Hallvard Molin Morstøl
Sverre Rynning-Tønnesen

AutoTrust: Automatic software package assessment using trust criteria

A study into software supply chain security for the pre-install phase

Master's thesis in Computer Science
Supervisor: Daniela Soares Cruzes
June 2023

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Hallvard Molin Morstøl
Sverre Rynning-Tønnesen

AutoTrust: Automatic software package assessment using trust criteria

A study into software supply chain security for the pre-install phase

Master's thesis in Computer Science
Supervisor: Daniela Soares Cruzes
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Motivation: This thesis is a study into software supply chain security for the pre-install phase. The work was motivated by the recent increase in software supply chain attacks through the use of malicious software packages.

Objective: The stated goal is to develop and evaluate AutoTrust, a CLI tool to improve the pre-install assessment of third-party software packages. We also wanted to consider what information is useful for evaluating packages, the advantages, and disadvantages of such tools, and find out how their risk assessment compares to other ways of assessing software packages.

Methods: The research method used is design science research. We used experiments, document studies, questionnaires, and interviews to gather the data for defining requirements, designing, demonstrating, and evaluating AutoTrust.

Results: We suggested and evaluated several trust criteria and found that the most useful were about deprecation or deprecated dependencies, known vulnerabilities, popularity, widespread use, license, and contributors. The main advantage of pre-install tools was the early detection of risky packages and the disadvantage was that it was hard for the developers to fully understand all the criteria used for the evaluation. From comparing AutoTrust to other tools we found it to be a valuable tool, distinct from existing alternatives, which highlights its usefulness. We also found that security tools should be combined with a manual inspection of packages, and not replace this assessment.

Contribution: The main contribution of this work is AutoTrust. The contribution also includes exploring if users find pre-install tools valuable. The last major contribution is the presentation, use, and evaluation of new trust criteria that can be used in the assessment of software packages.

Limitations: The main limitations of this work are linked to only directly comparing AutoTrust to OpenSSF Scorecard, and only four developers testing AutoTrust and attending an interview.

Conclusion: This work highlights the significance of conducting pre-installation evaluations of software packages with AutoTrust, which proved to be useful for developers. We also found it preferred to combine the use of security tools with manual assessment for optimal results. Additionally, we have identified numerous valuable trust criteria and evaluated their usability and importance.

Keywords: AutoTrust, software, supply chain, security, third-party, packages, dependency, pre-install assessment, OpenSSF Scorecard, NuGet

Sammendrag

Motivasjon: Denne masteroppgaven er en studie i forsyningskjedesikkerhet for programvare. Arbeidet var motivert av økningen i programvareangrep på forsyningskjeden ved bruk av ondsinnede programvarepakker.

Mål: Målet er å utvikle og evaluere AutoTrust, et terminalverktøy for å forbedre vurderingen av tredjeparts programvarepakker før man installerer dem. Vi ønsket også å vurdere hvilken informasjon som er nyttig for pakkeevaluering, fordelene og ulempene med slike terminalverktøy, samt finne ut hvordan AutoTrust sin risikovurderingen er sammenlignet med andre måter å vurdere programvarepakker.

Metode: Forskningsmetoden som ble brukt er "design science research". Vi brukte eksperimenter, dokumentstudier, spørreskjemaer og intervjuer til å samle inn data for å definere krav, designe, demonstrere og evaluere AutoTrust.

Resultater: Vi forelso og evaluerte flere tillitskriterier og fant ut at de mest nyttige handlet om foreldete pakker og avhengigheter, kjente sårbarheter, popularitet, utbredt bruk, lisens og kodebidragsytere. Den største fordelen med verktøy som AutoTrust var tidlig oppdagelse av risikable pakker, og hovedulempen var at det var vanskelig for utviklerne å forstå alle kriteriene som ble brukt i AutoTrust sin evaluering. Vi sammenlignet AutoTrust med andre verktøy, og fant ut at det var et verdifullt verktøy som var forskjellig fra eksisterende alternativer. Vi fant også at sikkerhetsverktøy bør kombineres med en manuell vurdering av pakker.

Bidrag: Hovedbidraget vårt er selve verktøyet, AutoTrust. I tillegg har vi sett på om utviklere finner slike sikkerhetsverktøy nyttige. Et tredje bidrag er funnet av, eksempler på bruk og evaluering av tillittskriterier som kan brukes til å evaluere programvarepakker.

Begrensninger: Hovedbegrensningene i denne oppgaven var at vi kun sammenligner AutoTrust direkte med OpenSSF Scorecard og at vi bare fikk testet det på fire utviklere.

Konklusjon: Denne oppgaven viser hvordan AutoTrust kan brukes til å evaluere programvarepakker før de blir lagt til i et prosjekt og hvor nyttig utviklerne mener verktøyet er. I tillegg fant vi at man bør kombinere sikkerhetsverktøy med manuell evaluering av pakker og har kommet opp med mange tillitskriterier for programvare samt testet dens brukbarhet og viktighet.

Nøkkelord: AutoTrust, programvare, distribusjonskjede, sikkerhet, OpenSSF Scorecard, pre-installasjonsevaluering, tredjeparts programvarepakker, tredjepartskomponenter, NuGet

Acknowledgement

This thesis was conducted at the Department of Computer Science, NTNU, from January 2023 to June 2023, as part of the course TDT4900 - Computer Science, Master's Thesis.

First of all, we would like to express our sincere gratitude and appreciation to Professor Daniela Soares Cruzes for her guidance, support, and expertise throughout the duration of this research project.

This thesis has been made in collaboration with Visma. We would like to thank all the employees that helped answer questions, test the tool, and provide feedback. We would like to send a special thanks to the Head of Security Development at Visma, Dr. Monica Iovan, for giving valuable insight into the requirements for the tool. We would like to extend our heartfelt appreciation to Nicolai Mikkel Brogaard and Joona Hoikkala for their guidance and support during the development of AutoTrust. Their advice and willingness to provide valuable feedback in the early stages of the project have greatly contributed to its success.

We are deeply grateful to our fellow master's students for their positive demeanor and sense of humor which have created a supportive and enjoyable work environment.

Finally, we extend our heartfelt gratitude to our friends and families for their invaluable support.

Contents

Figures

Tables

Acronyms

AI Artificial intelligence. 26, 110, 111

API Application programming interface. 17, 19, 25, 52, 63, 65, 68, 96, 103, 107, 109, 113, 114

CAPEC Common Attack Pattern Enumeration and Classification. 20, 21

CD Continuous deployment. 20, 27, 65, 102, 110, 112

CI Continuous integration. 20, 27, 65, 102, 110, 112

CLI Command-line interface. 2–4, 21, 48, 49, 51, 64, 77, 91, 99, 112, 115

CVE Common Vulnerabilities and Exposures. 20, 21, 113

CVSS Common Vulnerability Scoring System. 20, 21, 113

CWE Common Weakness Enumeration. 20, 21

FR Functional requirement. 49, 50, 64, 65, 91, 92

IDE Integrated development environment. 51, 64, 101, 110

LLM Large language model. 26, 111

NTNU Norwegian University of Science and Technology. iii, 38, 42, 43

OpenSSF Open Source Security Foundation. vii, 3, 24–26, 34–37, 40, 41, 47, 66–68, 72, 77–86, 93, 101–103, 105, 106, 108, 109, 111–114, 116, 117

OSV Open Source Vulnerability Database. vii, 20, 51–53, 56, 64, 68, 104, 113

QA Quality attribute. 49–51, 65, 68, 70, 92, 93, 109, 114

RQ Research question. 4, 77, 93, 95, 99, 101, 105, 110, 115, 116

SLR Systematic literature review. 1, 3, 10, 16, 20, 26, 28, 32, 35, 37, 47–50, 98, 99, 176

TC Trust criteria. vii, 1, 10, 14, 54, 58, 74, 75, 95–99, 107, 108, 110, 116, 147

URL Uniform Resource Locator. 22, 23, 56, 59, 68, 78, 104, 106

WSL Windows Subsystem for Linux. 70, 109

Chapter 1

Introduction

Software development is an integral pillar of our modern society, enabling us to automate processes, analyze data, and communicate with each other on a global scale. However, as software becomes increasingly complex and interconnected it becomes more exposed to attacks from threat actors. Threat actors can target multiple attack vectors like deployed systems, application repositories, networks, build systems, and dependencies, which all are part of what we call a software supply chain [1]. The term software supply chain security refers to the process of securing the entire software development lifecycle. This includes not only the code itself but also the various third-party components that are often used in software development. Third-party security is a critical aspect of software supply chain security, as third-party software is hard to evaluate, and adding this to your projects introduces new security risks. However, in modern software development, it is almost unavoidable to not depend on any third-party software. Consequently, it is critical to assess this software's security before using it.

1.1 Motivation

Software security remains a pressing issue for academia and the industry, and significant ongoing work is focused on securing software. In preparation for the work on this thesis, we conducted a systematic literature review (SLR) that can be found in Appendix C. In the SLR we evaluated 68 research papers about software supply chain security. It presents real-world attacks, attack methods, countermeasures, frameworks, trust criteria (TC), package managers, and security tools discussed in the research papers focusing on the software supply chain. It highlights the complexity of the software supply chain, the multiple attack vectors, and some actual attacks that have successfully targeted the supply chain.

The most prominent supply chain attack in recent times is the SolarWinds Hack that occurred back in 2020 [2–4]. The attackers inserted malicious code into the SolarWinds Orion Platform, and through this, they created a backdoor that the hackers used to access the accounts of the victim organizations. By injecting malicious code into an update or patch distributed by SolarWinds, the hackers were able to compromise the security of numerous organizations in a single stroke. SolarWinds' position as a trusted software provider made it an ideal target for this type of attack, allowing the hackers to bypass traditional security measures and gain access to sensitive information. What made this attack so harmful and concerning is that by attacking the software supply chain they were able to infiltrate approximately 18 000 customer networks [3].

A rising problem for supply chain security is the rising amount of malicious packages being uploaded to various package registries. Uploading malicious packages to package registries like npm, PyPI, Maven, and NuGet makes these packages accessible for all programmers using the registries. The attackers can then trick developers into using their malicious packages as a third-party dependency and infiltrate the program through this new dependency. From a developer's perspective, it can be hard to evaluate the security of packages and there are multiple examples of attacks exploiting this weakness. One example from npm was the politically motivated modification of the package *node-ipc* [5]. The maintainer of *node-ipc* added another package, *peacenotwar*, as a dependency to *node-ipc*. By doing this all users of *node-ipc*, that were located in either Russia or Belarus, would get a file placed in their desktop directory with contents relating to the current war-time situation of Russia and Ukraine [5]. This example was not doing anything destructive but could easily have been modified to do something more malicious.

The problem of malicious third-party dependencies is also present for NuGet packages. According to a recent article published by JFrog, they discovered malicious packages that might have been downloaded more than 150,000 times in the month preceding their removal [6]. The packages would execute a PowerShell script that triggered the download of a 2nd stage payload that could be remotely executed.

These recent attacks are just a few examples highlighting the persistent issue of software supply chain attacks, which does not seem to disappear anytime soon. The SolarWinds Hack showed the consequences of breaching one system or package, and how it can lead to the infiltration of numerous other systems. Mitigating these threats necessitates the collective creation of various security solutions, and we are compelled to participate in this endeavor.

1.2 Contribution

In this thesis, we have followed design science principles for information systems and information technology (IT). The objective of design science in these fields is to produce innovative artifacts such as models, methods, and systems that assist individuals in the creation, utilization, and upkeep of IT solutions [7].

To help limit the problem of supply chain attacks we have created an artifact to help developers analyze NuGet packages before adding them to their projects. The artifact is a command-line interface (CLI) tool that runs as part of the installation process of packages, which we have named AutoTrust. This tool analyzes the risk of NuGet packages based on different trust criteria. Trust criteria are a set of standards or measures used to evaluate the reliability, security, and overall quality of software. The tool can be found at `https://github.com/HallvardMM/AutoTrust`.

Our contribution encompasses more than just creating the AutoTrust tool. It involves an academically conducted user evaluation of AutoTrust which is a software supply chain tool for the pre-install phase. This evaluation serves to assess the advantages and disadvantages of AutoTrust. We also compare it to alternative methods of evaluating NuGet packages, such as other tools and manual reviews. Additionally, this thesis researches which trust criteria that are valuable when evaluating the risk associated with NuGet packages.

During this thesis work, we created and distributed a questionnaire to Visma employees to assess the trust criteria. Furthermore, we conducted extensive testing of the tool on 100 NuGet packages and assessed its performance on Windows, macOS, and Linux, thereby comparing its efficiency with the standard "dotnet add package" command. Moreover, we performed an experiment involving students to compare the tool against manual assessment. In addition, we compared the tool with other tools similar to AutoTrust such as npq, OSSGadget, and OpenSSF Scorecard. We also involved Visma employees by having them test the tool and conducted interviews with them to gather their opinion on the tool.

The work on this thesis has been in collaboration with Visma, a Norwegian software company. The decision to create a tool specifically for the NuGet package registry is partly based on NuGet being widely used by Visma employees. In addition, from the work with the SLR, we discovered that almost no literature focused on NuGet, compared to other package registries. There were also no tools like AutoTrust developed for NuGet that we found.

1.3 Goal and Research Questions

The objective of this thesis is to come up with a solution to help minimize developers' chance of adding malicious third-party software.

Goal | Develop and evaluate a CLI tool to improve the pre-install assessment of third-party software packages.

The specific goal is to develop a tool that can be proven to help developers with assessing the risk of NuGet packages. Choosing to focus on a single programming language ecosystem will help scope the goal so it will be feasible to finish during the given time frame and makes it possible to leverage NuGet-specific information. We also believe creating a tool for NuGet will contribute to insight other developers can use to create similar tools for other programming languages.

RQ-1 What information is useful for the assessment of security in third-party software packages prior to their installation?

The first research question (RQ) deals with finding and evaluating potential trust criteria. Here we wanted to follow an academic approach to find the most valuable trust criteria so that they can be part of the tools assessment.

RQ-2 What are the advantages and disadvantages of the CLI pre-install tool for developers in the process of evaluating third-party software packages?

With the second research question we wanted to analyze the tool based on usefulness and ease of use. We also wanted to find out what the tool succeeds with, and what should be further improved when it comes to the package evaluation.

RQ-3 How does the tool's assessment of third-party software packages' security risk compare to other ways of assessing these packages?

For the third research question we wanted to find out how the assessment given by the tool compares to the assessment given by developers themselves or other similar tools.

1.4 Thesis Structure

The structure of this thesis draws on the methodology of design science research while using the common structure used in masters' theses [8]. This thesis consists of six chapters, of which this is the first. The next chapter, chapter 2, presents the relevant background knowledge used when attempting to solve the problem. It also presents related academic research and other software supply chain security tools. Following that, chapter 3 explains the design science research methodology used to structure and guide this research. Subsequently, chapter 4 provides a comprehensive overview of the design process for AutoTrust, including its development, demonstration, testing, and the results from this. Afterward, in chapter 5, the research addresses the RQs, discusses the results, limitations, and implications for practice and research. The final chapter, chapter 6, concludes this thesis, mentions some other contributions made during this thesis, and proposes some future work.

Chapter 2

Background and Related Work

This chapter explains the concepts that are most relevant to this thesis. Firstly, a quick introduction to software supply chain security is provided. Then follows information about how to secure software dependencies. After that, follows an explanation of the criteria used to decide if a dependency can be trusted. Afterward, a brief analysis is provided of some of the most popular package managers, highlighting the metadata they offer to users. The chapter ends with an assessment of some tools designed for increasing software security and introducing related work.

2.1 Software Supply Chain Security

A supply chain refers to the businesses, organizations, people, activities, information, and resources involved in the production and delivery of a product or service. A physical supply chain encompasses the entire process from procurement of raw materials, production, packaging, distribution, and delivery to the final customer. A software supply chain is a bit different as there is no physical product to deliver, but both types of supply chains rely on multiple parties, as seen in Figure 2.1. There are many definitions of the software supply chain [9], but a good definition is from Du et al. that the "software supply chain is the whole development, release, deployment, and maintenance processes of software from source code to the final software delivering to users" [9].

Software security encompasses the techniques and measures taken to protect software systems and applications so they function correctly despite being at risk from malicious attacks such as unauthorized access, theft, damage, and disruption. This is to ensure the confidentiality, integrity, and availability of sensitive data processed and stored by the software systems. It involves protecting software from various threats such as malware, hacking, denial of service, and other types of cyber attacks. Techniques and practices such as authentication, encryption, firewalls, intrusion detection, and intrusion prevention are currently used by software security practitioners to minimize these threats [10]. The goal of software security is to ensure the safe and reliable operation of software systems and

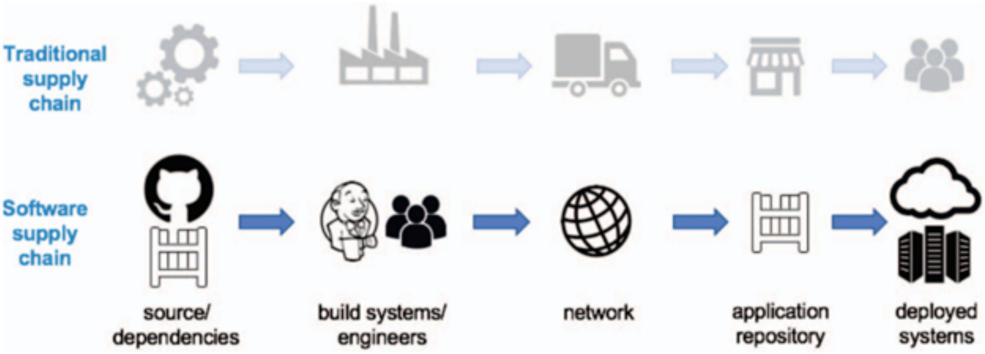


Figure 2.1: Illustration of the similarities between the movement in a software supply chain and traditional supply chain from [1], page 284.

to protect the sensitive information processed and stored by these systems from unauthorized access and malicious attacks.

Software supply chain security is focused on the intersection between software supply chain and software security. It encompasses the software security techniques used to secure the development, release, deployment, and maintenance processes. The focus of software supply chain security is to prevent software supply chain attacks. These attacks "aim at injecting malicious code into software components to compromise downstream users" [11]. Protecting against these attacks is an intricate challenge as multiple fields of study such as supply chain and operations management, security, cryptography, computer science, information systems, telecommunications, e-commerce, insurance, and risk analysis are needed to provide the necessary security [12]. The requirement for knowledge in all these fields arises because attackers only need to find a single weakness, while defenders are responsible for securing the entire attack surface, which in this case extends throughout the entire software supply chain [11].

It can be argued that many of the problems with securing the supply chain of modern-day software are linked to the fact that modern-day software normally is dependent on multiple third-party solutions, known as dependencies. These dependencies can be used as attack vectors to target the primary software, as they are generally created by people outside the project or organization, and it is hard to evaluate the code. One of the focus areas of software supply chain security is therefore how to protect the main application from security risks in the dependencies used.

2.2 Dependency Security

When creating software you are often dependent on software packages written by others. These packages are called dependencies after they are added to the project and an application is normally built on top of multiple other dependencies, as seen in the dependency graph in Figure 2.2. In Figure 2.2, the packages are illustrated as the vertices, and the relationships between the dependencies are illustrated as directed edges. Dependencies 1-3 are examples of direct dependencies, while dependencies 4-6 are examples of transitive dependencies. The main application relies upon both the direct and transitive dependencies to function as expected.

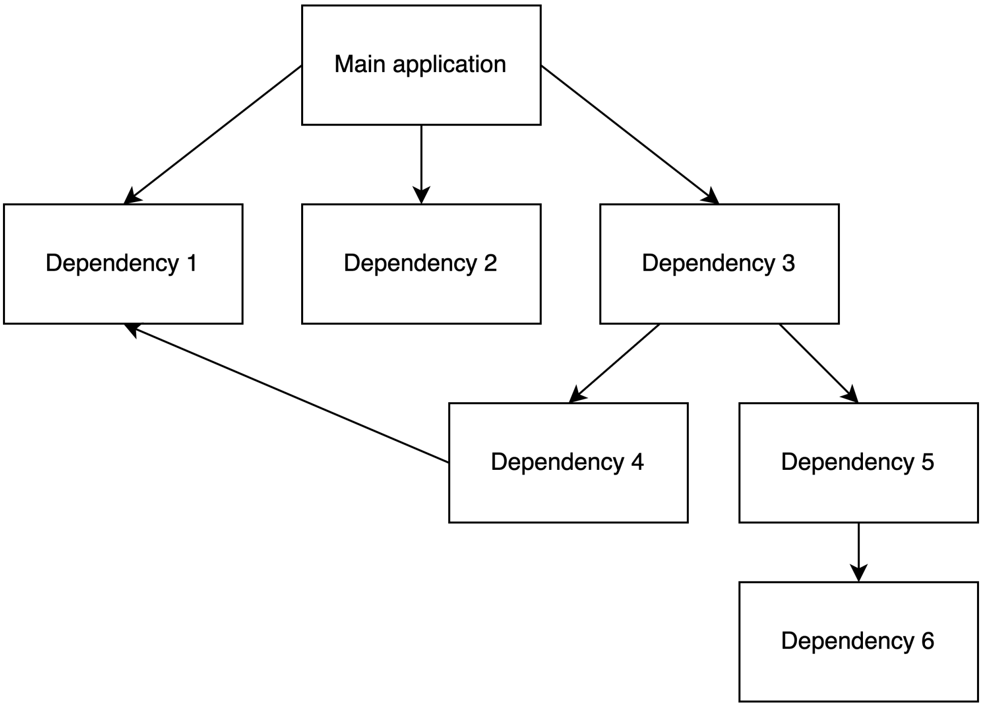


Figure 2.2: Dependency graph showing the direct and transitive dependencies of the main application.

