhun, Wachirayana Wanprasert, Chaiyong Ragkhitwetsagul, Morakot Choetkiertikul, Thanwadee Sunetnanta, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, and Kenichi Matsumoto. V-achilles: An interactive visualization of transitive security vulnerabilities. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[38] Md Mahir Asef Kabir, Ying Wang, Danfeng Yao, and Na Meng. How do developers follow security-relevant best practices when using npm packages? In *2022 IEEE Secure Development Conference (SecDev)*, pages 77–83, 2022.

[39] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, V. N. Venkatakrishnan, and Yinzhi Cao. Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1059–1076, 2023.

[40] Hee Yeon Kim, Ji Hoon Kim, Ho Kyun Oh, Beom Jin Lee, Si Woo Mun, Jeong Hoon Shin, and Kyounggon Kim. Dapp: Automatic detection and analysis of prototype pollution vulnerability in node.js modules. *Int. J. Inf. Secur.*, 21(1):1–23, feb 2022.

[41] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.

[42] Barbara Kitchenham, Rialette Pretorius, David Budgen, O. Pearl Brereton, Mark Turner, Mahmood Niazi, and Stephen Linkman. Systematic literature reviews in software engineering – a tertiary study. *Information and Software Technology*, 52(8):792–805, 2010.

[43] Maryna Kluban, Mohammad Mannan, and Amr Youssef. On measuring vulnerable javascript functions in the wild. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 917–930, New York, NY, USA, 2022. Association for Computing Machinery.

[44] Murat Koyuncu and Tolga Pusatlı. Security awareness level of smartphone users: An exploratory case study. *Mobile Information Systems*, 2019:1–11, 05 2019.

[45] R.G. Kula, D.M. German, A. Ouni, et al. Do developers update their library dependencies? *Empirical Software Engineering*, 23:384–417, 2018.

[46] R.G. Kula, D.M. German, A. Ouni, et al. Do developers update their library dependencies? *Empir Software Eng*, 23(1):384–417, 2018.

[47] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. Selecting third-party libraries: The practitioners' perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 245–256, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 672–684, New York, NY, USA, 2022. Association for Computing Machinery.

[49] Fernando López de la Mora and Sarah Nadi. Which library should i use?: A metric-based comparison of software libraries. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, pages 37–40, 2018.

[50] Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems: A systematic literature review. *Journal of Systems and Software*, 86(5):1294–1306, 2013.

[51] Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems – a systematic literature review. *Journal of Systems and Software*, 86(5):1294–1306, 2013.

[52] Tom Mens and Coen De Roover. An introduction to software ecosystems. Preprint, 2023.

[53] Hisham Muhammad, Lucas C. Villa Real, and Michael Homer. Taxonomy of package management in programming languages and operating systems. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, PLOS '19, page 60–66, New York, NY, USA, 2019. Association for Computing Machinery.

[54] Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. What are the characteristics of highly-selected packages? a case study on the npm ecosystem. *Journal of Systems and Software*, 198:111588, 2023.

[55] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. Nodest: Feedback-driven static analysis of node.js applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 455–465, New York, NY, USA, 2019. Association for Computing Machinery.

[56] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of node.js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 29–41, New York, NY, USA, 2021. Association for Computing Machinery.

[57] Christoforos Ntantogian, Panagiotis Bountakas, Dimitris Antonaropoulos, Constantinos Patsakis, and Christos Xenakis. Nodexp: Node.js server-side javascript injection vulnerability detection and exploitation. *Journal of Information Security and Applications*, 58:102752, 2021.

[58] Grigoris Ntousakis, Sotiris Ioannidis, and Nikos Vasilakis. Demo: Detecting third-party library problems with combined program analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2429–2431, New York, NY, USA, 2021. Association for Computing Machinery.

[59] Marc Ohm, Arnold Sykosch, and Michael Meier. Towards detection of software supply chain attacks by forensic artifacts. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ARES '20, New York, NY, USA, 2020. Association for Computing Machinery.

[60] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, New York, NY, USA, 2018. Association for Computing Machinery.

[61] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1513–1531, New York, NY, USA, 2020. Association for Computing Machinery.

[62] Donald Pinckney, Federico Cassano, Arjun Guha, Jonathan Bell, Massimiliano Culpo, and Todd Gamblin. Flexible and optimal dependency management via max-smt. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1418–1429. IEEE Press, 2023.

[63] Aanand Prasad, Ben Firshman, Carl Tashian, and Eva Parish. Command line interface guidelines. `https://clig.dev/`, 2023. Accessed: 9/20/2023.

[64] 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*, ARES '22, New York, NY, USA, 2022. Association for Computing Machinery.

[65] 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.

[66] Hamed Taherdoost. How to conduct an effective interview; a guide to interview design in research study. *International Journal of Academic Research in Management*, 11(1):39–51, 2022. Available at SSRN: https://ssrn.com/abstract=4178687.

[67] 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*, ESEC/FSE 2021, page 780–792, New York, NY, USA, 2021. Association for Computing Machinery.

[68] 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*, CCS '20, page 2093–2095, New York, NY, USA, 2020. Association for Computing Machinery.

[69] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45, 2020.

[70] Ying Wang, Peng Sun, Lin Pei, Yue Yu, Chang Xu, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. Plumber: Boosting the propagation of vulnerability fixes in the npm ecosystem. *IEEE Transactions on Software Engineering*, 49(5):3155–3181, 2023.

[71] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. What the fork? finding hidden code clones in npm. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2415–2426, 2022.

[72] Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. Wolf at the door: Preventing install-time attacks in npm with latch. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 1139–1153, New York, NY, USA, 2022. Association for Computing Machinery.

[73] B. Xu, L. An, F. Thung, et al. Why reinventing the wheels? an empirical study on library reuse and re-implementation. *Empirical Software Engineering*, 25:755–789, 2020.

[74] Nusrat Zahan, Parth Kanakiya, Brian Hambleton, Shohanuzzaman Shohan, and Laurie Williams. Preprint: Can the openssf scorecard be used to measure the security posture of npm and pypi? *arXiv preprint arXiv:2208.03412*, 2022.

[75] 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, 2023.

[76] Stan Zajdel, Diego Elias Costa, and Hafedh Mili. Open source software: An approach to controlling usage and risk in application ecosystems. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '22, page 154–163, New York, NY, USA, 2022. Association for Computing Machinery.

[77] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 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, Santa Clara, CA, August 2019. USENIX Association.