All the different dependencies below the main application can introduce security vulnerabilities and a certain risk. If the security risks are introduced intentionally the package can be categorized as "malware". Another issue is when the security risk is introduced unintentionally due to negligent security practices by the developers of the package. This is sometimes known as "weakware" [13]. The security risk related to a dependency is also affected by the access provided to the source code. If the package maintainers decide to display the source code publicly it is known as "open-source software", and the opposite is known as "closed-source software". It could be argued that when the source code is hidden it is almost impossible for the users to find security risks in the dependency [14].

It is also challenging to find security weaknesses in open-source projects, due to the size and complexity of modern software. This lack of transparency and the challenge of assessing the security of dependencies is the crux of the problem with dependency security. Even if it was possible to do a perfect security assessment of direct dependencies, these dependencies might have multiple dependencies of their own, which again can introduce security weaknesses, as seen with dependencies 4-6 in Figure 2.2. These elongated software supply chains make the task of doing a perfect security assessment practically impossible, as each link in the supply chain introduces additional vulnerabilities and risks that are difficult to identify and mitigate.

The dependency tree is not static as all of the packages and their relations might change due to added functionality or fixes. To keep track of the changes to the packages it is recommended and common to use versioning [10]. Packages versions usually follow semantic versioning using the format X.Y.Z where number X is a major change, number Y is a minor change, and number Z is a patch [15]. Major versions indicate significant changes that might not be backward-compatible, minor versions indicate new features added in a backward-compatible manner, and patch versions indicate bug fixes. X.Y.Z are non-negative integers, and each element increases numerically [15]. In a project, it is therefore also important to keep track of the dependencies' versions. When using a package one usually wants to keep that package version in use up to date in case they have improved the package or fixed security issues. However, new versions might introduce security risks or malicious code [16]. Package users therefore often would like not to be the first nor last to update a dependency. Ideally, they want enough other people to update and test the new version before themselves [17].

It is crucial to understand the risk involved with adding packages and the different forms of dependency attacks. These attacks can range from exploiting known vulnerabilities in the dependencies to more complex supply chain attacks, where malicious actors target upstream dependencies to compromise the entire software supply chain. Software supply chain attacks can be utilized by malicious actors for various purposes, such as data exfiltration, droppers, denial of service, or financial gain [11].

2.2.1 Forms of Dependency Attacks

Ladisa et al. identify three high-level types of attacks that can be used to compromise open source software supply chains. These include developing and advertising a distinct malicious package from scratch, creating name confusion with a legitimate package, and subverting a legitimate package [11]. To mitigate the risks of these attacks, the aforementioned dependencies should be vetted and assessed before including them in the main application.

There are three central phases during the development process where malicious third-party dependencies can compromise the main application and its developers. It is during the installation of the package, when updating the package, and during build time. These phases are shown in Table 2.1 together with specific attack types and variations of those attacks, which are explained below.

Table 2.1: Forms of dependency attacks.

Phase	Attack type	Attack Variations
Installation	Bait attacks	Advertise a malicious package
		Name confusion with a legitimate package (typosquatting)
		Name confusion with a previously legitimate package (brandjack package)
		Authors have introduced malicious code
Update	Subvert a legitimate package	Direct attack
		Influencer attack
Build	Dependency confusion	

During the installation of a package, the developer might be tricked into downloading a malicious package. Bad actors can advertise a malicious package they have created. The advertisement can either be as part of a blog post, as a seemingly helpful comment on a developer forum, or by creating a package that solves a general problem. Another way the developer could be fooled is if the attackers create name confusion with a legitimate package. One example of name confusion is by employing typosquatting techniques, where the attackers create a similarly named package as what the user intended to download with a common misspelling [18]. Another attack similar to typosquatting is when users download a package that has the correct name but is actually a brandjacked package. This occurs when the original package has been removed by its author from the package registry, and an attacker reuses the package's identifier to introduce malicious code under the same name [11, 16]. The user can also download the intended package but the main authors might have introduced malicious code themselves. This can be done for monetary gain or as hacktivism [5]. These forms of attacks are known as bait attacks [19] since they trick the user into installing the malicious package directly.

The attackers can compromise the users by having them update to a new malicious package version. They can subvert a legitimate package and target a package that already has users, contributors, and maintainers by gaining access and making malicious changes before publishing a new version [16]. To distribute a malicious version the attackers can either do a direct attack or an influencer attack. The direct attack is where they compromise the main package either by gaining unauthorized access to the target package or by having malicious code contributions approved by the maintainers [19]. The attackers might also do an influencer attack where they target and tamper with one of the dependencies of the main victim package. They can manipulate existing dependencies, or add new dependencies into the victim package [19].

Attackers could also obtain information about non-public packages a company uses and hosts internally. The attackers can use this information to create public packages with the same name and trick developers and misconfigured build pipelines to fetch the malicious publicly hosted package instead of the internally hosted benign package. This attack, known as dependency confusion [20], differs from the others as it does not necessarily target the initial installation or the update but happens when the attackers publish the package and the build process starts to fetch the public package instead of the private ones. Compared to the other types of attack it does not necessarily need manual input from the victim [21].

2.3 Trust Criteria

Despite the risk associated with third-party software dependencies, they are useful for development. It is, therefore, necessary to identify methods for evaluating which dependencies can be safely utilized and which should be avoided. Using open-source software reduces development time which makes programming more cost-effective by utilizing pre-existing code. It can be used to save costs as fewer hours are spent on re-implementing existing functionalities which again can reduce the time-to-market [22]. Developers need to assess which packages to trust based on available data and time. It is usually not feasible to inspect all packages thoroughly so the developers need to look for indicators of packages with good security. We have decided to name those indicators of trustworthy packages "trust criteria" (TC).

When relying on a large number of dependencies, it becomes impractical for a consumer to conduct a thorough review of each one, and they are usually forced to rely on metadata to evaluate which packages to trust [11]. There are multiple data points developers can use to evaluate packages based on these trust criteria. There has been some prior research on this topic by authors who have used different names for these criteria. Mills and Butakov used the term "evaluation criteria" and focused on criteria that can be used to find secure projects [23]. Zahan et. al focused on indicators of bad or weak packages and called these indicators "weak links" [24]. Both these papers were found while working on last year’s SLR, and they proved to be influential in our pursuit of identifying trust criteria.

The trust criteria in Table 2.2 are a combination of the result from last year’s SLR, a review of additional research papers, and criteria found during the development of AutoTrust. The additional research papers were found by snowballing the most relevant papers from the SLR, and from searching for papers released after the SLR was conducted on Scopus[1], Web of Science[2], and Google Scholar[3]. Trust criterion 30 was added after the development phase, as described in section 4.3. It has been included to ensure that all TC involved in the design of AutoTrust are represented.

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

[2] https://www.webofscience.com/wos/woscc/basic-search

[3] https://scholar.google.com/

Table 2.2: Trust criteria to assess software packages. The evaluate column proposes some ways to analyze data for the trust criteria.

No.	Trust criteria	Data Type	Evaluate	Source
	Time and human relations			
1	The component has been in widespread use for a considerable amount of time.	Numerical	Stable version (1.0.0 or higher). "Considerable time" will be defined by risk appetite.	[10, 14, 23, 25, 26]
2	The component is widely used and popular.	Numerical	The number of downloads. The number of stars, forks, and watchers. The number of other projects depends on the repository.	[10, 23, 27]
3	There is a reasonable time since the latest update was published.	Numerical	Old updates and newly updated packages are associated with risk.	[17]
4	The company you are working for is already using the package in another project.	Categorical/ Numerical	Check if another project in the company is using the package. The number of other projects that are using the package at the company.	[28]
	Licensing and documentation			
5	The component has a software certification from a certified provider.	Categorical	Type of certification.	[10, 14, 26]
6	The component provides a hash and signature that can be used to make sure that the software has not been tampered with.	Categorical	The presence of hash or signature, and it matches the stated values.	[10, 27]
7	The component provides a standard or well-written license.	Categorical	Type of license.	[10, 26]
8	The component has detailed documentation.	Categorical/ Numerical	Presence of detailed documentation. Length of documentation.	[10, 23]

Continued on next page

Table 2.2 – continued from previous page

No.	Trust criteria	Data Type	Evaluate	Source
	Maintainers			
9	The component has an adequate number of maintainers and contributors.	Numerical	Make sure there are more than one maintainer and the total number should match the complexity of the repository.	[10, 23, 24, 29]
10	The component is being developed by an active maintainer domain.	Numerical	Number of maintainers and the frequency of their commits.	[10, 17, 23–25]
11	The maintainers of the component are not overloaded.	Numerical	Make sure the maintainers do not work on too many projects.	[10, 24]
12	The maintainers of the component are using a programming language that they are familiar with.	Categorical	The maintainers have other projects using the same programming language. The team is using good practices and standards for that language.	[10, 25]
13	There are no maintainer accounts associated with an expired email domain.	Categorical	If one of the maintainers uses an email with an expired domain.	[17]
14	The package has not changed ownership recently.	Categorical/ Numerical	If the package has changed owner. If yes evaluate if the change has happened recently.	[19]
15	The maintainers, owners, and suppliers of the component are trustworthy.	Categorical	Information about the identity of the maintainers and owners can be used (they use their real identity, profile picture, email, linked social media and organization, and have a history of co-authoring with other developers). They are using two-factor authentication.	[10, 17, 24, 27]

Continued on next page

Table 2.2 – continued from previous page

No.	Trust criteria	Data Type	Evaluate	Source
Maintenance lifecycle				
16	The component's maintenance lifecycle is up to date.	Categorical/ Numerical	Uses tools for Automated dependency updates. The number of open issues and pending pull requests.	[10, 23, 25, 27]
17	There are a small number of open issues, and they are not very old.	Numerical	Number of open issues. The age of open issues.	[10, 23]
18	The developers of the component are using automated code analysis to review the code.	Categorical	If automated code analysis of tools is being used. The popularity and quality of the tools.	[10, 17, 23]
19	The code reviews in the project are of high quality.	Categorical/ Numerical	The presence of indicators that indicate a good review process of new contributions. All pull requests are reviewed and have comments on them.	[30, 31]
Reported problems and deprecation				
20	The project does not have reported security vulnerabilities.	Categorical	No open security issues. No publicly disclosed vulnerabilities on vulnerability databases.	[32, 33]
21	There is no history of prior harmful effects associated with the component.	Numerical	Number of old incidents disclosed in vulnerability databases.	[32, 33]
22	The component is not deprecated.	Categorical	Repository is not deprecated.	[10, 17, 23, 24]
23	The component does not depend on deprecated packages.	Categorical	Has one or more direct or transitive dependency to deprecated packages.	[10, 17, 23, 24]
Other				
24	The size of the repository.	Numerical	The size of the project in bytes or files.	[30, 34]

Continued on next page

Table 2.2 – continued from previous page

No.	Trust criteria	Data Type	Evaluate	Source
25	The project has few direct and transitive dependencies.	Numerical	The number of direct and transitive dependencies.	[34, 35]
26	The component does not contain installation scripts.	Categorical	Presence of installation scripts.	[10, 24]
27	The component has a name that does not resemble that of a popular package.	Numerical	There is no typo in the package name. The package does not use typical typosquatting naming techniques.	[18, 19, 36]
28	There is no difference between the source code and the package.	Categorical	There is no difference between the available source code and the packaged code.	[37]
29	The source code is possible to access.	Categorical	No restrictions on access to the source code.	[14, 24]
30	The package has some verification from the package provider that makes it less deceptive in its identifying properties.	Categorical	The registry maintainers allow package owners to provide some extra identification to verify	[38]

The trust criteria in Table 2.2 is divided into two groups: using numerical data and using categorical data. Numerical data is countable data such as the aforementioned number of downloads or the number of GitHub stars, while categorical data can only be divided into predefined or fixed values. If a project contains a license, the license used is an example of a categorical trust criterion as there is a predefined number of different licenses. When considering TC using numerical data, such as the number of downloads, developers need to assess what they consider to be appropriate thresholds. If a package has two downloads during the last week, it might not contribute to increased trust, but if the package has hundreds of thousands of weekly downloads this might be used as an argument to trust the package. Low numbers in numerical data might actually be used to flag potential issues, an example would be that it is likely that a malicious package using typosquatting has way fewer downloads than its benign counterpart package. Some criteria could also be either categorical or numerical. If one uses the existence of good documentation as a trust criterion then the presence of a README.md file could be a categorical indicator of documentation, but the number of code lines in the README.md file could be used as a numerical indicator.

Some of the trust criteria can be easier to evaluate due to the availability of the metadata such as the number of downloads or GitHub stars. Both the number of downloads and the number of GitHub stars are popularity indicators that show if other developers trust the same package. Other criteria such as discrepancies between the source code and package might be harder to evaluate since the whole code base needs to be evaluated. Looking at the differences between source code and packages might be an indicator that someone has tampered with the code during the build process [37]. However, the easiness of evaluation in itself cannot be used as an indicator for security, and this is therefore not used as a factor when finding the trust criteria.

There are multiple data sources to evaluate when assessing the quality of third-party dependencies and if one should trust them. Some of the criteria in Table 2.2 are easier to evaluate than others and none of these criteria can perfectly differentiate weakware and malicious packages from the rest. Which trust criteria we ended up implementing in AutoTrust are presented in section 4.3.

2.4 Code Hosting Platforms

To be able to do an evaluation of third-party dependencies, the information needed has to be open and accessible for the users intending to use the package. Some of this information about dependencies is provided by code hosting platforms, such as popularity metrics and commit history. A code hosting platform is like a web-hosted file archive for the software code, documentation, and web pages that can be accessed either publicly or privately.

Three of the largest code hosting platforms in 2023 are GitHub, GitLab, and Bitbucket. They are all used to host Git projects [39]. In 2017 Alamer and Alyahya did a large assessment of multiple open-source code hosting platforms, and they found few differences between the three mentioned platforms regarding functionality [40]. Out of 59 features they assessed, there were 8 features where GitHub, GitLab, and Bitbucket scored differently from each other. The differences that are still relevant as of February 2023 are that GitLab does not support "Show trending developers (popular developers)", and Bitbucket does not support "Starring projects/repositories" [41], and "Graph summarizing contributions and activities". If a developer is on the list of trending developers it could contribute to increased trust in that developer and the projects they maintain. As mentioned the number of stars can be used as a popularity metric. A developer with lots of contributions and activities would have increased trust as it shows that they are dedicated to their work. Overall, the differences are minor between the platforms, and developers should still be able to use the information provided by the code hosting platforms to assess the third-party dependencies hosted on them, no matter which one is used.

2.5 Package Managers and Package Registries

To bring third-party software into a project it is common to use package managers and package registries. Both the package managers and registries can be sources of package metadata, used to evaluate trust criteria. Package managers are software tools used to aid in the installation and maintenance of software packages for operating systems or programming languages [19, 42]. The package managers ensure that the necessary dependencies are installed from package registries and that compatibility issues between packages are resolved.

The packages provided by the package managers are hosted on package registries. The package registries also include information about the package metadata and the configuration needed to install them [24]. Many modern programming languages have a central package registry that hosts most of the packages for that programming language ecosystem, such as npm, PyPI, Maven, NuGet, etc.

Based on our previous SLR the package registries that got the most attention in research papers about software supply chain security were npm, Maven Central, and PyPI [10]. We also evaluate NuGet as this package registry is what we are focusing on in this thesis. To get a better understanding of what kind of information is accessible to developers wanting to download packages, an assessment of npm[4], PyPI[5], Maven Central[6], and NuGet[7] was performed. The results are shown in Table 2.3.

Table 2.3: Evaluation of package information in selected package registries.

Information	npm	PyPI	Maven	NuGet
Package Info				
Package Name	●	●	●	●
Package Version	●	●	●	●
License Type	●	●	●	●
Install Command	●	●	●	●
README Information	●	●	●	●
Link to Repository	●	●	●	●
Link to Project Page	●	●	●	●
Package Size	●		●	●
Total Files	●			
Package Maintainers	●	●	●	●

Continued on next page

[4] https://www.npmjs.com/

[5] https://pypi.org/

[6] https://central.sonatype.com/

[7] https://www.nuget.org/

Table 2.3 – continued from previous page

Information	npm	PyPI	Maven	NuGet
Version Information				
Age of Latest Release	●	●	●	●
Age of Version	●	●	●	●
List of Versions	●	●	●	●
Popularity Metrics				
Number of Downloads	●			●
Number of GitHub Stars		●		
Package Maintenance				
Number of Open Issues	●	●		
Number of Open Pull Requests	●	●		
Package Classifiers				
Keywords/Categories	●		●	●
Environment		●		●
Framework		●		
Intended Audience		●		
Supported Programming Language Versions	●	●	●	●
Operating System		●		
Topic		●		
Package Inspection				
Test Platform	●			●
Code	●			
API Diff Explorer				●
Dependencies				
Direct Dependencies	●		●	●
Direct Dependents	●		●	●
Deprecation				
Deprecated Package Information	●	●		●
Deprecated Version Information	●	●		●
Security Information				
Security Holding Package	●			●
Warning If Not on Latest Release		●		●
Security Evaluation			●	

Continued on next page

Table 2.3 – continued from previous page

Information	npm	PyPI	Maven	NuGet
Reserved Prefix Information				•

From the results in Table 2.3 it is clear that all the package registries showed what might be considered essential information like name, version information, and license. All the package registries also allowed the package maintainers to include general information, often in the form of a README, which is a text file containing information about the repository. It is helpful for users that package maintainers can include information they consider relevant, however, some of the information should be standard. Three of the package registries showed direct dependencies, but none of the registries showed transitive dependencies. Not showcasing the chain of dependencies might increase the difficulty in locating vulnerabilities or malicious dependencies [35].

2.5.1 npm

npm is a registry used by JavaScript programmers and it provides most of the same information as the other registries. One benefit of npm is that it shows the code on the project page, so developers can inspect it before installing. Another benefit of npm is that it fetches some information from GitHub such as open issues and pull requests. However, it does not provide information about the number of GitHub stars as a popularity metric. In addition, npm should show a warning if the user is not browsing the latest release like PyPI and NuGet do.

One of the differences between npm and other registries is how they handle malicious packages. If an npm package is found to be malicious, the npm team creates a placeholder in the package registry called a "Security holding package" with some information about the incident. The information on npm is, however, less descriptive if only some versions are malicious[8]. One example of such a security incident is with the UA-parser-js npm package[9], where three harmful versions (0.7.29, 0.8.0, and 1.0.0) were removed due to them containing malicious code. The incident occurred because the attackers hijacked the author's npm account. Completely removing the versions from the registry, instead of unlisting them and blocking downloads, can make it harder to understand why versions are skipped and hide the history of the package.

2.5.2 PyPI

PyPI is the central package registry for Python. It has more information about package classifiers such as environment, framework, intended audience, and op-

[8] https://github.com/npm/documentation/issues/438

[9] https://github.com/faisalman/ua-parser-js/issues/536

erating system which the other registries do not. This makes it easier to search for packages and filter out the ones that are not relevant.

PyPi does not provide the number of downloads but uses GitHub stars as a popularity metric. It might be more challenging for users to evaluate the popularity of a package based on the number of GitHub stars instead of the number of downloads. When doing the comparison of the registries PyPI displayed the information about open issues and open pull requests combined, which made it harder to assess those metrics (this is now changed, see section 6.1). As of February 2023, there is no ability to deprecate a package in PyPI[10], but one can set the development status as "Inactive environment" to indicate that a package was deprecated.

2.5.3 Maven Central

Maven Central is the package registry for Java projects. It is the registry that includes the least package metadata, but it was the only registry that directly provides a security evaluation of the packages, by using Sonatype Safety Rating, BOM Doctor Report, and OSS Index. The Maven Central registry did not provide any popularity metrics, but other websites did this[11].

Some things could be improved with the Maven Central registry. For instance, the package *maven-compiler-plugin@3.9.0* was published by two different authors. One of these versions was benign[12] and the other was reported malicious[13] [43]. This malicious version is still displayed without any warnings on the Maven Central registry, even though it is not possible to download it. All the contributors of the original package are displayed on the malicious package which boosts the credibility of the package, which also is an issue.

2.5.4 NuGet

NuGet is primarily used for programming languages that run on the Microsoft .NET platform, including C#, Visual Basic, and F#. It is one of the package registries that provided the most information. The main thing that stood out from the rest is that NuGet links to a website where users can evaluate the application programming interface (API) differences between versions[14]. Both npm and PyPI include information from GitHub, but NuGet does not provide any of this Github information, such as the "Number of GitHub stars", "Number of open issues", or "Number of open pull requests". The fact that NuGet does not have integration

[10] https://github.com/pypi/warehouse/issues/345

[11] https://mvnrepository.com/

[12] https://search.maven.org/artifact/org.apache.maven.plugins/maven-compiler-plugin/3.9.0/maven-plugin

[13] https://search.maven.org/artifact/com.github.codingandcoding/maven-compiler-plugin/3.9.0/jar

[14] https://www.fuget.org/

with GitHub is peculiar since Microsoft is the founder of .NET Foundation, which are the developers for NuGet, and is the parent company to GitHub [44, 45].

2.6 Security Tools and other Solutions

There are other sources than the code hosting platforms and package managers where users can get information about software packages. Several tools and databases exist that aim to give users a security evaluation of software packages. The tools might target different parts of the supply chain and many of the security tools are used as part of the continuous integration/continuous development (CI/CD) or are regularly scanning the project's dependencies. In the SLR we found 24 different security tools mentioned in academic papers such as tools made by the companies Snyk, SonaType, Mend, OpenSSF, Veracode, and Black Duck [10].

2.6.1 Security Databases

Many of these companies provide databases with security assessment of open source packages such as Snyk vulnerability DB[15], Sonatype OSS Index[16], Mend Vulnerability Database[17], OpenSSF Security Metrics Project[18], Veracode Vulnerability Database[19]. Some other vulnerability databases that were not mentioned in the SLR are Open Source Vulnerability Database (OSV)[20] and GitHub Advisory Database[21]. These databases inform about both packages with known security vulnerabilities and identified malicious packages. Even though these databases are valuable, one downside is that they cannot display information before the vulnerability is discovered. Some tools, therefore, regularly scan a project's dependencies and query security databases to see if some of the dependencies have any newly reported issues. If the tool finds an issue it warns the developer with a notification. A tool that uses this technique is Dependabot by GitHub[22]. The reactive nature of such tools is a downside of these solutions as too many notifications might lead the developers to notification fatigue and make the developers slow to update the dependencies [33].

2.6.2 CVE, CWE, CVSS, and CAPEC

The four identifications, CVE, CWE, CVSS, and CAPEC, are used to help distinguish and compare different cybersecurity events in the security databases. Com-

[15] https://security.snyk.io/

[16] https://ossindex.sonatype.org/

[17] https://www.mend.io/vulnerability-database/

[18] https://metrics.openssf.org/

[19] https://sca.analysiscenter.veracode.com/vulnerability-database

[20] https://osv.dev/

[21] https://github.com/advisories

[22] https://github.com/dependabot

mon Vulnerabilities and Exposures (CVE)[23] is used to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities. The CVE record contains a consistent description of the vulnerability, CVE ID, and references.

Common Weakness Enumeration (CWE)[24] is a standard for classifying and describing the types of software and hardware weakness types that can lead to vulnerabilities. Whereas CVE describes, identifies, and names specific vulnerabilities, CWE describes the underlying weaknesses that can lead to vulnerabilities.

Common Vulnerability Scoring System (CVSS)[25] enables the creation of a numerical score that reflects the severity of a vulnerability's main features. This score can then be translated into a qualitative representation such as low, medium, high, and critical or as a number. To create the score different factors are evaluated such as the attack vector, the attack complexity, what privileges are required, if a user other than the attacker needs to participate for a successful compromise to occur, the scope of the affected components, and if the attack affects confidentiality, integrity, or availability.

Common Attack Pattern Enumeration and Classification (CAPEC)[26] contains known patterns of attacks employed by adversaries to exploit weaknesses in the cyber domain. These attack patterns are design patterns used for destructive rather than constructive intentions and are generated from analysis of real-world events.

The identification provided by CVE, CWE, CVSS, and CAPEC makes it possible to distinguish and discuss cybersecurity events. The cyber security databases might reference the same event and these standards and identifications help to confirm that the databases mention the same issues.

2.6.3 Pre-install and Pre-update Tools

There are some existing security tools that aim to make it easier for users to assess a package before installing or updating it. These tools try to fetch relevant information before installation, download, or updates to help developers determine if they want to include a third-party package in their system. They might fetch data from code hosting platforms, package registries, or security databases to give the users insight into the package they want to use in their project.

npq

One such tool made for the JavaScript environment is npq[27]. It aids developers in auditing npm packages before installation. It fetches and displays some package information in the CLI and then prompts the user if they still would like to install the package [46]. It checks a set of evaluation criteria called "marshalls". The marshalls that npq checks are:

[23] https://cve.org/

[24] https://cwe.mitre.org/

[25] https://www.first.org/cvss/

[26] https://capec.mitre.org/

[27] https://github.com/lirantal/npq

- **Age:** Checks npm registry if the package age is less than 22 days.
- **Author:** Checks npm registry if the latest version of the package does not have an author field.
- **Downloads:** Checks npm registry if the packages download count in the last month is less than 20.
- **README:** Checks npm registry if the package does not have a README file, or the README is a security placeholder package by the npm staff.
- **Repository:** Checks npm registry if the latest package version has been found without a correct and working repository URL.
- **Scripts:** Checks npm registry if the package has a pre/post-install script.
- **License:** Checks the latest version if it has a license.
- **Expired domains:** Checks npm registry for maintainers' email addresses and looks up the email domain to verify that it is not expired.
- **Snyk:** Checks Snyk's database if the package has reported vulnerabilities.

If the user uses the alias command it is also possible to make sure that the npq checks are run every time the user wishes to install or update npm packages.

OSSGadgets

Microsoft is working on a collection of tools to download and evaluate open-source packages called OSSGadgets[28]. As of February 2023, OSSGadgets are still in public preview and are not ready for production use. The tools are written in C#, but support auditing packages from multiple sources such as Cargo, Cocoapods, Composer, CPAN, CRAN, Go, Hackage, Maven, npm, NuGet, RubyGems, PyPI, Ubuntu, and Visual Studio Marketplace. OSSGadgets differs from npq and AutoTrust as it is more intended to do low-level tasks like basic analysis of dependencies and estimating its health, instead of being part of a developer's daily work. OSSGadgets consists of 12 tools [47]:

- **OSS-characteristics** uses Microsoft's ApplicationInspector[29] to report patterns to help assess what the package does. It then reports the package's characteristics and features based on what it found using ApplicationInspector's over 400 rule patterns [48].
- **OSS-defog** examines the contents of a software package for an obfuscated text that is either Base-64- or Hex-encoded. It searches through the files and reports matches of Base-64- or Hex-encoded text longer than 8 characters.
- **OSS-detect-backdoor** attempts to identify potential backdoors and malicious code in a package's code. The tool reports the patterns it finds suspicious with filename, severity, and ordered by the confidence that it found a backdoor or malicious code. However, the authors report that it has a high false-positive rate.
- **OSS-detect-cryptography** identifies cryptographic implementations within

[28] https://github.com/microsoft/OSSGadget

[29] https://github.com/Microsoft/ApplicationInspector

a package. It does this by using an embedded set of rules[30] or a custom rule set from the user.

- **OSS-diff** compares two packages and presents the differences between them. It uses the DiffPlex package[31]. It provides insights into the changes made to a project over time, enabling users to understand the impact of those changes and make informed decisions about their use of the software.
- **OSS-download** is used for downloading and optionally extracting open-source packages from various sources, such as code hosting repositories and package registries without needing specific ecosystem-specific tools [49].
- **OSS-find-source** attempts to locate the source code from GitHub of a given package. It searches the package metadata for GitHub URLs and outputs that list of URLs.
- **OSS-find-squats** tries to find typosquatting for a given package. It looks for bitflips; homoglyph attacks; adding, removing, doubling, or swapping letters; prefixes or suffixes added; and common typos on the keyboard. The algorithm is currently focused on the package name, but they are working on improvements to find typo squats such as using information about changes in metadata, the lack of metadata, low usage, and diff[32]. It is also available as a separate NuGet package[33].
- **OSS-find-domain-squats** is similar to OSS-find-squats, but it tries to identify potential typosquatting for a given domain name such as "microsoft.com".
- **OSS-health** calculates the likelihood that a package will continue to meet stakeholder expectations in the future using data from GitHub. It attempts to evaluate if the project will continue to address bugs, release new features, if the community is active, if the information flow between maintainers is adequate, and if security issues are addressed promptly. It is using information such as size, issues, pull requests, contributors, subscribers, forks, stars, releases, and issues using security keywords. Based on this information it uses an algorithm to calculate a health score. The authors believe that the algorithm can be improved.
- **OSS-metadata** retrieves metadata from native package registries, deps.dev, or libraries.io for a given package and normalizes the metadata about the package into a common schema.
- **OSS-risk-calculator** uses OSS-Health and OSS-Characteristics to calculate a risk level in a range from 0 (no risk) to 1 (very high risk) for the risk of using a package. The user can decide to not use the health data by setting the *–no-health* command line option. The algorithm checks if the package has any characteristics matching "cryptography", "authentication", "authorization", or "data deserialization". Of the health data, the algorithm puts the most emphasis on how the project handles security issues and the least on recent

[30] `https://tinyurl.com/OSSGadget-CryptographyRules`

[31] `https://github.com/mmanela/diffplex`

[32] `https://github.com/microsoft/OSSGadget/issues/226`

[33] `https://www.nuget.org/packages/Microsoft.CST.OSSGadget.FindSquats`

activity. The maintainers believe that the algorithm can be improved.

Out of these 12 tools, the OSS-risk-calculator is the tool that shares the most in common with the other pre-install and pre-update tools. However, the way that the OSS-risk-calculator operates is different as it is not integrated with the installation process and does not give clear information to the user about what it evaluated and why it gave the security score.

OpenSSF Scorecard

Open Source Security Foundation (OpenSSF) has made a tool named OpenSSF Scorecard[34] "to help open source maintainers improve their security best practices and to help open source consumers judge whether their dependencies are safe" [50].

It works by running 18 checks on GitHub repositories to assess the security risk of the project. OpenSSF are also working on adding support for GitLab and have implemented 13 of the 18 checks for GitLab repositories. The 18 checks used in the OpenSSF Scorecard as well as the description from the project's README file can be seen in Table 2.4 [50]. They have also added the option to use the tool with the package managers npm, PyPi, and RubyGems.

Table 2.4: The OSSF Scorecard checks [50].

Name	Description	Risk Level
Binary-Artifacts	Is the project free of checked-in binaries?	High
Branch-Protection	Does the project use Branch Protection?	High
CI-Tests	Does the project run tests in CI, e.g. GitHub Actions, Prow?	Low
CII-Best-Practices	Has the project earned an OpenSSF (formerly CII) Best Practices Badge at the passing, silver, or gold level?	Low
Code-Review	Does the project practice code review before code is merged?	High
Contributors	Does the project have contributors from at least two different organizations?	Low
Dangerous-Workflow	Does the project avoid dangerous coding patterns in GitHub Action workflows?	Critical
Dependency-Update-Tool	Does the project use tools to help update its dependencies?	High

Continued on next page

[34] https://github.com/ossf/scorecard

Table 2.4 – continued from previous page

Name	Description	Risk Level
Fuzzing	Does the project use fuzzing tools, e.g. OSS-Fuzz?	Medium
License	Does the project declare a license?	Low
Maintained	Is the project at least 90 days old, and maintained?	High
Pinned-Dependencies	Does the project declare and pin dependencies?	Medium
Packaging	Does the project build and publish official packages from CI/CD, e.g. GitHub Publishing?	Medium
SAST	Does the project use static code analysis tools, e.g. CodeQL, LGTM (deprecated), SonarCloud?	Medium
Security-Policy	Does the project contain a security policy?	Medium
Signed-Releases	Does the project cryptographically sign releases?	High
Token-Permissions	Does the project declare GitHub workflow tokens as read only?	High
Vulnerabilities	Does the project have unfixed vulnerabilities? Uses the OSV service.	High

OpenSSF Scorecard calculates a score between 0 and 10, where 10 represents the best possible score for each of the checks in Table 2.4. Similar to the OSS-risk-calculator it gives a score for the whole project. The aggregated score produced by OpenSSF Scorecard is calculated as a weighted average of the individual checks, taking into account their respective risk level. The risk levels presented in Table 2.4 are assigned weights as follows: "Critical" with a weight of 10, "High" with a weight of 7.5, "Medium" with a weight of 5, and "Low" with a weight of 2.5.

OpenSSF Scorecard and npq differ in that OpenSSF Scorecard is mainly using data from the code hosting platforms while npq is mainly using data from the npm package registry. The process of npq is that it extends the "npm install"-command while OpenSSF Scorecard is made as a separate tool. OpenSSF Scorecard is most similar to the OSS-risk-calculator as they both use GitHub metrics to assess the security score of the project, but the checks of OpenSSF Scorecard are more thorough. However, OSS-risk-calculator considers the characteristics of the package which OpenSSF Scorecard does not. Another difference is that OpenSSF Scorecard has an API where one can query pre-calculated scores of open source projects. A more in-depth comparison of npq, OSSGadget, OpenSSF Scorecard, and AutoTrust will be presented in section 4.3.8.

2.6.4 ChatGPT-driven Software Supply Chain Security Tools

During the last year, it has been a massive interest in large language models (LLMs) driven by the popularity of the artificial intelligence (AI) chatbot ChatGPT[35]. This chatbot has proven useful for solving multiple different tasks, and some of these tasks were not even intended by the developers. As a result, many people try to find new ways of leveraging this new technology. As of April 2023, we have found two examples of software supply chain security tools made for evaluating third-party packages that use ChatGPT: Socket AI[36] and DroidGPT[37].

Socket AI works a bit differently compared to npq, OSSGadgets, and OpenSSF Scorecard as it provides security feedback on pull requests, which would help prevent unwanted packages to be part of the source code repository but does not prevent the developer from installing something malicious [51]. It can evaluate npm and PyPI packages by analyzing the package's source code using ChatGPT to find vulnerabilities. The creators say that Socket AI is still struggling with highly obfuscated code, cross-file analysis, and prompt injection specifically targeting AI systems [51].

DroidGPT is still in private beta. It allows users to research open-source software packages in a conversational manner for Cargo, Go, Maven, npm, and PyPI [52]. It combines ChatGPT with Endor Labs' proprietary risk data. DroidGPT uses ChatGPT to create an improved search experience where users can ask for recommendations or more open-ended questions than what a traditional search bar on a package registry would allow. This differs from how Socket AI uses ChatGPT since they use ChatGPT to scan open-source code for vulnerabilities.

2.7 Related Work

There has been conducted some related research that focuses on software supply chain security. We analyzed research publications during our SLR that can be found in Appendix C. The papers we found to be most influential to our thesis are presented here.

Mills and Butakov presented in their paper a list of nine evaluation criteria and thresholds that can be used to evaluate open-source libraries [23]. They looked at "Age of the project", "Code releases", "Project freshness", "Contributor strength", "Documentation", "Popularity", "Open issues", "Repository state", and "Automated code analysis". Mills and Butakov then used the evaluation criteria to test 14 packages for iOS local authentication. None of the tested packages passed all the evaluation criteria. One of their criteria was to evaluate if automated testing in GitHub

[35] https://openai.com/blog/chatgpt

[36] https://socket.dev/blog/introducing-socket-ai-chatgpt-powered-threat-analysis

[37] https://www.endorlabs.com/droidgpt

was used by the project, but none of the select projects used a programming language supported by GitHub's automated tests. The work shows the difficulty of finding correct trust criteria and thresholds, especially if they are to be used to approve or reject a package. The evaluation criteria they presented have inspired some of the trust criteria in Table 2.2.

Ferreira et al. proposed a solution where developers could restrict the access packages had in an application for npm [33]. Instead of trusting a package fully, they had a set of permissions that could be approved to restrict the access the packages got to the network, filesystem, and operating-system processes. This idea would work similarly to how mobile phones allow users to restrict the permissions applications get.

Zahan et al. presented six signs that a package exposes to a higher risk of a supply chain attack, named weak links [24]. The weak links they proposed were "Expired maintainer domain", "Installation script", "Unmaintained package", "Too many maintainers", "Too many contributors", and "Overloaded maintainer". They then analyzed 1,494,105 npm packages to assess how many packages had weak links. They also surveyed 41 of the top 10% of npm package maintainers if they agreed with the weak links. The maintainers agreed that the weak links "Expired maintainer domain", "Installation script", and "Unmaintained package" were relevant but did not support the remaining three. Through the survey, they also received seven new trust criteria which they present but did not evaluate further. The suggested trust criteria were: "Ownership transfer or adding new maintainers", "Maintainer identity", "Maintainer two-factor authentication", "No source code repository", "npm package vs source code repository", "CI/CD pipeline", and "Open pull request". The weak links were used to create some of the trust criteria in Table 2.2.

Ohm et al. analyzed 174 malicious packages from npm, PyPI, and RubyGems [16]. They reviewed those packages and found that: on average a malicious package is available for 209 days; most malicious packages start the attack on installation; 41% of the packages check for a condition before triggering further execution, such as checking for application state, the dependency tree, or operating system; most malicious packages mimic existing packages' names via typosquatting; most packages aim at data exfiltration; most packages do not rely on OS-specific functions; and nearly half of the packages use obfuscation techniques [16].

Vaidya et al. presented a systematic study of security issues that affect language-based ecosystems and focused on npm and PyPI [19]. In the study, the authors assessed the structure of these ecosystems by evaluating information about downloads and dependencies. In addition, they also did a further exploration of the supply chain attack method typosquatting in npm and import-squatting in PyPI. They also proposed a set of guidelines to contain future supply chain attacks and found that it is hard to disambiguate benign and malicious packages. The authors argued that providing developers decision support tools could prove effective in stopping supply chain attacks, and emphasized the importance of tools and metrics designed to help developers assess the risk associated with incorporating external dependencies [19].

In their work, Ladisa et al. proposed a general taxonomy for attacks on open source supply chains, independent of specific programming languages and ecosystems [11]. The taxonomy encompasses all stages of the supply chain, from code contributions to package distribution. Their research resulted in the identification of 107 distinct attack vectors on open source software supply chains, which were linked to 94 real-world incidents and mapped to 33 mitigating safeguards. Additionally, the authors surveyed 17 domain experts and 134 software developers to gather feedback on the taxonomy and evaluate the utility and costs of the identified safeguards against software supply chain attacks.

2.7.1 Gap in Research and Practice

Based on the evaluation of the papers there seem to be many different evaluation criteria proposed. Some of them are focused on positive traits such as the popularity and presence of automated code analysis while others are focused on negative traits such as the presence of installation scripts and maintainers using expired email domains. To the best of our knowledge, there appears to be a limited number of studies that have conducted a comprehensive meta-analysis of various trust criteria proposed in the literature. Are the criteria possible to assess based on the given data? Are the criteria insightful for developers? Do they help distinguish malicious packages from benign packages? In the SLR we saw that four package registries got the most focus: npm, Maven, RubyGems, and PyPI [10]. Would the evaluation criteria and their thresholds differ if other package registries were evaluated such as Crates for Rust or NuGet for C#?

Many of the criteria are also reliant on thresholds such as the number of downloads. It might not be possible to find thresholds that will work the same for all package registries, but proposing and testing thresholds on a larger scale for different trust criteria is also lacking in the research literature. Most of the work proposes some set threshold and tests the criteria based on this threshold, and without having done a proper check of the threshold you might have criteria that are valuable but not fully utilized because of suboptimal thresholds. However, set thresholds might not be optimal since attackers might inflate metrics such as downloads, GitHub stars, pull requests, and maintainer numbers to make the package pass the set threshold [6, 19].

Most of the papers evaluate trust criteria by testing a set number of packages to see if they pass or fail the evaluation metrics of the trust criteria [16, 23, 24], or by interviewing experts and see if they agree with the assessments [24]. There seems to be a gap in the research literature to see if developers can use the trust criteria and make evaluations of software packages. Are developers able to use the information to distinguish high- and low-risk packages? Will automated tests showcasing trust criteria and thresholds lead to notification fatigue? Will such tools make the developers lazy, and lead to more harm than good, such as was the case with password policies [53, 54]? Is the cost measured in time and labor worth more than the extra security provided by the evaluation criteria [55]?

The gap in practice is often linked to the problem that there is no clear metadata and information that the developers can use to assess if the package can be trusted. A recommendation from the literature is that third-party software providers should include a software bill of materials [17, 27, 56]. A software bill of materials is a list of all dependencies used to create the software, but many software authors, unfortunately, do not provide this. The code hosting platforms and package registries provide different information which makes it hard for developers to agree upon one common standard for what should be included as part of due diligence before adding third-party software. Many developers are also probably not aware of the risks associated with third-party software or do not have enough time or resources to do a proper evaluation of the software packages.

We believe that the popularity of tools such as Github's Dependabot shows that developers are interested in finding solutions that can assist with evaluating third-party software packages without taking up to much of the developers' time. As discussed in subsection 2.6.3, there are not many tools that help during the pre-install and pre-update phase, and such tools might be accepted by developers if they do not significantly disrupt the developers' workflow.

Chapter 3

Research Methodology

3.1 Research Methodology

This chapter describes the methodology used for this research project. Design science is the research paradigm we used to ensure that a scientific approach is followed during the development and validation of the artifact. Interviews, questionnaires, experiments, and document analysis were used to generate qualitative and quantitative data.

3.1.1 Design Science

Design science is a research paradigm focusing on the development and validation of an artifact. A design science project consists of an object of study and its two major activities. The study's object is the artifact, and its two main tasks are to design and investigate this artifact in context [7, 57]. The artifacts are developed and used by people with the goal of solving practical problems in their correct environment [7]. There are four main types of artifacts: constructs, models, methods, and instantiations [7]:

- **Constructs** are concepts, definitions, terms, and notations, used to describe and formulate problems and their solutions.
- **Models** are used to represent potential solutions to practical problems, and can therefore be used during the development of other artifacts.
- **Methods** are either formalized or informal prescriptive knowledge which defines processes or are guidelines for how to achieve goals or solve problems. These methods can be detailed and explicitly defined such as algorithms or be more vaguely described like best practices or rules of thumb.
- **Instantiations** are working systems that can be used in practice. They can be an instance of another artifact, where the other artifact works as an idea for the working system.

Constructs are too small to create working systems, and even though it could be argued that methods could be used in some cases to create instantiations it is usually models that are used for this [7]. This thesis used the design science methodology during the research, to ensure that a scientific approach was followed during the creation and evaluation of the artifact. One of the main contributions of this thesis is AutoTrust, which is an instantiation artifact.

3.1.2 The Method Framework Used for the Design Science Research

This thesis follows the method framework presented by Johannesson and Perjons [7]. The activities that are part of the framework are structured into five phases: "Explicate problem", "Define requirements", "Design and develop artifact", "Demonstrate artifact", and "Evaluate artifact" [7]. The activities proposed in the framework may look highly sequential, but a design science project is carried out in an iterative way moving between working on the different activities.

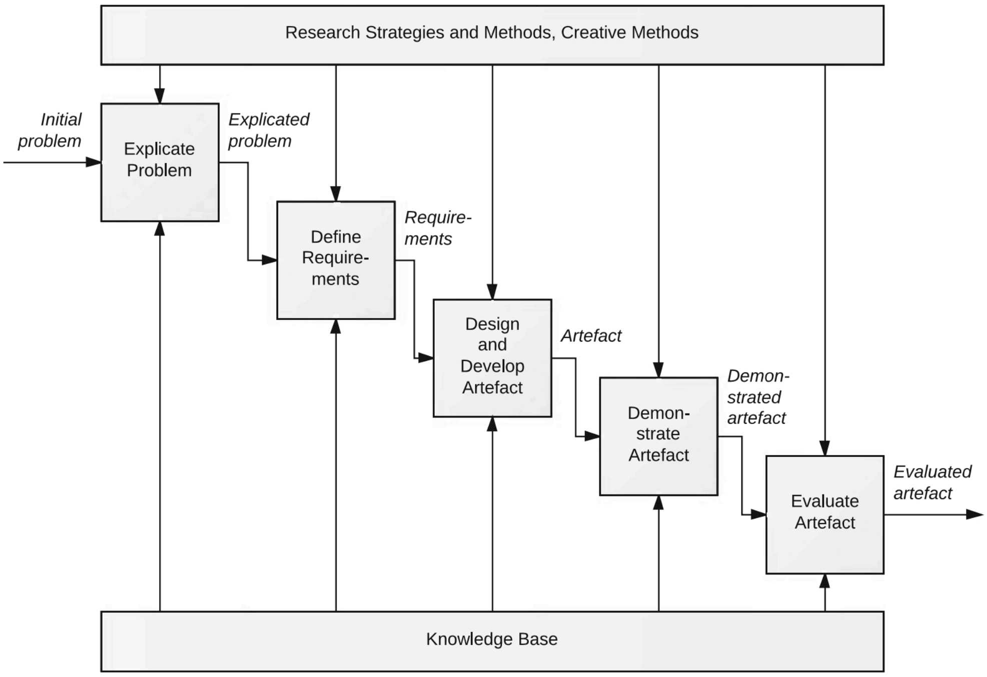


Figure 3.1: The method framework for design science research with research strategies and knowledge base from [7], page 82.

The arrows in Figure 3.1 indicate the input and output from each action. These arrows should not be interpreted as temporal orderings but as input–output relationships meaning that in principle, every activity can receive input and produce output for any other activity [7].

Research strategies, research methods, and creative methods, which are allocated on the top of Figure 3.1, describe what knowledge is used for governing an activity, and are used to aid the development process. A research strategy is an overall plan for conducting the research study, such as experiments, surveys, and case studies. Research methods tell the researcher how to collect and analyze data, which can be done through interviews, questionnaires, or other methods. Creative methods are more relevant during the design and development phase and can be activities such as brainstorming, participative modeling, empathetic thinking, and lateral thinking [7].

The design process will also be affected by the knowledge that the developers and other stakeholders possess. This accumulated knowledge is known as the knowledge base.

Most research projects tend to focus on only some of the phases in the method framework as it might be too comprehensive to perform all the phases in detail [7]. Based on the focus of the design science project there are at least five typical cases of design science research: "Problem-focused design science research", "Requirements-focused design science research", "Requirements- and development-focused design science research", "Development- and evaluation-focused design science research", and "Evaluation-focused design science research" [7].

In this thesis project, we conducted "Development- and evaluation-focused design science research", as we were developing an artifact, testing, and evaluating it. To ensure that the development is done in a proper way the five phases of the method framework by Johannesson and Perjons were used [7]. The next sections explain the theory behind each phase while chapter 4 describes what we did during each phase in this project.

Explicate Problem

The task of the explicate problem activity involves exploring and examining a practical issue. It is important to accurately define and justify the significance of the problem by demonstrating its relevance to a particular practice. The problem should have a broader appeal, not only affecting a local practice, such as at a single company but also impacting a global practice, such as a research field [7]. Additionally, during the explicate problem phase, an investigation into the root causes of the problem was done, to identify and analyze the events that lead to the problem. The SLR was used to help scope the problem which the artifact should solve.

Define Requirements

During the second phase, the researchers attempt to define a solution to the explicated problem. In design science, this solution will be an artifact with a set of requirements. These requirements are a collection of demands on the proposed artifact that are needed for the artifact to serve its purpose. The requirements will not only help for the defining the needed functionality but will also help to define the structure and the artifacts environment [7]. Defining the requirements used to create an artifact was based on the results from the SLR, a review of additional research papers, and discussions with a research security engineer from Visma.

Design and Develop Artifact

The design and develop artifact phase is where the artifact is constructed. The artifact should address the explicated problem and fulfill the requirements defined in the define requirements phase. This phase contains four activities: Imagine and Brainstorm, Assess and Select, Sketch and Build, and Justify and Reflect [7]. It is important to justify and reflect on the selections made regarding functionality as well as its structure as it will improve the verifiability of the project [7]. In order to gather further input on the design and development of the artifact, we had a conversation with two security engineers from Visma. Additionally, we distributed a questionnaire among Visma security experts in order to identify which trust criteria they considered the most essential to implement. We developed AutoTrust using the acquired information from the conversations and the questionnaire. As this phase entailed both the design and the development it was the most extensive phase.

Demonstrate Artifact

The demonstrate artifact phase is where the developed artifact is tested. First, the researchers need to choose and justify a case where the artifact can be tested. The case needs to be representative of the problem and provide enough of a challenge to work as an adequate test bed. The researchers also need to justify which parts of the artifact will be tested. Different cases might help demonstrate different components of the artifact. The demonstration can be considered a weak form of evaluation [7]. The artifact and its functionality and correctness were tested by analyzing 100 NuGet packages and by comparing the execution time of the artifact with the .NET command for adding packages.

Evaluate Artifact

The fifth and last phase is the evaluate artifact phase. The main objective of this phase is to determine how well the artifact solves the explicated problem with the constraints defined by the requirements. This determination can be done by the use of evaluation strategies that can be divided into different categories: an ex-ante evaluation, an ex-post evaluation, a naturalistic evaluation, and an artificial evaluation [7]. The ex-ante evaluation is done when the artifact is evaluated without being used or even being fully developed, while the ex-post evaluation means that the artifact is evaluated after it has been employed [7]. A naturalistic evaluation means that the artifact is evaluated in the real world, while an artificial evaluation implies that the artifact is evaluated in an artificial setting such as a laboratory [7].

According to Johannesson and Perjons, there are six different evaluation goals that can be the purpose of the evaluation phase [7]:

- Evaluate if the artifact is effectively solving the problem.

- Evaluate the functional and non-functional requirements proposed used on the artifact.
- Compare the artifact to other similar artifacts that intend to solve the same or a similar problem.
- Investigate the side-effects such as unintended or harmful effects of the artifact.
- Investigate formalized knowledge about a designed artifact by confirming, disproving, or enhancing the design theory.
- Evaluate the artifact formatively in order to identify opportunities for improvement in further design.

For evaluating the artifact we used the first 4 of these goals. We did three different tests for evaluating the artifact, based on these goals. The first test involved comparing the artifact's evaluation of three NuGet packages with the manual evaluation of those same packages carried out by computer science students. In the second test, we conducted another experiment where we assessed 100 packages using OpenSSF Scorecard, and then compared the results with the evaluation performed by the artifact on the same 100 packages. For the third test, we engaged Visma employees in testing the artifact and conducted interviews to gather their feedback on its performance and usefulness.

3.2 Research Strategies

The research strategy is the overall plan for the research study, and it is used as a guide for planning, executing, and monitoring the study [7]. Common research strategies are experiments, surveys, case studies, ethnography, grounded theory, action research, phenomenology, simulation, mathematical and logical proof [7]. In Roel J. Wieringa's book about design science, he presents five research strategies highly relevant to information systems and software engineering, where one of them is technical action research [57]. We have chosen to use both experiments and technical action research to help us find the answer to the research questions.

3.2.1 Technical Action Research

Technical action research involves employing a newly created artifact to assist a client in order to test it out in the real world [57]. Technical action research is an example of validation research, where one can observe how the artifact interacts in its real-world environment [57]. It is important to note that during technical action research, the artifact is not used by the stakeholders outside of the research context [57]. When designing these kinds of research strategies it is important to consider the artifact, the test subjects, the environment, and how to collect data about the artifact's performance. These factors need to be aligned so they can support the planned inferences from the data collection.

There are three roles the researcher needs to have during technical action research [57]:

1. Technical researcher, who designs the artifact to solve the general problem.
2. Empirical researcher, who answers some validation knowledge questions about the artifact.
3. Helper, who creates the client-specific version of the artifact to help the client with their needs.

The researcher has to fulfill all these roles and keep them separate while the artifact is tested as part of the client's engineering cycle [57].

We have taken the role of the technical researcher both during the work on the SLR and during the "Explicate problem", "Define requirements", and the design section of the "Design and develop artifact" phases. The role of a helper was mainly taken during the "Define requirements" and the "Design and develop artifact" phases as these are the phases where the focus is placed on the client's needs and system. After the artifact was created we took the role of an empirical researcher and evaluated how well the artifact solved the problem during the "Demonstrate artifact" and "Evaluate artifact" phases. During this thesis project, the client was Visma, and some of their developers tested the artifact during their everyday work.

3.2.2 Experiments

An experiment is a type of empirical investigation that explores the cause-and-effect relationships between variables and follows the steps shown in Figure 3.2. The main objective of conducting an experiment is to either confirm or reject a presumed causal relationship between a specific factor and an observable outcome [58]. This casual relationship is formulated in the form of a hypothesis, which the experimenter wants to test. As part of evaluating the artifact, we conducted two experiments and formulate three different hypotheses. The hypotheses are expressed through dependent and independent variables. The dependent variables correspond to the outcome of the experiment, while the independent variables are the variables that we controlled and changed in the experiment.

After having formulated the hypothesis, the next step of experiment planning involves selecting subjects. During the first experiment, the subjects were the computer science students and in the second they were the NuGet packages we wanted to use in the evaluation. This selection can be either a probability or non-probability sampling. Then one has to plan the experiment design, which must be planned carefully as the conclusion one can draw from the experiment depends on the chosen design. The last steps are instrumentation, consisting of objects, guidelines, and measurement instruments, and validity evaluation where one evaluates the validity of the result.

We have conducted both an experiment on computer science master's students in their final year and an experiment evaluating 100 packages with OpenSSF Scorecard. For the student experiment, we wanted to find out how the risk evaluation given by computer science students differs from the evaluation done by AutoTrust. To answer this we created two hypotheses that we wanted to reject:

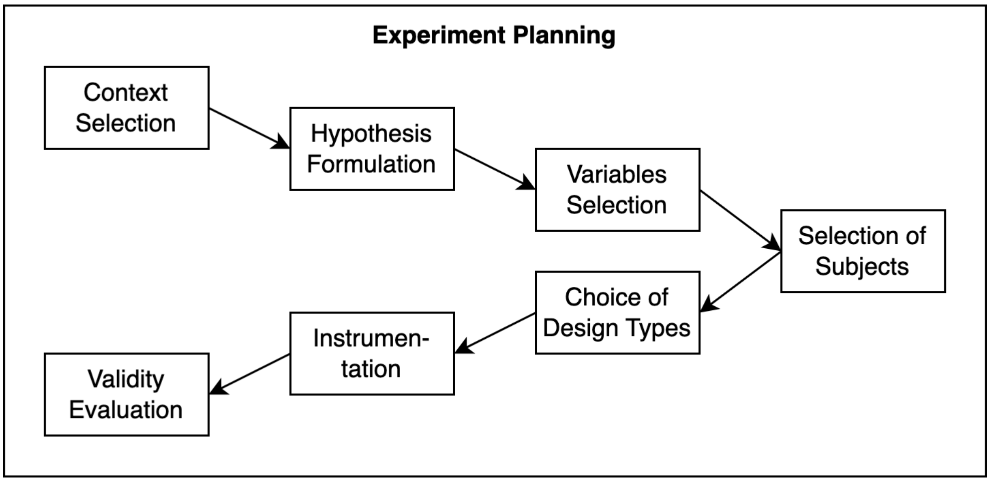


Figure 3.2: Experiment planning from [58], page 77.

- **Hypothesis 1:** The students will provide a risk review of NuGet packages that is equal to the AutoTrust review.
- **Hypothesis 2:** The students will consider the same amount of trust criteria or more than AutoTrust when evaluating NuGet package risk.

For the OpenSSF Scorecard experiment we wanted to find out how the ranking of packages done by OpenSSF Scorecard compares to the ranking done by AutoTrust. To answer this we created one null hypothesis and one alternative hypothesis. When conducting experiments, the experimenter aims to reject the null hypothesis with the highest degree of significance possible. The alternative hypothesis is the hypothesis that is supported when the null hypothesis is rejected. The null hypothesis and the alternative hypothesis we wanted to test were:

- **H_0:** The ranking of packages and the tool used to give those rankings are not related given the 100 packages. The proportions of rankings are the same for the two tools.
- **H_a:** The ranking of packages and the tool used to give those rankings are related given the 100 packages. The proportions of rankings are not the same for the two tools.

To refute the hypotheses, we employed two distinct methods for conducting the experiments. For the student experiment the hypothesis testing was conducted visually through diagrams accompanied by explanatory texts [59] to support the findings. In the OpenSSF Scorecard experiment, we utilized the chi-square statistic for hypothesis testing, explained in subsection 3.4.2. The reason for not using the chi-square statistic for the student experiment was due to a small sample size of only 12 students.

In the OpenSSF Scorecard experiment, we utilized the Chi-square statistic for hypothesis testing, explained in subsection 3.4.2. However, due to the small sample size in the student experiment, the Chi-square statistic is not applicable. Instead, the hypothesis testing for the student experiment was conducted visually through diagrams accompanied by explanatory texts [59] to support the findings.

The independent variables for the student experiment are the packages that the students reviewed and the order of these packages. The dependent variables are the overall security risk score and the trust criteria consideration score, both given by the students. A total of 12 students participated in the testing of 3 different packages. Since there are 6 possible orders in which the packages can be tested, each order was assigned to 2 students. To find computer science students we used convenience sampling, choosing students working in our office area at the university.

In the OpenSSF Scorecard experiment the independent variable was the 100 packages to be reviewed. The dependent variable was the calculated risk scores. As there were tools and not humans doing the evaluation of this experiment, the order of packages was not considered.

We opted for a one-factor design with two treatments that we wanted to compare against each other for all hypotheses in our design selection process. For the student experiment, the two treatments are manual risk evaluation, done by computer science students, and automatic risk evaluation, done by us using the AutoTrust tool. For the OpenSSF Scorecard experiment the two treatments are the risk evaluation given by AutoTrust and the evaluation given by OpenSSF Scorecard. The last two steps of experiment planning are instrumentation and validity evaluation, and they will be explained in subsection 4.5.1 and subsection 4.5.2.

3.3 Data Collection Methods

To aid the different phases of the method framework there is a need to collect useful data to make correct decisions and create quality output for the next phase. There are multiple data collection methods and Johannesson and Perjons point out that the five most used data collection methods are: questionnaires, interviews, focus groups, observation studies, and document studies [7]. They also describe how some data collection methods are closely associated with specific research strategies, such as surveys which typically use questionnaires.

3.3.1 Document Studies

Document studies are an investigation into already published information about the topic to discover what is known and unknown about a topic. As part of the preparation for this thesis, a SLR was conducted to gather information from document types such as academic publications, government publications, and news [10]. The SLR has been used to increase the knowledge base and will contribute to the "Explicate problem" and "Define requirements" phases. The details on how the SLR was conducted can be found in Appendix C.

3.3.2 Questionnaires

Questionnaires are usually used to obtain answers to questions that are brief and unambiguous from a set of respondents [7]. Questionnaires are great since they are inexpensive, but they can result in skewed data. For example, if the first question affects how the respondents answer the later questions or if the population asked is not representative to generalize the answers. For design science, questionnaires can help to obtain greater insight into the stakeholders, help to define requirements, or evaluate the general response from a large group about an artifact.

We used questionnaires in this thesis to have Visma developers evaluate the proposed trust criteria and to have computer science students manually assess the risk of NuGet packages.

Visma has multiple development teams and the different teams can have different guidelines and workflows. To gain insight into this, a questionnaire was sent out to security experts at Visma through a Google form. The questions used in the questionnaire can be found in section B.2. We chose to ask security experts at Visma as we believed they would have the most knowledge about software supply chain security. We used a Google form as Visma has a data processing agreement with Google. The results from the questionnaires can be found in section 4.3.2.

During the student experiment, we also used questionnaires to collect data from the computer science students' assessment of NuGet packages. In order to maintain anonymity and privacy, we opted for Google Forms as personal information was not collected. Additionally, we aimed to mitigate the influence of question order on responses, a common issue with questionnaires, by altering the sequence in which the packages were presented for assessment. The form can be found in section B.4.

3.3.3 Interviews

An interview is a two-way communication session where the respondent should do most of the speaking. The respondent provides information while the researcher controls the agenda by asking questions. Interviews are effective for gathering more complex data as researchers can ask follow-up questions if things are unclear or the respondent is providing interesting information. Interviews are however time-consuming as the appointment for the interview needs to be planned and the researcher needs to analyze and categorize the answers from the interviews, which can be video, audio, and field notes. Interviews can be used to help explicate problems, define requirements, obtain information about designs, and evaluate if the test subjects approve of the artifact.

We used interviews to evaluate how well the tool worked as part of a developer's workflow. The interviewees were software engineers at Visma who tested out the artifact as an integral part of their programming tasks in their daily work. The interviews were conducted online using Microsoft Teams which is the standard NTNU software used for video meetings.

Prior to conducting the interviews, we ensured that participants had consented to record their answers. Once this was verified, we asked the questions that can be found in section B.5. The responses were recorded with video and the dictaphone extension on Teams. After the interviews were conducted the replies were analyzed. The results can be found in section 4.5.

3.4 Data Analysis

After the data is collected using the data collection methods mentioned above it needs to be analyzed. The raw data collected needs to be prepared, interpreted, analyzed, and presented before it can be used to draw conclusions [7]. There are two main forms of data analysis: quantitative and qualitative.

3.4.1 Quantitative

Quantitative data analysis is a research methodology that involves the use of numerical data to analyze and draw conclusions about a given phenomenon or research question [60]. This type of analysis typically involves collecting numerical data through various methods such as surveys, experiments, or observations, and using statistical tools and techniques to analyze and interpret the data. The numerical data used in quantitative data analysis is either continuous or discrete.

Continuous data refers to data that can take on any value within a certain range, such as height, weight, or temperature. These data points can be measured to arbitrary precision, and the values are typically represented on a continuous number line. Continuous data is often analyzed using descriptive statistics such as mean, median, and standard deviation.

Discrete data, on the other hand, refers to data that can only take on finite possible values, such as the number of siblings a person has or the number of students in a class. These data points are whole numbers and cannot be measured in decimal points. Discrete data is typically visualized using frequency distributions, histograms, and measures of central tendency such as mode or median.

We collected and analyzed quantitative data during the runtime demonstration of AutoTrust and the .NET command installing the 10 most downloaded NuGet packages (subsection 4.4.2). Specifically, we examined the execution times of the installation methods, both of which are continuous data measured in time.

3.4.2 Qualitative

Qualitative data analysis is the process of examining non-numerical data to identify patterns, themes, and insights [60]. It involves analyzing data such as interviews, field notes, and recordings to gain an understanding of the experiences, perspectives, and meanings of the participants. Qualitative data analysis typically involves coding the data into categories, which are then analyzed to identify patterns and themes. How the qualitative data are interpreted may be affected by the researchers' background, values, and experiences [7].

The qualitative data variables are differentiated between ordinal and nominal data:

- Nominal data is a type of categorical data where the values represent discrete and separate categories or groups that cannot be logically ordered or ranked. Examples of nominal data include gender, race, nationality, religious affiliation, and political party affiliation. Nominal data can be analyzed using frequency counts and percentages, and statistical measures such as mode can be used to compare groups. However, nominal data cannot be analyzed using measures of central tendency, such as mean or median, as there is no inherent ordering to the categories.
- Ordinal data are used on categories when there is an order or ranking, but there is no consistent meaning to the difference. An example can be when measuring economic status, the income ranking could be wealthy, middle income, or poor.

We used multiple different methods to gather qualitative data throughout this study. The qualitative data are gathered through the Visma questionnaire, presented in subsection 3.3.2; the 100-packages demonstration, presented in subsection 4.4.1; the student experiment, presented in subsection 4.5.1; the testing of OpenSSF Scorecard, presented in subsection 4.5.2; and the interviews with Visma software engineers, presented in subsection 4.5.3.

From the Visma questionnaire, we collected nominal data about how the teams handle dependencies and what the respondents have experience with. We also collected ordinal data about the trust criteria since we asked the interviewees to rank trust criteria based on their perceived usefulness:

1. Does not matter
2. Not necessary
3. Not necessary but insightful
4. Valuable
5. Crucial

In the 100-packages demonstration, we gathered the overall ranking made by the AutoTrust tool. As this ranking is a discrete star rating from 1-5, it will be ordinal data. From the student experiment, we gathered ordinal data by having the students rank various NuGet packages based on the perceived usefulness explained above. In addition, we gathered nominal data by having the students explain their reasoning and thoughts around the trust criteria presented. When testing the 100 packages with OpenSSF Scorecard we gathered ordinal data, as OpenSSF Scorecard gives a discrete ranking from 0-10 to each of the packages. Finally, the interview with the software engineers at Visma provided us with nominal data, as they provided their thoughts and feedback after having used AutoTrust. To assess the data and reject the hypotheses, we employed both a general analysis and a mathematical approach utilizing chi-squared statistics.

General Analysis

To analyze qualitative data there are three different approaches [7]:

1. Content analysis: is used to classify elements of the qualitative data and then calculate the frequencies of the elements in categories.
2. Grounded theory: is similar to content analysis, but the categories emerge gradually during the researcher's work.
3. Discourse analysis: is used to find the hidden meaning of a text, and the researcher tries to use concepts and theories to interpret the text.

To analyze and extract data from the Visma questionnaire, the student experiment, and the Visma interviews we utilized the method of content analysis. By employing content analysis, we were able to gain a deeper understanding of the responses provided by the participants, as well as to draw broader conclusions based on the data collected.

In addition, we used graphs and visual comparisons for understanding the results of the 100-packages demonstration and the testing of OpenSSF Scorecard. For the hypothesis testing of the student experiment and evaluating it against the 100-packages demonstration we combined graphs with the result of the content analysis performed on the student experiment.

Chi-Squared Statistics

To accept or reject the null hypothesis of the OpenSSF Scorecard experiment we performed the chi-squared test of independence. The chi-squared test is a statistical test used to determine whether there is a significant difference between two categorical variables. By comparing the observed data with the expected data under the assumption of the null hypothesis, it becomes possible to determine if there is a significant deviation, thereby leading to the rejection of the null hypothesis [61].

A chi-squared test is performed in multiple steps:

1. Formulate a null hypothesis and alternative hypothesis
2. Create a contingency table that cross-tabulates the observed frequencies of the categories for each variable
3. Use the null hypothesis, stating that there is no association between the variables, to calculate the expected frequencies for each cell in the contingency table.
4. Compute the chi-squared test statistic by summing up the squared differences between the observed and expected frequencies, using the following formula, where O_{ij} represents the observed frequency in cell (i, j) of the contingency table, and E_{ij} represents the expected frequency in cell (i, j) of the contingency table:

$$\chi^2 = \sum \frac{(O_{ij} - E_{ij})^2}{E_{ij}} \tag{3.1}$$

5. Determine the degrees of freedom. It is equal to $(r-1) \times (c-1)$, where r is the number of rows and c is the number of columns in the contingency table.
6. Determine the critical value corresponding to a chosen significance level and the degrees of freedom.
7. Compare the chi-squared test statistic with the critical value.
   - If the chi-squared test statistic is greater than the critical value we can reject the null hypothesis. This indicates that there is evidence of an association between the variables.
   - If the chi-squared is less than the critical value we cannot reject the null hypothesis. This indicates that there is no evidence of an association between the variables.

3.5 Ethics

According to Oates, it is crucial to consider ethical concerns during a research project [62]. This includes treating participants fairly, ensuring their rights are followed, and minimizing potential negative consequences. To gather data from participants we have used questionnaires and interviews. It is important to ensure the participants' privacy, therefore, the data is anonymized and made confidential. All participants have been informed about the study's purpose, methodology, and benefits, and have voluntarily signed a consent form to participate. They have also had the opportunity to withdraw their participation at any point without an explanation.

This thesis follows the ethical guidelines from the Norwegian University of Science and Technology (NTNU) [63]. To ensure that all data is handled properly we have created a notification form and a data management plan following the recommendations from Sikt[1]. The notification form and data management plan are used to ensure that we follow the requirements of the Personal data act [64], which includes the General Data Protection Regulation (GDPR) requirements. In addition, they make sure that the data collection and management follow the standards of the Research Council of Norway and the European Union. The notification form in section A.1 was approved on the 28. of January 2023 by Sikt and a copy of the approval can be found in section A.2.

We decided not to collect any personal information using the questionnaire as it was not needed for the project. We still provided information about the project and how the data would be processed. The respondents were still asked to consent to the information provided, even though no personal data was collected. For the interviews, the participants gave their specific, informed, unambiguous, and voluntary consent, in line with the Personal Data Act [64]. As the consent should not be given just orally we also sent out a consent form to all participants and had them sign it at the start of the interview. The information provided to the

[1] https://sikt.no/

participants to obtain consent was both read out loud and included in the consent form and was based on the recommendations from Sikt. The information letter about consent can be found in section B.1. After the interviews, a transcript of the interview was sent to the participants so they could correct any errors.

The data from the questionnaire was collected using Google Forms because Visma has a data processing agreement with Google. The Google form was also created and distributed by a Visma employee as Visma wanted someone with a Visma email account to be the owner of the form. The data storage followed the data management plan and was stored securely on the NTNU's Office 365 Sharepoint, as NTNU has a data processing agreement with Microsoft. For improved security, the data was anonymized continuously. The data was deleted at the end of the project and is not archived. The data management plan can be found in section A.3.

We do not have any affiliation with Visma, but our supervisor works at both NTNU and Visma. However, we believe that this employment has not affected the integrity of the project or the recorded data. Another argument that reduces the likelihood that we or Visma would intentionally alter the results, is that the tool will be released as open source and not proprietary so there is no monetary interest for us or Visma.

The artifact's design is based on the researchers' work and there is often a tendency to have increased enthusiasm for personal craftsmanship, which can lead to improper evaluation of the artifact. However, both negative and positive results regarding the artifact contribute to new knowledge, and we believe that we have not intentionally skewed data to only accentuate the artifact's strengths. Overall, we hold the belief that we have conducted good measures to ensure that the research project follows ethical standards.

Chapter 4

AutoTrust

This chapter explains what we did during the different steps of the design science research when developing AutoTrust. First, we explicate the problem, meaning we formulate the problem, justify its importance, and investigate its underlying problems. Then we define the requirements for an artifact that can be used to solve the problem. The next step is to design and develop the artifact, which includes designing both the functionality and structure of the artifact. Then we demonstrate how the artifact can be used, proving its feasibility. The last step is evaluating the artifact, where we show the results and evaluate how good the artifact is at solving the explicated problem. The artifact, AutoTrust, can be found at `https://github.com/HallvardMM/AutoTrust`.

4.1 Explicate Problem

This phase is the first in the design science methodology. It lays the foundation for further design, development, and testing. The goal is to scope the problem and give an exact definition of the problem. After the problem is justified we argue why it is of general interest, solvable, and leads to novel research.

4.1.1 Position the Problem

In software development, the use of tools and code developed by other programmers is essential. This can encompass a variety of elements, including the programming language, operating system, and dependencies imported directly into projects. Code incorporated into a project is commonly known as third-party dependencies. Software developers often rely on these dependencies due to time constraints or other limitations. However, a concern is that vulnerabilities in these dependencies can potentially also lead to vulnerabilities in the programs that depend on them.

When a threat actor inserts malicious code into a package that is used within a project it is referred to as a supply-chain attack. There is an increasing number of these attacks and the problem is affecting all developers no matter if they are working for academia, in corporations, or privately [10]. One example of software supply-chain attacks is when a malicious software package is introduced as a dependency, which infects the software project. Consequently, ensuring software development and software maintenance security becomes essential for developers.

4.1.2 Formulating the Problem

Third-party dependencies can lead to exploits either through injected malware or weakware. The difference between these is that the damage characteristics of malware are intentional and planned, and often require some technical expertise to implement and insert. Weakware on the other hand refers to weaknesses in the software that are mostly unintentional or accidental [13]. A compromise of a package will compromise the rest of the project as well, so for large projects, the consequences can be huge. To avoid getting compromised it is crucial to do a proper evaluation of the third-party software to minimize the chances of downloading or updating a dependency with something malicious. This evaluation can be done by evaluating the provided metadata for the third-party dependencies. The overall problem with third-party software boils down to:

> *Threat actors can use dependencies as an attack vector to infiltrate software projects and there is no simple way of evaluating the security of these dependencies prior to installing them.*

4.1.3 Justify the Problem

The importance of software security and stopping supply-chain attacks is increasing and the 2020 SolarWinds hack, mentioned in section 1.1, is a recent reminder of the consequences that can occur from a breach [4]. A compromise of a company can lead to a long-lasting loss of reputation and trustworthiness, and the cost of both time and money can be massive. A compromise of this nature can have severe repercussions not only for companies but also for individual software users. This highlights the significance of software and dependency security for all stakeholders involved, including developers, owners, and users.

Third-party dependencies or packages are stored in a package registry and it is through them the dependencies are distributed. These registries can therefore be used by malicious actors to distribute their malicious packages. There are multiple examples of malicious packages that have been found, and back in 2022, over 140 000 phishing packages were published to NuGet, npm, and PyPI by the same threat actors [65]. Also, according to Sonatype, a company specializing in software supply chain security tools, there has been an average 700% jump in repository attacks from 2020-2022 [66].

To address the issue of malicious packages, a root cause analysis can reveal various underlying factors contributing to the problem. The causes found are represented in Figure 4.1, which is an Ishikawa diagram. The Ishikawa diagram consists of a main horizontal line that represents the problem, and associated slanting lines that represent direct problem causes. These slanting lines are again related to shorter horizontal lines which represent indirect problem causes. Then the smallest slanting lines are representing different aspects of the indirect problem causes [7]. The purpose of this diagram is to visualize all the potential factors that can lead to using vulnerable third-party software. This diagram helps identify all the potential factors causing an overall effect on the problem and a possible solution. By addressing these different root causes a better result can be achieved than just addressing the symptoms of the problem.

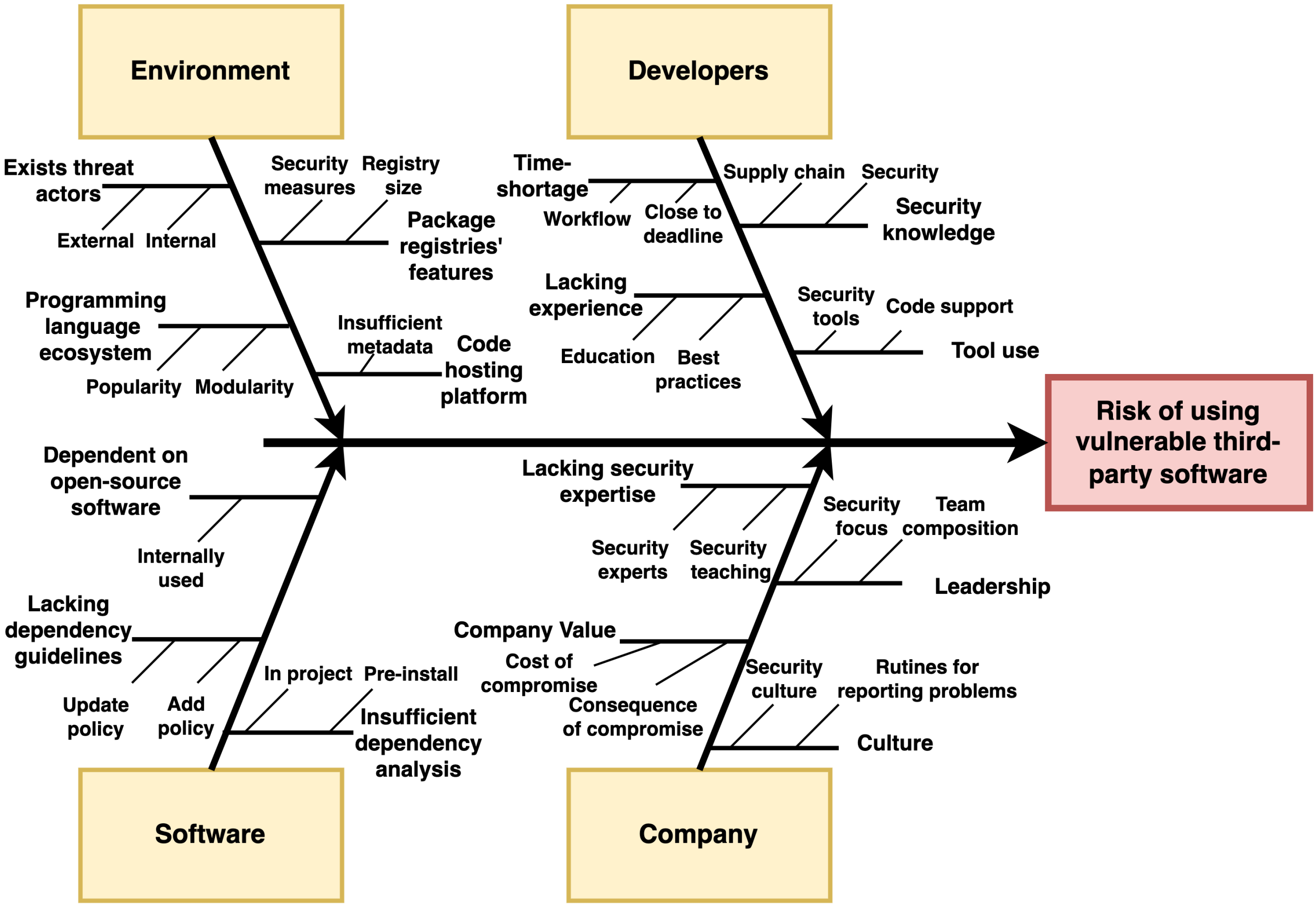


Figure 4.1: Ishikawa diagram representing problem causes.

As one can see from Figure 4.1, the problem with risky third-party software is complicated. This diagram illustrates that there are multiple underlying causes of the problem, all of which require attention and resolution.

4.1.4 Ensure General and Solvable Problem

Solving the problem with vulnerable third-party dependencies and packages through automated detection of all vulnerable packages is probably unfeasible. However, according to Ruturaj et al., tools and metrics that help assess the risk of external dependencies would go a long way toward preventing attacks [19].

To ensure the problem is solvable and manageable we have decided to focus on software packages and not other types of dependencies. Since no malicious code from a package can impact the user before the package is added, our emphasis lies in detecting and preventing the installation of malicious packages. As there are multiple package registries built for different programming language ecosystems and the information they provide is different we chose to focus on one specific registry.

4.1.5 Sources of the Problem

The general problem of supply chain security and ways to eradicate or minimize the problems have been addressed in both academia and industry. As seen in the SLR there are multiple academic papers written about this problem, in addition, there are multiple online articles describing both the issue and possible solutions [10]. Some of the current solutions that attempt to deal with the problem are npq, OSSGadget, and OpenSSF Scorecard, referenced in subsection 2.6.3.

4.1.6 Research Novelty

While both npq and OSSGadget offer valuable features, neither of them has undergone scientific development with a specific focus on research and testing, as we do in this thesis. Considering OpenSSF Scorecard, it is worth noting that Zahan et al. conducted an evaluation of the tool [67], and they compared it with the npm and PyPI ecosystem. Our work delves into the NuGet ecosystem, emphasizing its significance while shifting the focus away from npm and PyPI, as they have been the most discussed in the literature [10]. Another novelty of this research is the presentation, use, and evaluation of new trust criteria. Not all of these trust criteria presented in Table 2.2 are incorporated in npq, OSSGadget, or OpenSSF Scorecard. This raises the question of whether these new trust criteria are valuable and feasible to assess, and we have therefore evaluated them. Which trust criteria are included in AutoTrust, npq, OSSGadget, and OpenSSF Scorecard is presented in section 4.3.8. Another novelty of this research is exploring if users find these pre-install tools valuable and what their current process is when adding new software packages.

4.2 Define Requirements

For solving the explicated problem with dependencies being used as attack vectors the next step of the design science process is to define requirements. This process of defining the requirements was based on the literature from the SLR, a questionnaire, and discussions with a research security engineer from Visma. The questionnaire, mentioned in subsection 3.3.2 with the questions that can be found in section B.2, was used when asking the security expert participants how they were dealing with and downloading dependencies. The discussion with the research security engineer was valuable to make sure that the artifact and its requirements would fit with Visma's current development practices.

4.2.1 Outline Artifact

The artifact we made is an instantiation artifact as it is a working system that can be used in practice. The artifact provides a risk assessment of NuGet packages prior to the developers adding the packages to their projects. This assessment is done by fetching metadata that is used for evaluating the trust criteria's thresholds and informing the developer if the package passed or failed the set thresholds.

Then after the risk assessment is provided, the user is prompted if they would like to add the package. If the user wants to install it, the installation is done by the artifact calling the standard .NET command for adding packages. This command can also be run separately in the CLI:

```
dotnet add package NameOfPackage
```

One example where the tool would prove useful is when a developer is trying to download a deprecated package without knowing it is deprecated. When they run the script, it gives a warning in the CLI saying that the package is deprecated including other potential failed trust criteria. The user is then asked if they still want to download the deprecated package and can take an informed decision based on their new knowledge.

Justify Importance

In the work by Ladisa et al., mentioned in section 2.7, one of the safeguards against software supply chain attacks proposed was "Establish vetting process for open-source components hosted in internal/public repositories". The domain experts gave it a high ranking on both utility and cost with 4.1/5 on utility and 4.3/5 on cost. The software developers ranked it 3.8/5 on cost [11]. This strengthens the credibility as the domain experts consider such a vetting as useful, but costly to do. Creating an artifact for automated assessment would cut the cost and make such vetting more useful.

An article published in late March 2023 by the JFrog Security Research team also shows why this artifact can be useful. They reported that there had been "a sophisticated and highly-malicious attack targeting .NET developers via the NuGet repository" [6]. It was one of the first reports about malicious NuGet packages that targeted developers directly [6]. The malicious packages used typosquatting techniques to bait the users into downloading them. The packages contained a payload with a "download and execute"-attack using the *init.ps1*-file that is executed on installation. The malicious packages had a combined download count of over 150 000, however, this might have been inflated with bots by the attackers to boost the credibility of the package [6]. If the developers used the artifact they could get a warning that the package contained an *init.ps1*-file and they could inspect the file before downloading it and getting compromised.

4.2.2 Elicit Requirements

One way to classify requirements for an artifact can be as functional requirements (FRs) and quality attributes (QAs) [68]. Functional requirements specify the demands that are imposed on the artifact based on the problem to be addressed and the criteria from the stakeholders [7]. Quality attributes are qualifications of the functional requirements or the general artifact [68]. They describe how well the artifact fulfills its overall functionality.

Functional Requirements

There are multiple functional requirements for the artifact. The list below contains the FRs for the artifact and a sub-level for each requirement with a justification for them.

- **FR-1:** The artifact should evaluate packages before they are installed.
    - During the SLR we found that some attacks target the developer when they install packages [24]. The packages should therefore be evaluated before they are installed.
- **FR-2:** The artifact should be possible to run from the CLI.
    - In the conversation with the research security engineer, we agreed to extend the "dotnet add package"-command and the artifact should run in combination with this command.
- **FR-3:** The artifact should run in the .NET ecosystem and work for all NuGet packages from the central repository.
    - In the conversation with the research security engineer, we agreed to scope the artifact to focus on the .NET ecosystem and the central NuGet repository. NuGet and .NET were chosen since Visma employees work in this environment.
- **FR-4:** The artifact should display information to the user if the package passed, or failed the thresholds of the trust criteria. In unclear circumstances, the user should get a warning.
    - The requirement arose during the talk with the research security engineer. The artifact should inform the user, but in the end, the user should have the final say if the package gets added.

- **FR-5:** The artifact should fetch package metadata in real-time.
    - This criteria came from the SLR as it was found that the data sources might update frequently and the user should get the newest assessment of the package.
- **FR-6:** The artifact should provide an option for getting a more detailed explanation of the evaluation process.
    - This requirement arose during the talk with the research security engineer. The users should be able to get a more in-depth explanation to learn what is evaluated and understand why a package fails some thresholds.

Quality Attributes

The quality attributes of the artifact help in the process of improving it as they provide guidelines on what to focus on during the design and development. The bullet points below describe the five main QA categories we considered during the development of the artifact and a sub-level describing the actual criteria to fulfill the attributes.

- **QA-1:** Usability
    - The package should be easy to use by supporting a verbosity flag, and help flag, and have a similar syntax to other .NET tools
    - The artifact should be possible to install and start using within 15 minutes of opening the GitHub repository with the help of a README.

The artifact should be easy to use and install. It should give clear information so users understand why a package failed a given threshold and why this is associated with increased risk.

- **QA-2:** Portability
    - The system should work with newer versions of Windows, macOS, and Linux.

Developers at Visma are using different operating systems and the artifact should work for all the developers as we believe it is more useful when every team member can run an automatic package assessment.

- **QA-3:** Security
    - The tool should not use any third-party dependencies except packages provided directly in .NET, also known as system packages.
    - The tools should not download packages without the user's confirmation.

As mentioned in section 2.6, software tools might be used by threat actors as an attack vector. It is important that this artifact does not lead to an increased security risk for the developers using it as this contradicts the main intention of the artifact.

Also, the reason for allowing system packages is that these are directly included in .NET and are not added from NuGet or other external sources.

- **QA-4:** Performance
    - The installation time for a package using the tool should not exceed three times the standard runtime of the "dotnet add package" command.

Today the process of adding new packages is quick. Since we are adding an extra step to the installation process it is important that the artifact does not impose a long time-penalty as developers might choose not to use it.

- **QA-5:** Modifiability
    - It should be possible to implement new trust criteria without altering existing ones.

In the cybersecurity field it is a cat-and-mouse game between attackers and defenders with constantly changing terms. The artifact needs to easily support new trust criteria and thresholds to help identify indicators of new and more advanced attack forms.

4.3 Design and Develop Artifact

After the requirements were defined the development and design phase was initiated. After some discussions, it was decided that the tool would work as an CLI tool that would extend the functionality of the "dotnet add package"-command. This would work as a shift-left approach where the evaluation of the package's security risk would be performed earlier in the lifecycle. The .NET packages can be installed in two primary ways: via the CLI, or directly from the Visual Studio integrated development environment (IDE) through either the NuGet Package Manager or the Package Manager Console. We decided to only extend the CLI option.

4.3.1 Architecutre

After the initial idea was assessed and selected, we started sketching and building the artifact. AutoTrust needs input data about the package name, package version, and other optional arguments like project folder path and flags. This information is used to fetch data from NuGet, Github, and OSV. After the data is fetched, AutoTrust assesses the package based on the data and thresholds for the trust criteria. This information leads to a security risk score in the form of a discrete star rating between 1 and 5. The assessment of each trust criteria and the security risk score is then displayed to the user with a prompt if they want to install the package. If the user answers affirmatively the package is installed by running a "dotnet add package" process. This flow can be seen in Figure 4.2.

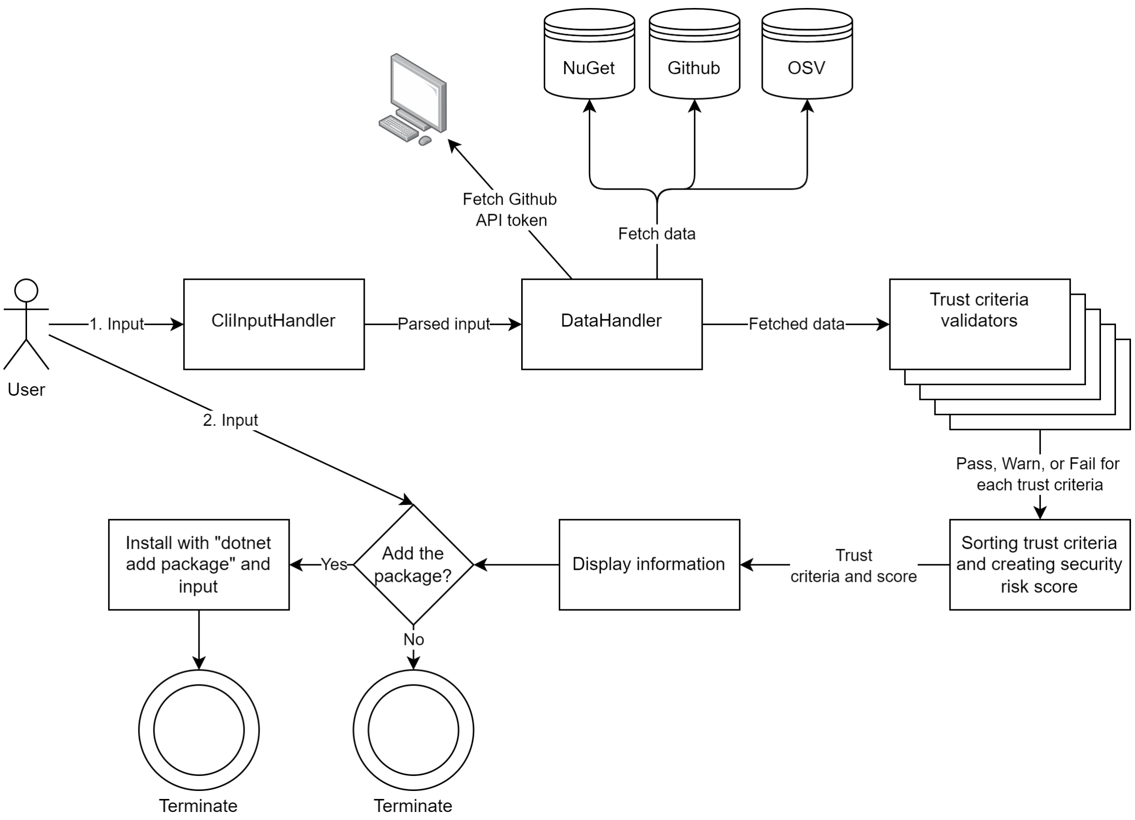


Figure 4.2: The process flow of AutoTrust.

In Figure 4.2 the DataHandler step fetches data from NuGet, GitHub, and OSV. NuGet contains a lot of package-specific information that is used by the AutoTrust tool during the evaluation. Almost all of the NuGet packages we found that had their source code open, hosted it on GitHub. We decided to fetch relevant data from GitHub that could be used when evaluating the packages, but if the source code is not open it still evaluates metadata from NuGet. GitHub also has some restrictions on the number of calls a user can do to their API. If a GitHub token is sent with the API calls to GitHub these restrictions are less strict, and the user can run the tool 83 times instead of 5 per hour without being denied by GitHub. The user is, therefore, asked to create a fine-grained personal access token on GitHub. This token is intended for personal API use and must be stored on the local machine so AutoTrust can use it.

As discussed in subsection 2.6.1 there are multiple databases available containing information about vulnerabilities. NuGet provides some information about registered vulnerabilities, but we decided to also use OSV to increase the chances of finding reported vulnerabilities. An overview of all the data fetched, and used in the package assessment, from NuGet, GitHub, and OSV is displayed in Figure 4.3. In the figure, some points like README are mentioned multiple times because AutoTrust prioritizes data from NuGet before using the data from GitHub and OSV.

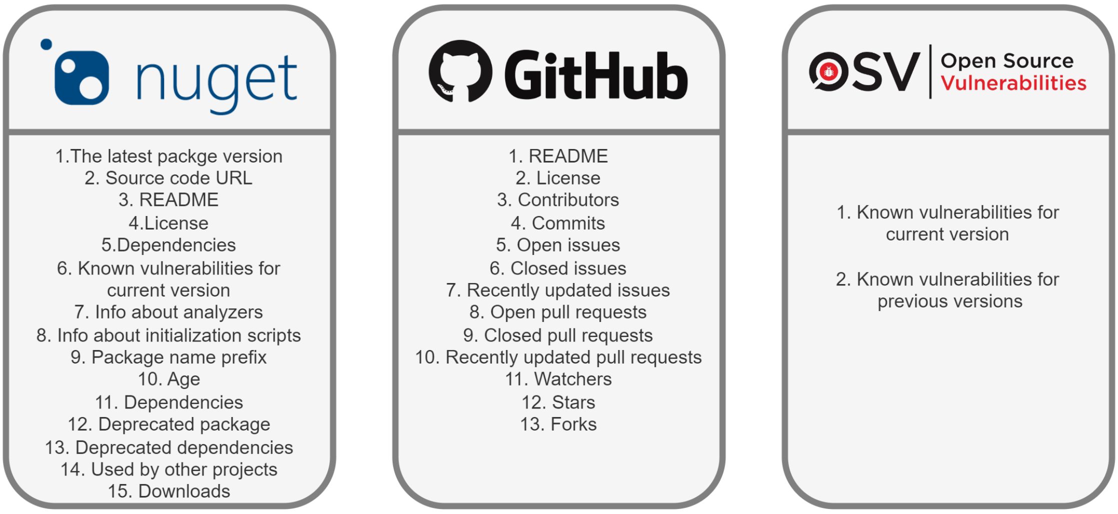

nuget
1.The latest packge version
2. Source code URL
3. README
4.License
5.Dependencies
6. Known vulnerabilities for current version
7. Info about analyzers
8. Info about initialization scripts
9. Package name prefix
10. Age
11. Dependencies
12. Deprecated package
13. Deprecated dependencies
14. Used by other projects
15. Downloads
GitHub
1. README
2. License
3. Contributors
4. Commits
5. Open issues
6. Closed issues
7. Recently updated issues
8. Open pull requests
9. Closed pull requests
10. Recently updated pull requests
11. Watchers
12. Stars
13. Forks
OSV Open Source Vulnerabilities
1. Known vulnerabilities for current version
2. Known vulnerabilities for previous versions

Figure 4.3: Data from NuGet, GitHub, and OSV

The data fetched from NuGet, GitHub, and OSV is then used to decide if AutoTrust shall pass, warn or fail the criteria for the different package. The criteria pass if everything seems fine, warn if some things could be better, and fail if the criteria are outside the given thresholds. The details about how and which criteria are validated are presented in subsection 4.3.2. Based on which trust criteria pass, warns, or fails a security risk score is calculated. More information about how the security risk score algorithm works is presented in subsection 4.3.5. After completing the trust evaluation process, the information is presented to the user. The results are denoted by the use of distinct visual cues: a checkmark and green text signify passed criteria, an exclamation point and yellow text indicate warning criteria, and an X and red text are used for failing criteria. Furthermore, a discrete security score, ranging from one to five, is displayed in the form of stars. The user is then prompted if they would like to install the package. An example of the output of AutoTrust can be seen in Figure 4.4.

4.3.2 Deciding on Trust Criteria

The process of selecting the appropriate trust criteria for AutoTrust was carried out systematically in three main activities. The three main activities were a questionnaire to Visma developers, a one-pager on Visma's intranet for getting attention and feedback on the topic, and discussions with two security engineers from Visma.

Replies to Visma Questionnaire

One of the first parts of designing the AutoTrust tool was to decide which trust criteria in Table 2.2 should be included. This was partly done by sending out a questionnaire to Visma security experts. Of the 15 respondents, all of them had more than 5 years of experience, and everyone except one had experience with NuGet. The security experts were asked to rank each trust criterion based on how valuable it would be to assess when deciding whether to use a third-party package. The ranking of the criteria was using the values "Does not matter", "Not necessary", "Not necessary but insightful", "Valuable", and "Crucial". The result from the ranking of the 15 responses is displayed in section B.3. Since the data collected from the Visma Questionnaire was ordinal categorical data, a conventional arithmetic mean was deemed suboptimal and median and mode were used instead. The trust criteria were organized based on the median and mode assigned to each trust criterion. In addition to calculating the median and mode, we analyzed the corresponding distribution depicted in the graphs displayed in section B.3. All of the considered TC, displayed in table Table 2.2, had a median and mode from "Not necessary but insightful" and above, indicating that the respondents considered the TC useful.

One-pager and Conversation with Security Engineers

We shared a one-pager, as shown in section B.9, through an internal Visma forum. From the one-pager, we received feedback on the threat of analyzers that can be used in .NET to run malicious scripts. Based on this, an additional check for analyzers was suggested added to the AutoTrust tool. After reviewing the feedback, we had conversations with two security engineers from Visma. During the conversations, the trust criteria were deliberated to elicit further feedback on the implementation. This led to a decision on which criteria to be implemented, how they should be implemented, and which thresholds should be used.

In addition to the original trust criteria and analyzers check, we decided to also check if the NuGet packages have a verified prefix. This came up as part of creating the tool when we found out that NuGet supported this feature.

4.3.3 Validators

After having decided on which TC to include, we created multiple validators that together implement the chosen TC to the extent possible. In this context, a validator in AutoTrust refers to an implementation of a trust criterion specifically tailored to the available data in the NuGet ecosystem. These validators are depicted in the "Trust criteria validators" step in Figure 4.2. The reason for converting the TC into validators is that the practical implementation was affected by outside factors such as available package metadata. When running the script it is possible to specify which version to use in the assessment, and if no version is specified then the latest stable released package will be checked. In addition, when we check dependencies, we currently check all dependencies down to a depth of 2, where the depth of the package being checked is 0. Below we describe the trust criteria checked for each validator together with a description of how they work.

Age

- **Trust Criterion 3**: Satisfactory time since the latest update was published.

To analyze the age of the package, AutoTrust checks if the package version is newer than 3 weeks or older than 1 year.

Analyzers

- **Trust Criterion 26**: The component does not contain installation scripts.

In .NET packages the analyzers are stored in a folder named "analyzers". We check if this folder is present and if there are any files in it for the package being analyzed and the dependencies.

Contributors

- **Trust Criterion 9**: The component has an adequate number of maintainers and/or contributors.
- **Trust Criterion 10**: The component is being developed by an active maintainer domain.

For analyzing contributors we check if the GitHub repository for the package has at least 2 contributors and that there is 1 active maintainer. A maintainer is defined as active if it has at least 3 commits during the last year or out of the 100 last commits if there are more than 100 commits during the last year.

Deprecated Package

- **Trust Criterion 22**: The component is not deprecated.

To see if a package is deprecated we check if it is marked as deprecated by NuGet.

Deprecated Dependencies

- **Trust Criterion 23**: The component does not depend on deprecated packages.

For deprecated dependencies we check all dependencies of the package being checked down to the specified depth to see if any are deprecated.

Direct and Transitive Dependencies

- **Trust Criterion 25**: The project has few direct and transitive dependencies.

For the dependencies we check if there are more than 20 direct dependencies and 50 transitive dependencies.

Documentation

- **Trust Criterion 8**: The component has good documentation.

For the documentation we see if it is possible to find a README on either GitHub or NuGet. If none of those are present it checks for a project URL on NuGet, or a wiki or homepage on GitHub. Ideally, we should also have analyzed the quality of the documentation but this would be too complex to do. However, if it finds a README it checks if the size of the file is more than 300 bytes to exclude the really short ones.

Initialization Script

- **Trust Criterion 26**: The component does not contain installation scripts.

We check if the package or any of the dependencies contains an initialization script by searching through the repository. The files we look for are "init.ps1", "install.ps1", and "uninstall.ps1".

Known Vulnerabilities

- **Trust Criterion 20**: The project does not have reported security vulnerabilities.
- **Trust Criterion 21**: There is no history of prior harmful effects associated with the component.

To look for known vulnerabilities we check if we can find the package version being checked or any prior version in the OSV database or NuGet database. These databases contain a list of vulnerable NuGet packages.

License

- **Trust Criterion 7**: The component provides a standard or well-written license.

We check both NuGet and GitHub to see if the packet has a known license. The list of known licenses is taken from Software Package Data Exchange (SPDX) [69]. In addition to checking for licenses, we also warn if any package uses a license considered high or medium risk. This ranking of the different licenses is based on a risk evaluation provided by Synopsys [70].

Open Issues

- **Trust Criterion 16**: The component's maintenance lifecycle is up to date.

- **Trust Criterion 17**: There are a small number of open issues, and they are not very old.

For open issues we check to make sure that the project has at least 1 open issue. In addition, we warn if the number of open pull requests is more than 60% of the total number, or if less than 30% of the open pull requests have been updated over the last year. We also warn the user if the package has less than 30 issues in total.

Open Pull Requests

- **Trust Criterion 16**: The component's maintenance lifecycle is up to date.

The thresholds and check of open pull requests are almost identical to the test of open issues except that it warns if there are less than 10 pull requests in total. We decided the limit to 10 instead of 30, as the number of pull requests usually are lower than the number of issues.

Popularity

- **Trust Criterion 2**: The component is widely used/popular.

We use several different features to evaluate the popularity. On NuGet, we check if the project has less than 10 000 downloads, and we warn if less than 10 NuGet packages or GitHub repositories are using the package. On GitHub, we check if the package has less than 2 GitHub stars, or if it has no forks or watchers.

Verified Prefix

- **Trust Criterion 30**: The package has some verification from the package provider that makes it less deceptive in its identifying properties.

Checks NuGet to see if the package owner has reserved the package name prefix.

Widespread Use

- **Trust Criterion 1**: The component has been in widespread use for a considerable amount of time.

To check for widespread use we look at both the versions of the package and the number of downloads. We check that the package's oldest version is at least older than 1 year. Also, we check that the total download count on NuGet for the 10 latest versions is more than 100 000. If the package has less than 10 prior versions the user is warned.

4.3.4 Trust criteria not added

Due to time constraints, lack of available data, or not being prioritized, we were not able to add all the trust criteria found in Table 2.2 to the AutoTrust tool. The list of the trust criteria not added is presented here with a short description of why:

- **Trust Criterion 4**: The company you are working for is already using the package in another project.
  - This could be done by scanning through the company's source code, but we were not allowed to access all of Visma's source code.
- **Trust Criterion 5**: The component has a software certification from a certified provider.
  - There is no standard way that NuGet supports third-party package certifications, that we found.
- **Trust Criterion 6**: The component provides a hash and signature that can be used to make sure that the software has not been tampered with.
  - NuGet supports signatures with verification as default and blocks the installation if a package has been tampered with by someone [71].
- **Trust Criterion 11**: The maintainers of the component are not overloaded.
  - NuGet has information about how many packages a maintainer owns which could indicate if they are overloaded. However, we did not find this data to be extensive enough to answer the TC. It was also ranked low in the replies to the Visma Questionnaire seen in section B.3, as it had a median and mode of 3 indicating that the respondents only considered it "Not necessary, but insightful" and none of the respondents considered it "Crucial".
- **Trust Criterion 12**: The maintainers of the component are using a programming language that they are familiar with.
  - This is hard to evaluate as one would have to evaluate all the earlier projects of the maintainer. It was hard to automatically evaluate if the maintainers were using good practices and standards for the programming languages C#, F#, or Visual Basic.
- **Trust Criterion 13**: No maintainer accounts are associated with an expired email domain.
  - As the email of users is not displayed in NuGet this was not possible to check. The email registered on GitHub could have been used to evaluate this, but there is no way to verify that the email address is the same.
- **Trust Criterion 14**: The package has not changed ownership recently.
  - We found no way of checking who the previous owner of a package was or if the owner had changed at all.
- **Trust Criterion 15**: The maintainers, owners, and suppliers of the component are trustworthy.

    - The data from NuGet and GitHub was not extensive enough for us to be able to consider this.

- **Trust Criterion 18**: The developers of the component are using automated code analysis to review the code.
    - There is no information stating if automated code analysis has been used, so we would have had to check for indicators in the README or codebase to see if they might have used it. We believe this approach would lead to many false positives.
- **Trust Criterion 19**: The code reviews in the project are of high quality.
    - Checking this would include checking and evaluating the pull requests on GitHub. We anticipate that evaluating pull requests would be too time-consuming, and therefore not feasible.
- **Trust Criterion 24**: The size of the repository.
    - It was ranked low in the replies to the Visma Questionnaire seen in section B.3, as it had a median and mode of 3. This indicates that the respondents only considered it "Not necessary, but insightful" and none of the respondents considered it "Crucial".
- **Trust Criterion 27**: The component has a name that does not resemble that of a popular package.
    - We believe a typosquatting check would potentially slow down the performance quite extensively. For instance, we tried the OSSGadget's tool OSS-Find-squats mentioned in subsection 2.6.3, and it took 4 minutes and 25 seconds to run for the Newtonsoft.json package.
- **Trust Criterion 28**: There is no difference between the source code and the package.
    - NuGet distributes packages that contain .dll-files which are binary files. NuGet does not support reproducible builds by default[1], so we deemed this too challenging to evaluate.
- **Trust Criterion 29**: The source code is possible to access.
    - NuGet allows the package owners to link to the source code. However, there is no commonly used way of verifying that the provided URL is linked to the correct source code. Purposely including the wrong URL was done in the attack previously referenced in section 4.2.1.

4.3.5 Security Risk Score

All the implemented trust criteria validators are important for assessing the risk of the package. However, there is a difference in how influencing each validator

[1] https://github.com/dotnet/sdk/issues/2679

should be for the overall risk assessment of the package. We, therefore, decided to assign each validator an individual risk-influencing score. The risk influencing score given to the different analyzers is based on the initial questionnaire sent out to Visma, described in section 4.3.2, and the conversations with the two security engineers from Visma. The different scores given to the validators are shown in Table 4.1, where 1 is the lowest risk and 10 is the highest. We used the range 1 to 10 as it is sufficient to allow for nuance between the validators.

Table 4.1: Risk-influencing score of each analyzer.

Trust Criteria Validators	**Risk-influencing Score (1-10)**
Age	6
Analyzers	3
Contributors	3
Deprecated Package	10
Deprecated Dependencies	10
Direct and Transitive Dependencies	5
Documentation	5
Initialization Script	8
Known Vulnerabilities	10
License	7
Open Issues	3
Open Pull Requests	3
Popularity	7
Verified Prefix	7
Widespread Use	6

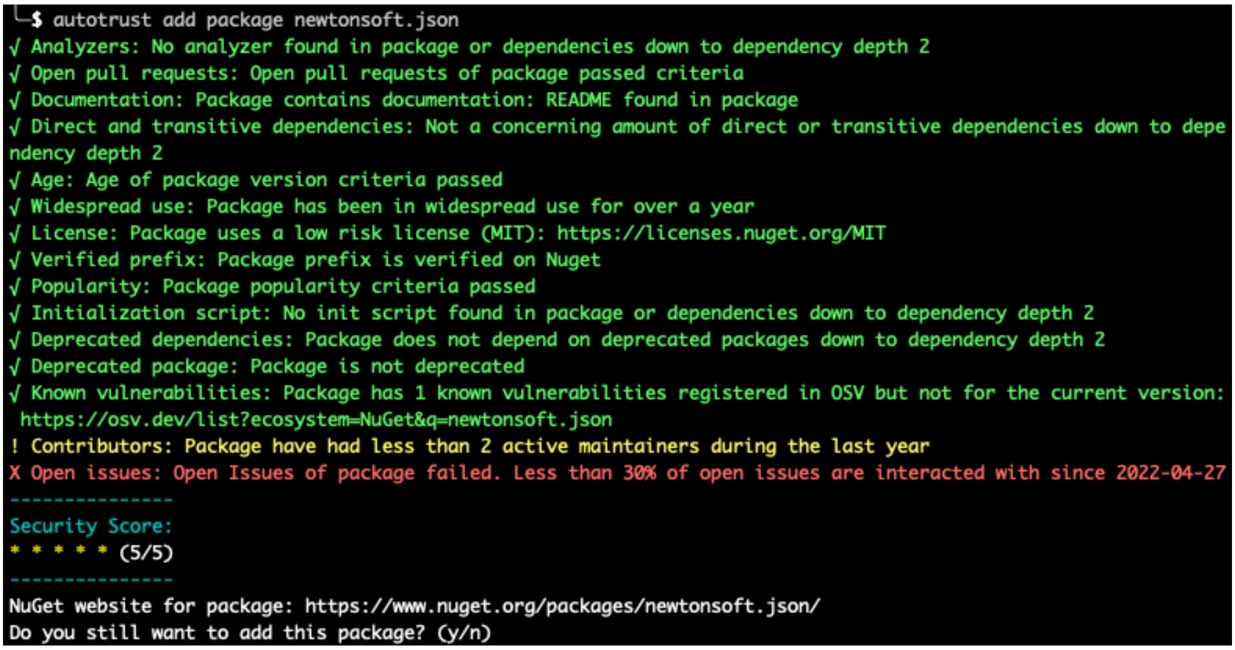
The package being analyzed is ranked based on a discrete score system from 1 to 5 stars, with 1 indicating high risk and 5 indicating low risk. The results of this ranking provide a quick overview of the package's overall risk level. We chose to use stars because they are easily comprehensible. Using stars helps users understand that a package with a rating of 5 stars is not necessarily completely risk-free, as stars often are used to convey an impression and not be a guarantee.

The star rating for a package is determined by the combined risk-influencing score of the validators, which is adjusted based on whether the validators receive a pass, warning, or failure. Validators that pass contribute the full risk-influencing score, whereas those with warnings have their score halved before being included in the total. The validators that fail are not contributing to the total. The total validator score is then divided by the maximum achievable score to obtain a percentage. This percentage is subsequently mapped to a star rating based on the following thresholds:

- 90-100%: 5 stars
- 80-89%: 4 stars
- 60-79%: 3 stars
- 40-59%: 2 stars
- 0-39%: 1 star

4.3.6 Finished Artifact

The output presented to the user when analyzing the "Newtonsoft.json" package is shown in Figure 4.4. The validators are sorted based on their pass, warning, or fail status, as well as their risk-influencing score. The failed validators are placed at the bottom, followed by the warning validators in the middle, and the passed validators at the top. Within the grouping of passed, warning, and failed validators, those with the highest risk-influencing score are positioned at the bottom. The risk-influencing score order and the criteria grouping were chosen such that the high-risk failed validators will not scroll outside the terminal window. At the bottom of the output, the overall security score with the number of stars is displayed.



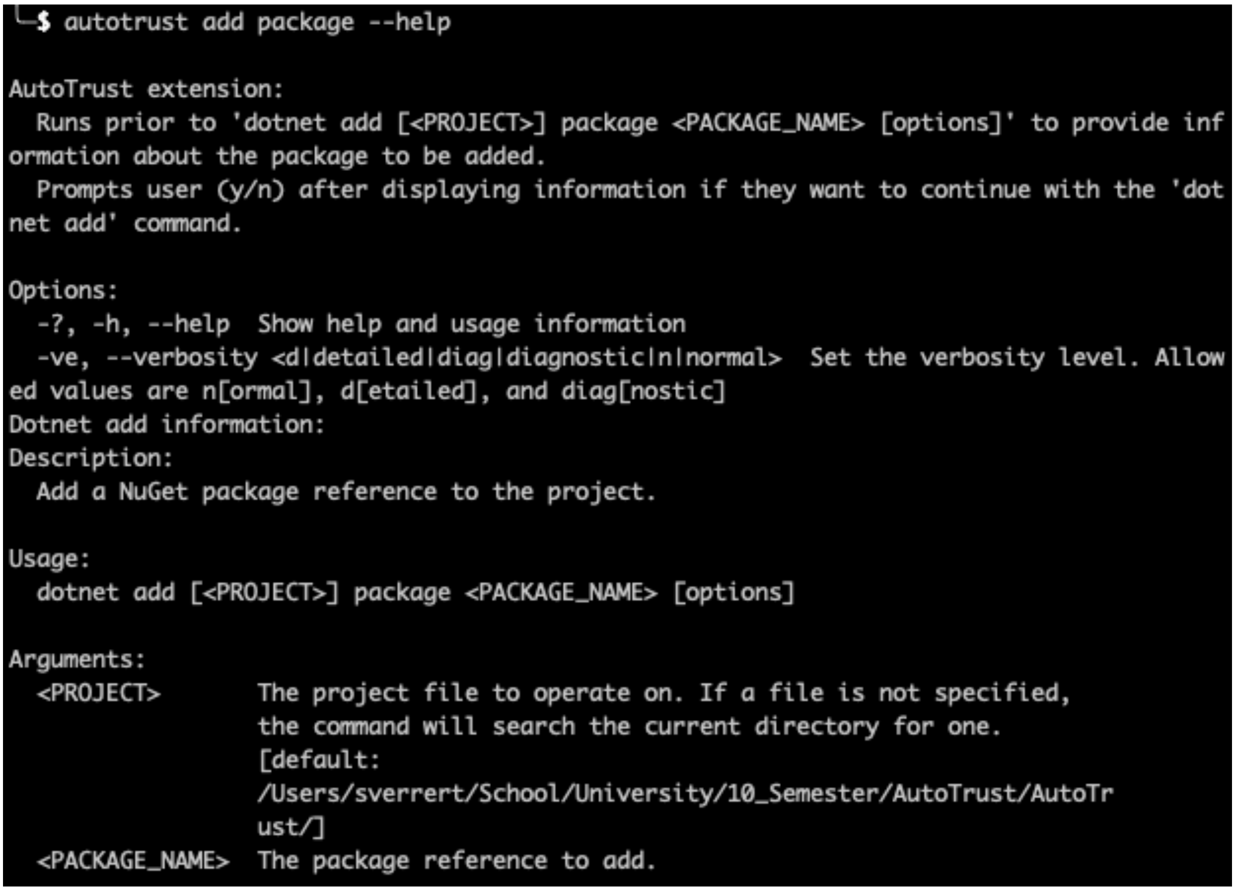
```
└$ autotrust add package newtonsoft.json
√ Analyzers: No analyzer found in package or dependencies down to dependency depth 2
√ Open pull requests: Open pull requests of package passed criteria
√ Documentation: Package contains documentation: README found in package
√ Direct and transitive dependencies: Not a concerning amount of direct or transitive dependencies down to depe
ndency depth 2
√ Age: Age of package version criteria passed
√ Widespread use: Package has been in widespread use for over a year
√ License: Package uses a low risk license (MIT): https://licenses.nuget.org/MIT
√ Verified prefix: Package prefix is verified on Nuget
√ Popularity: Package popularity criteria passed
√ Initialization script: No init script found in package or dependencies down to dependency depth 2
√ Deprecated dependencies: Package does not depend on deprecated packages down to dependency depth 2
√ Deprecated package: Package is not deprecated
√ Known vulnerabilities: Package has 1 known vulnerabilities registered in OSV but not for the current version:
 https://osv.dev/list?ecosystem=NuGet&q=newtonsoft.json
! Contributors: Package have had less than 2 active maintainers during the last year
X Open issues: Open Issues of package failed. Less than 30% of open issues are interacted with since 2022-04-27
---------------
Security Score:
* * * * * (5/5)
---------------
NuGet website for package: https://www.nuget.org/packages/newtonsoft.json/
Do you still want to add this package? (y/n)
```

Figure 4.4: Output from AutoTrust for the Newtonsoft.Json package.

4.3.7 Optional Flags

The output displayed in Figure 4.4 shows AutoTrust's normal behavior, however, we have added some optional flags that are used to give the user more details. The two flags we included were a "--help" flag and a "--verbosity" flag. The verbosity flag has three levels with the options "normal", "detailed" and "diagnostic". We tried to follow the standard syntax for .NET commands.

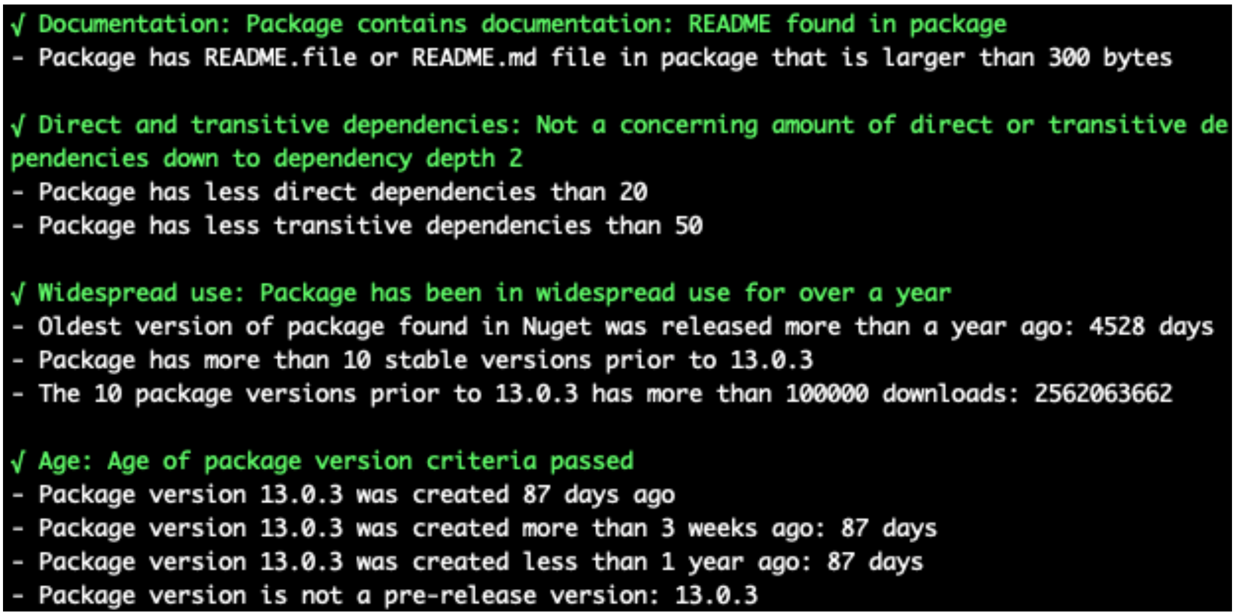
As seen in Figure 4.5, the "--help" flag displays information to the user about what the tool does and how they can use it. It also includes the information that is normally sent from .NET when using the "--help" flag. For improved user experience, abbreviated forms are also supported through "-?" and "-h".



```
└─$ autotrust add package --help

AutoTrust extension:
  Runs prior to 'dotnet add [<PROJECT>] package <PACKAGE_NAME> [options]' to provide inf
ormation about the package to be added.
  Prompts user (y/n) after displaying information if they want to continue with the 'dot
net add' command.

Options:
  -?, -h, --help  Show help and usage information
  -ve, --verbosity <d|detailed|diag|diagnostic|n|normal>  Set the verbosity level. Allow
ed values are n[ormal], d[etailed], and diag[nostic]
Dotnet add information:
Description:
  Add a NuGet package reference to the project.

Usage:
  dotnet add [<PROJECT>] package <PACKAGE_NAME> [options]

Arguments:
  <PROJECT>       The project file to operate on. If a file is not specified,
                  the command will search the current directory for one.
                  [default:
                  /Users/sverrert/School/University/10_Semester/AutoTrust/AutoTr
                  ust/]
  <PACKAGE_NAME>  The package reference to add.
```

Figure 4.5: Output from AutoTrust when executed with a help flag.

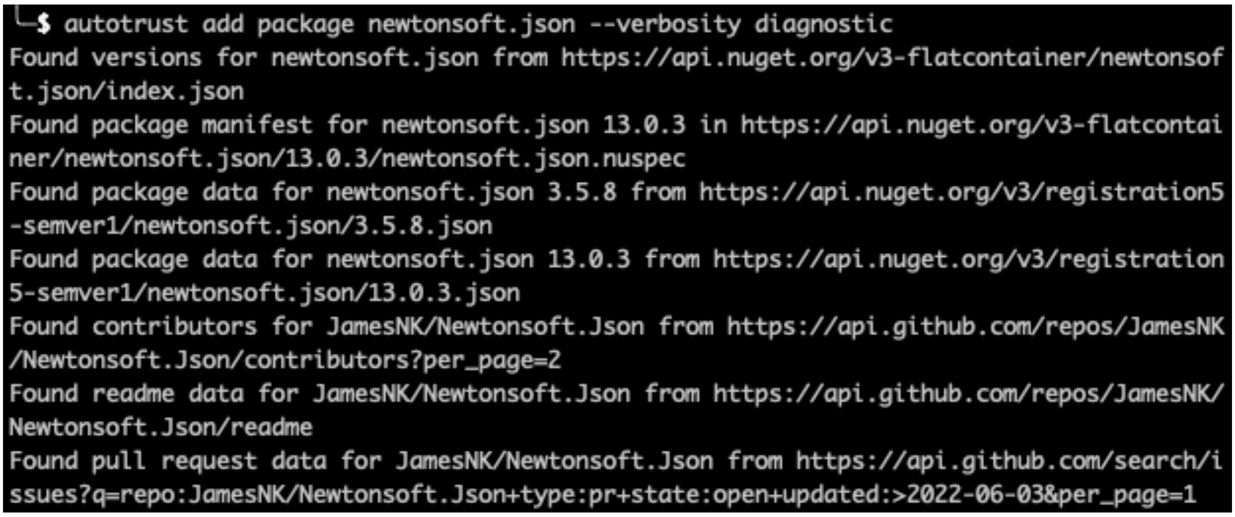
The other flag we added was a "--verbosity" flag used to provide more information to the user. The output from running AutoTrust with the flag is shown in Figure 4.6. It takes three different inputs "normal", "detailed", and "diagnostic" that changes how much additional detail is provided by AutoTrust. Running the tool with the "--verbosity normal" level is the same as running without specifying a verbosity flag, as can be seen in Figure 4.4. The "--verbosity detailed" level includes information about what is used to decide if each separate validator passed, warned, or failed.



```
√ Documentation: Package contains documentation: README found in package
- Package has README.file or README.md file in package that is larger than 300 bytes

√ Direct and transitive dependencies: Not a concerning amount of direct or transitive de
pendencies down to dependency depth 2
- Package has less direct dependencies than 20
- Package has less transitive dependencies than 50

√ Widespread use: Package has been in widespread use for over a year
- Oldest version of package found in Nuget was released more than a year ago: 4528 days
- Package has more than 10 stable versions prior to 13.0.3
- The 10 package versions prior to 13.0.3 has more than 100000 downloads: 2562063662

√ Age: Age of package version criteria passed
- Package version 13.0.3 was created 87 days ago
- Package version 13.0.3 was created more than 3 weeks ago: 87 days
- Package version 13.0.3 was created less than 1 year ago: 87 days
- Package version is not a pre-release version: 13.0.3
```

Figure 4.6: Output from AutoTrust for the Newtonsoft.Json package when executed with a verbosity detailed level.

The "--verbosity diagnostic" level includes the same information as the "detailed" level, but also information about if the different API calls return success or an error message. The information about the API calls will be written first, and then the information from the "--verbosity detailed" level will follow below. An example of some API calls can be seen in Figure 4.7.



```
└$ autotrust add package newtonsoft.json --verbosity diagnostic
Found versions for newtonsoft.json from https://api.nuget.org/v3-flatcontainer/newtonsof
t.json/index.json
Found package manifest for newtonsoft.json 13.0.3 in https://api.nuget.org/v3-flatcontai
ner/newtonsoft.json/13.0.3/newtonsoft.json.nuspec
Found package data for newtonsoft.json 3.5.8 from https://api.nuget.org/v3/registration5
-semver1/newtonsoft.json/3.5.8.json
Found package data for newtonsoft.json 13.0.3 from https://api.nuget.org/v3/registration
5-semver1/newtonsoft.json/13.0.3.json
Found contributors for JamesNK/Newtonsoft.Json from https://api.github.com/repos/JamesNK
/Newtonsoft.Json/contributors?per_page=2
Found readme data for JamesNK/Newtonsoft.Json from https://api.github.com/repos/JamesNK/
Newtonsoft.Json/readme
Found pull request data for JamesNK/Newtonsoft.Json from https://api.github.com/search/i
ssues?q=repo:JamesNK/Newtonsoft.Json+type:pr+state:open+updated:>2022-06-03&per_page=1
```

Figure 4.7: Output from AutoTrust for the Newtonsoft.Json package when executed with a verbosity diagnostic level.

The "--verbosity" flag also supports abbreviated forms like "-ve" with input "n", "d", or "diag". The normal abbreviation following .NET syntax would be "-v", but as this was already used to specify the package version we opted to use "-ve".

4.3.8 Justify and Reflect

We believe that it made sense to only extend the "dotnet add package"-command in the CLI and not the options of adding packages directly from the Visual Studio IDE. This helped to narrow the scope of the project. It also makes for a solution that is more similar across platforms as the option for adding packages from Visual Studio IDE is different for different operating systems [72]. In addition, making a more general artifact can inspire similar solutions for other programming languages and their package managers. The issue with this decision is that some users might prefer using the Visual Studio IDE, and adding packages using this method leaves them unprotected.

There are multiple code hosting platforms that developers can use to host their open-source code used for NuGet packages. While manually assessing what information was available on NuGet, we saw that most of the NuGet packages that were open source, used GitHub. GitHub is also the largest code hosting platform for Git so we thought it made sense to focus on GitHub first. In the future, it would be beneficial to check other code hosting platforms such as GitLab in case the NuGet package code is hosted there.

To find more reported vulnerabilities, we made the decision to incorporate OSV as an additional vulnerability database alongside NuGet. We chose to use OSV as it is trustworthy, has data about NuGet vulnerabilities, and does not need the user to create tokens. There are many vulnerability databases and in the future, it might make sense to extend the options or have the user choose if they would use the vulnerability database of their choice.

Finding good validators and thresholds is challenging. Finding validators that help to distinguish benign, poorly written, and malicious packages that work for popular and less popular packages is not an easy task. Good thresholds for the validators might also be affected by how willing the user is to take risks. An option to support user configuration for enabling and disabling validators and changing thresholds was discussed. However, keeping the validators and their thresholds predefined made it easier to do an equal evaluation during this research process. It would also probably be hard for teams to find proper thresholds that fit their risk appetite. This might lead to much time spent on discussing thresholds that could be saved if AutoTrust is kept predefined.

Targeting Requirements

During the design and development phase, we made some decisions to try to meet the functional requirements presented in section 4.2. FR 1-4 were met by creating a CLI tool that runs prior to the "dotnet add package"-command using data from NuGet. To target FR-5 there is no caching done, as this might introduce the possibility that a cache becomes stale and does not have new data that might be less favorable for the package ranking. To target FR-6 an option to add a "--verbosity" flag for extra information to the user was added, as shown in subsection 4.3.7.

To target QA-1, regarding usability, we have taken multiple actions. We added a README to help with the installation and description of the artifact and the checks. The syntax for adding packages and using flags was made to be as similar as possible to the original "dotnet add package"-command. We support a help flag to allow users to get extra information about the package, and as mentioned above, a verbosity flag to allow users to get more information about each check. We have used multiple visual cues using both color and symbols to make it easier to assess the packages quickly. The security score was added to allow users to assess which of the checks are more crucial. Users can also use the alias command in the terminal to change the regular "dotnet add package"-command to run the "autotrust add package" functionality. By doing this they do not have to change their habit and can run "dotnet add package" to use AutoTrust.

To enhance QA-2 about portability between different operating systems we have used .NET's console app template. This allows the creation of a command-line application for Windows, Linux, and Mac. We also made sure to test the application for the different operating systems as discussed in subsection 4.4.1. We tried to minimize the use of solutions that might not work on different operating systems and terminals such as using more complex symbols during loading or for the output. There are some minor differences when storing the GitHub token, but that is mostly linked to how Windows, Mac, and Linux store their environment variables differently.

To improve QA-3, dealing with security, we have only used packages provided directly in .NET, and made the tool such that the package is not downloaded without an affirmative response from the user. We were considering using the GitHub API Client Library for .NET[2], but decided not to so that we would stay true to the stated quality attribute. Using the library could have allowed users to get GitHub tokens by logging in instead of having the token as an environment variable. The project could also have its security improved by using more automated security tools. Examples of this could be to regularly run automatic code scanning and analysis or include them in a CI/CD pipeline to catch security issues.

To optimize QA-4 which concerns performance we decided to run the fetching of the data and the trust criteria validating in parallel. Using *Tasks.Whenall()* for parallel execution significantly improves the data fetching and the validating of the package's data.

To better the QA-5 dealing with AutoTrust's modifiability we tried to separate the logic for the different trust criteria validators, the sorting of the trust criteria, and fetching of relevant data for the trust criteria. We have created a common interface for all the trust criteria and running the task in parallel allows adding more checks without a large time penalty.

[2] `https://github.com/octokit/octokit.net`

Comparing with npq, OSSGadget, and OpenSSF Scorecard

As introduced in subsection 2.6.3, there are three tools, npq, OSSGadget, and OpenSSF Scorecard that are somewhat similar to AutoTrust in that they can be used to analyze packages before they are installed. In Table 4.2 we have summarized the equalities and differences between the tools. The validators column in this table refers to the different checks being performed by the tools.

Table 4.2: Comparison of AutoTrust, npq, OSSGadget, and OpenSSF Scorecard.

Validators	AutoTrust	npq	OSSGadget	OpenSSF Scorecard
Age	●	●		●
Author		●		
Analyzers	●			
Binary-Artifacts				●
Branch-Protection				●
Characteristic			●	
CI-Tests				●
CII-Best-Practices				●
Code-Review				●
Contributors	●		●	●
Cryptographic Implementations			●	
Dangerous-Workflow				●
Dependency-Update-Tool				●
Deprecated Package	●			
Deprecated Dependencies	●			
Direct and Transitive Dependencies	●			
Documentation	●	●		
Expired Email Domain		●		
Fuzzing				●
Initialization Script	●	●		
Known Vulnerabilities	●	●		●
License	●	●		●
Malicious Code Scanning			●	
Obfuscated Text Scanning			●	
Open Issues	●		●	

Continued on next page

Table 4.2 – continued from previous page

Validators	AutoTrust	npq	OSSGadget	OpenSSF Scorecard
Open Pull Requests	●		●	
Packaging				●
Pinned-Dependencies				●
Popularity	●	●	●	
Repository URL	●*	●	●	
Static Code Analysis Tools				●
Security Issues			●	
Signed-Releases				●
Size of Repository			●	
Token-Permissions				●
Typosquatting			●	
Verified Prefix	●			
Widespread Use	●			

Based on the information in Table 4.2, the number of overlapping validators between the different tools in decreasing order is as follows:

- AutoTrust and npq have 6 out of the 34 validators in common.
- AutoTrust and OpenSSF Scorecard have 4 validators in common.
- AutoTrust and OSSGadget have 4 validators in common.
- OpenSSF Scorecard and npq have 3 validators in common.
- OSSGadget and npq have 2 validators in common.
- OSSGadget and OpenSSF Scorecard have 1 validator in common.

This shows that the overlap between the tools is quite small. AutoTrust and npq are the most similar, which is logical considering their shared purpose of pre-installation evaluation. Moreover, they employ a similar approach by utilizing validators and running them in parallel. AutoTrust does not have as much in common with OpenSSF Scorecard and OSSGadget which makes sense as these tools are not primarily built for pre-installation assessment.

Another difference between the tools is that AutoTrust and npq only use the package metadata while OSSGadget also inspects the source code. OpenSSF Scorecard does check the source code but only for binaries and if the dependencies are pinned. It is through the source code inspection that OSSGadget checks for malicious code, obfuscated text, characteristics, and cryptographic implementation. The source code inspection provides some extra insight but it also takes more time. In addition, AutoTrust, OpenSSF Scorecard, and OSSGadget calculate a security risk score of the package, which is not provided by npq.

OSSGadget has multiple validators that operate differently than the checks performed by AutoTrust, npq, and OpenSSF Scorecard as they perform a more direct analysis of the project code. For the "Characteristics" validator, OSS-characteristic identifies characteristics that are coding features of the package. Through this assessment, it identifies coding features to tell what the software is and what it does. Using OSS-health, OSSGadget looks for "Security issues" by searching through open and closed issues for security keywords to give the developer an understanding of the presence of security-related issues. For "Malicious code scanning" and "Obfuscated text scanning", OSSGadget uses OSS-detect-backdoor and OSS-defog to scan the source code looking for malicious code or obfuscated text.

Unlike the other tools, OpenSSF Scorecard performs all its checks on GitHub or GitLab repositories. As a consequence, by looking more in-depth at the best security practices on these platforms OpenSSF Scorecard has many validators that do not overlap with the other tools. Also, as OpenSSF Scorecard has an open API the other tools could potentially query this API and benefit from OpenSSF Scorecard's in-depth checks for the code hosting platforms.

Both AutoTrust and OpenSSF Scorecard fetches data about known vulnerabilities from OSV, while npq looks for known vulnerabilities from Snyk's database. Another difference between the tools is how npq, OSSGadget, and AutoTrust consider the "Repository URL". The OSSGadget tool OSS-find-source helps to locate the GitHub source code for a package. The npq tool displays a warning if the npm package does not have a valid and working repository URL. AutoTrust does not have a separate validator to check if a link to the repository URL is provided. It displays a warning if it does not find a link to a GitHub repository, and it fails the checks in the validators that are dependent on GitHub data. That is why we marked "Repository URL" for AutoTrust with a "*" in Table 4.2.

4.4 Demonstrate Artifact

After AutoTrust was designed and developed we wanted to demonstrate that it can in fact solve some part of the explicated problem in section 4.1 before doing further evaluations. We decided that testing the tool on 100 NuGet packages would give an indicator of whether AutoTrust is a feasible solution to the problem of automatically assessing packages prior to installing them. We also wanted to verify that QA-2 and QA-4 regarding portability and performance were met. This was done by testing the tool on different operating systems and measuring the execution time for 10 NuGet packages.

4.4.1 100-Packages Demonstration

We chose to test the 50 most downloaded packages on NuGet, 40 random packages, and 10 problematic packages we found manually that had one or multiple issues. The random packages were chosen by generating 40 random three-letter sequences and inserting them in the search bar on NuGet. The result was sorted by relevance, and we picked the first result that was shown. The 10 problematic packages that were specifically selected were chosen because they were either deprecated, used deprecated dependencies, had known vulnerabilities, had initialization scripts, used a high-risk license, had known vulnerabilities for previous versions, missed a license, or used a high-risk license. We chose 2 deprecated packages and 2 packages with known vulnerabilities as they had the highest risk-influencing score possible. All the packages with name, version, how they ranked for the different criteria, and their security risk score is shown in section B.6. The overall ranking of the different packages can be seen in Figure 4.8, where the green section shows the top 50 most downloaded packages, the yellow shows the 40 random packages, and the red section shows the 10 problematic packages that were selected because of issues.

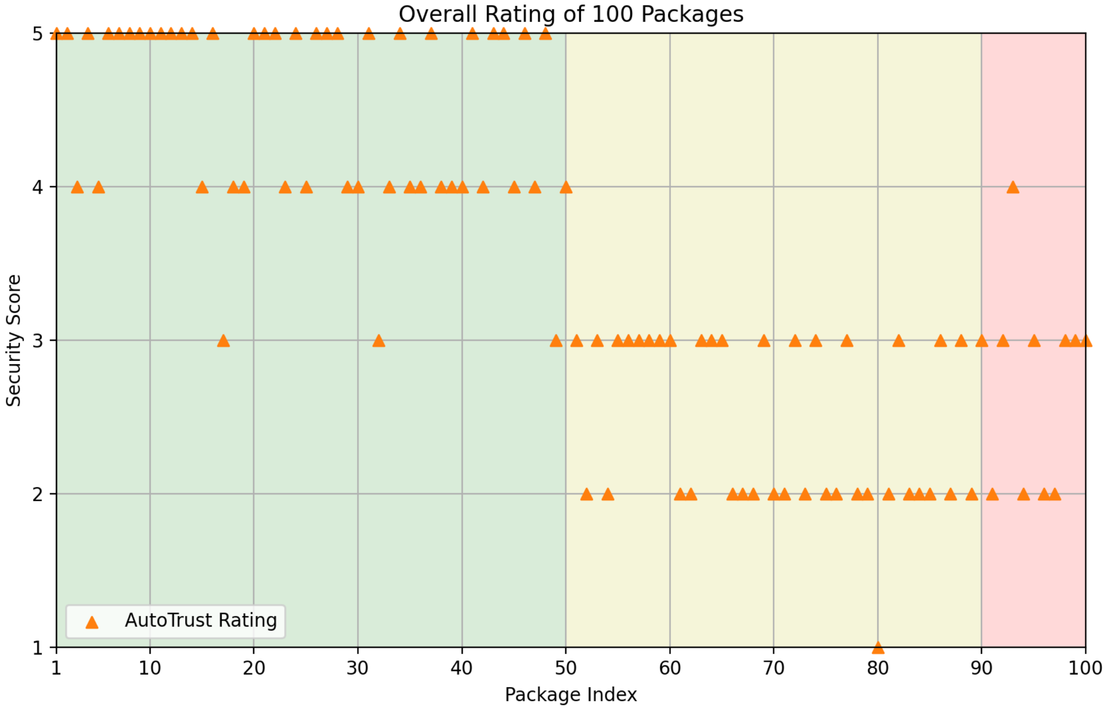


Figure 4.8: Overall rating of the 100 packages. The green area: top 50 packages, yellow area: 40 random packages, red area: 10 problematic packages.

The security risk score is ranked between 1-5 stars and is qualitative ordinal data. The median ranking for all the packages was 3 and the mode was 5. The median and mode security risk score for the top downloaded packages was 5, for the random packages it was 2, and for the problematic packages, it was 3. These results are presented in Table 4.3.

Table 4.3: Median and mode of the package ranking for the different package categories made by AutoTrust.

Package category	Median	Mode
Total packages	3	5
Top 50 packages	5	5
Random packages	2	2
Problematic packages	3	3

We see that the top 50 packages scores higher than the random and problematic packages. For the problematic packages, the tool was able to find and report all the issues we had manually found. AutoTrust gave a higher security risk score to the problematic packages than the random packages. This might be due to the fact that two of the problematic packages were Microsoft packages and more of the problematic packages had their source code available on GitHub. Another factor is that none of the random packages were ranked above 4 while one of the random packages PTJK.GenericRepoSpecPattern received a score of 1 star.

4.4.2 Time Demonstration

We also wanted to test whether AutoTrust has fulfilled the goals of QA-2 and QA-4, regarding portability and performance. When testing AutoTrust we used two different laptops where one was running Windows 11 and Windows Subsystem for Linux (WSL) and the other macOS. The specifications of the laptops can be found in section B.7.

We tested the 10 most downloaded packages on NuGet by running "autotrust add package" and "dotnet add package" using the "Measure-Command" for Windows and the "time" command for macOS and Linux. The arithmetic mean of the 10 execution times from running each command is displayed in Table 4.4. The full results in seconds can be found in section B.7.

Table 4.4: Average execution time of AutoTrust and .NET install package command.

Operating System	Average Time AutoTrust	Average time .NET
Windows	5.43 s	2.08 s
MacOS	5.66 s	3.38 s
Linux	4.89 s	2.87 s

In addition to the arithmetic mean, we calculated the ratio between the .NET and AutoTrust commands for the three operating systems. The ratio can be seen in Figure 4.9.

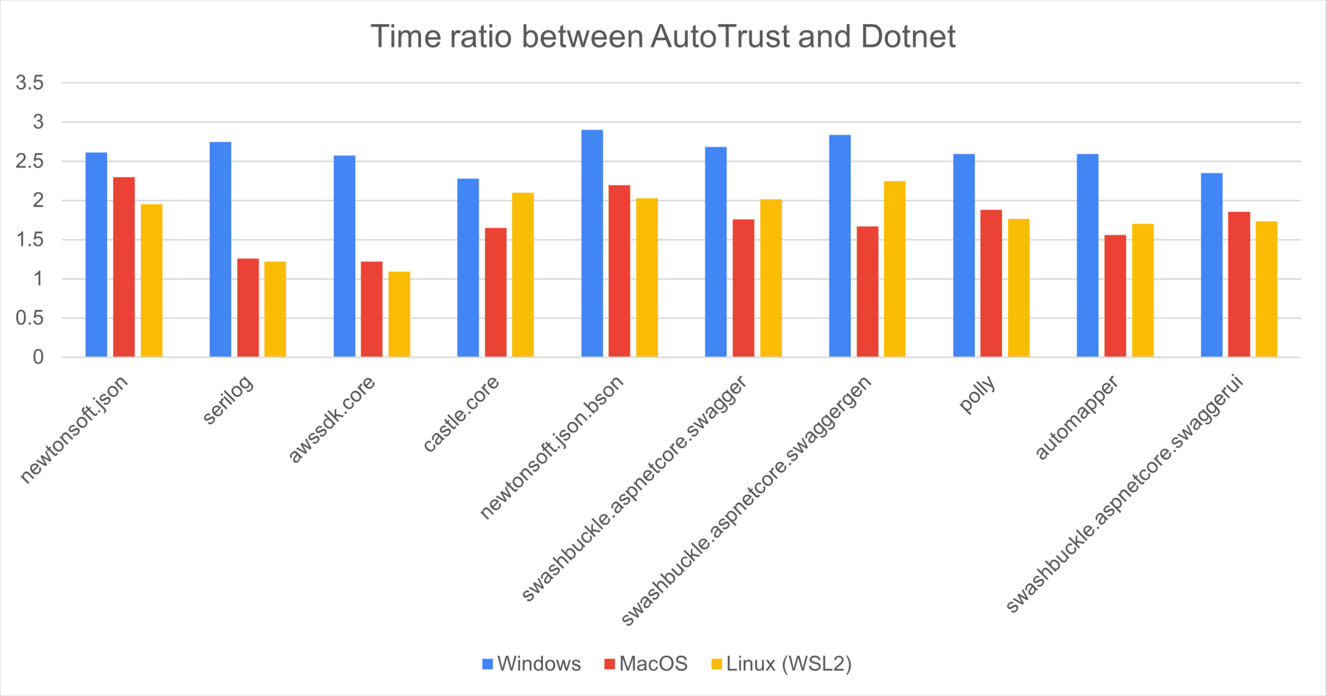


Figure 4.9: The ratio of execution times for AutoTrust divided by Dotnet using three different operating systems.

From this demonstration, we see that we were able to use AutoTrust with Windows, macOS, and Linux and that the tool did not use more than three times as much time running "autotrust add package" as "dotnet add package", which was one of the performance goals. Figure 4.9 also shows that the ratio difference was largest when using the Windows operating system. A reason for this might be that .NET is developed by Microsoft and is therefore better optimized for Windows.

We also tried running the scripts using different shells on the various operating systems to see if they gave different results. We tried the terminals cmd.exe and Powershell for Windows, along with Bash and Zsh for macOS and Linux. However, no significant differences were observed among the different shells.

4.5 Evaluate Artifact

The "Evaluate artifact" phase is the last in the design science methodology. Evaluating the artifact is part of the stated goal of this thesis. After creating an artifact it is needed to evaluate if it is useful in solving the problem it aims to solve. To evaluate AutoTrust we have decided to focus on 4 of the 6 evaluation goals presented in section 3.1.2 as they are the ones focusing on the artifact itself:

- **Evaluation Goal 1:** Evaluate if the artifact is effectively solving the problem.
- **Evaluation Goal 2:** Evaluate the functional and non-functional requirements proposed used on the artifact.

- **Evaluation Goal 3:** Compare the artifact to other similar artifacts that intend to solve the same or a similar problem.
- **Evaluation Goal 4:** Investigate the side-effects such as unintended or harmful effects of the artifact.

During the earlier phases of the project, we have already seen indicators that AutoTrust has either completely or partially fulfilled these evaluation goals. However, we still needed further insight to evaluate AutoTrust properly. In order to obtain this knowledge we have performed user tests, interviews, and experiments. We decided to divide this evaluation phase into three separate studies to more accurately answer these evaluation goals.

The first evaluation we did was an experiment with computer science students, the second was an experiment comparing AutoTrust with OpenSSF Scorecard, and the third was general user testing and interviews with Visma employees. All of the evaluations are ex-post evaluations as they were done after AutoTrust was created. The result of these evaluations will be described below, followed by an analysis of the five artifact evaluation goals.

4.5.1 Student Experiment

We wanted to use the student experiment to evaluate if the artifact has fulfilled evaluation goals 1 and 3, by comparing the manual assessment made by software developers to the automatic assessment made by AutoTrust. Since AutoTrust is a tool that can be used by developers with varying knowledge about software supply chain security, we believe that it is a fair evaluation to compare the AutoTrust package evaluation to the evaluation done by computer science master's students in their final year. The evaluation consisted of having the students manually assess three packages: Serilog (2.12.0), Analytics (3.8.1), and PTJK.GenericRepoSpecPattern (1.0.1). These packages were chosen since we wanted packages with evenly distributed AutoTrust ratings across the entire scale. Serilog got 5 stars, Analytics got 3 stars, and PTJK.GenericRepoSpecPattern only 1 star. The packages were chosen from the 100-packages demonstration mentioned in subsection 4.4.1. In order to reduce the likelihood of the order influencing the score, the packages were presented to the students in varying orders.

As described in subsection 3.2.2, the second to last step of experiment planning is evaluating the instruments used. The main instruments used were the students' own laptops with internet access, the online questionnaire, the NuGet website, and potentially other websites the students felt were valuable. The questionnaire worked as a guideline to help the students through the experiment. With the questionnaire, we attempted to direct the students on what to do, but not how to do it. The questionnaire worked as the only instrument used for data collection. The NuGet website was the primary source the students used to find information about the packages, but they also used other websites to search for additional information, in the same way a regular developer would.

During the experiment, we maintained close proximity to the students to provide immediate assistance and clarification in case they encountered any uncertainties. Also, this was an artificial evaluation since it took place in a contrived and artificial setting, and none of the students were actually intending to use the proposed packages in a project.

The experiment was divided into two parts to more precisely evaluate the two hypotheses. In the first part of the experiment, the students were asked to do a critical risk evaluation of the aforementioned NuGet packages. In the second part, the students filled out what criteria they used for evaluating the NuGet packages. When designing the questionnaire, we took this into consideration and divided it into two parts which can be seen in section B.4.

In the first part of the experiment, we wanted to see how the students would rank the packages on a scale from 1-5, where 1 is high risk and 5 is low risk. The ranking made by the students can be seen in Figure 4.10.

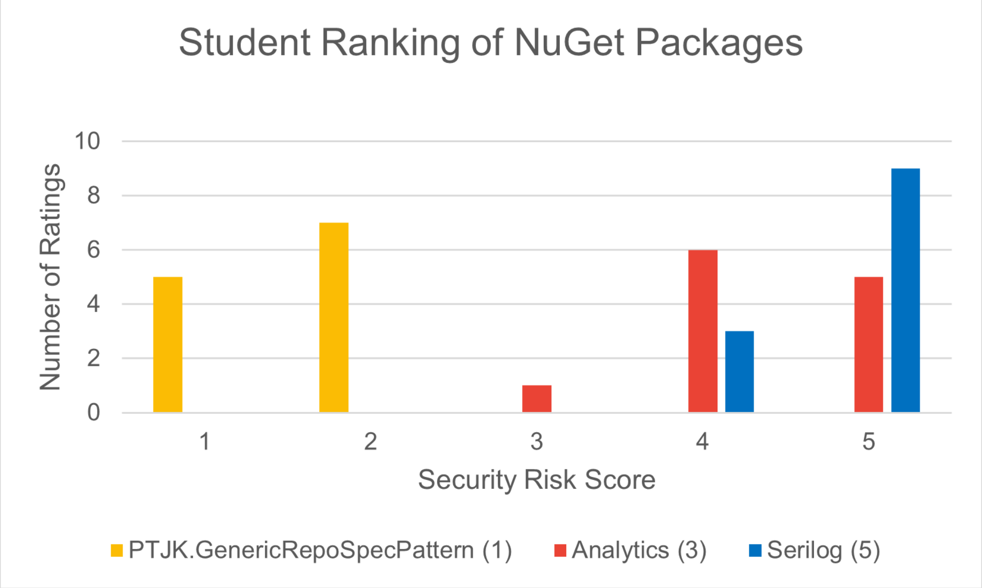


Figure 4.10: The students' ranking of the NuGet packages.

Compared to the ranking given by AutoTrust, there is a difference in the ranking for both Analytics and PTJK.GenericRepoSpecPattern as both were ranked higher by the students. From Figure 4.10 we also see that Analytics has a mode of 4 and PTJK.GenericRepoSpecPattern has a mode of 2, both of which are 1 higher than the ranking of AutoTrust. For the Serilog package, the rating given by the students was quite close to the rating from AutoTrust, with a mode of 5. Out of the total 36 ratings given, 15 of the package ratings were the same by the students and AutoTrust. Looking at the difference, the students rated 3 packages 1 below AutoTrust, 13 packages 1 above, and 5 packages 2 ratings above. In total, there were 21 ratings different from the AutoTrust rating, where 18 of them were higher. Based on these comparisons, in our view, the students considered the packages more trustworthy compared to AutoTrust.

To get more insight into the students' reasoning when ranking the packages, we asked the students to justify why they gave the specific ratings. The takeaways were that popularity was the most commonly used metric. Many of the students used search engines to search for articles and web pages to gain an overall impression of the package. The students also did consider the characteristics of the package. If the packages were made for what they considered more high-risk components, such as databases, it would affect their perceived risk. Examining the package's characteristics is a similar approach to what OSSGadgets does with their OSS-Characteristics and OSS-risk-calculator tools. Some of the students expressed that they lacked familiarity with using NuGet and were uncertain about which criteria to evaluate. One of the students even wrote that they thought the Analytics package might have a deprecated dependency, which it has, due to the fact that the package had not been updated in a long time, however, they were not able to find and report it.

In the second part of the questionnaire, we listed all the validators used by AutoTrust and asked if they had considered the same criteria. The number of students evaluating the same criteria as each validator can be seen in Figure 4.11. We can see that the criteria the students considered the most were popularity, widespread use, age of the package, and documentation. These metrics are clearly displayed on the NuGet website and are easy to understand, but it might be hard to evaluate how their values affect the risk.

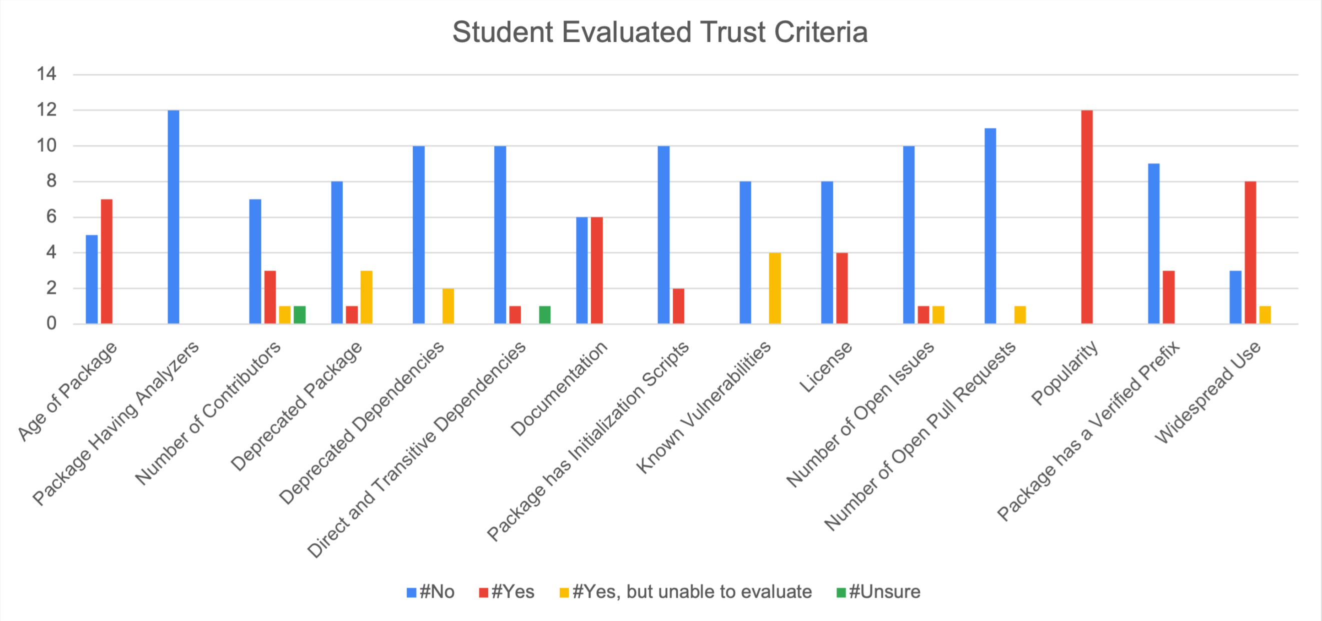


Figure 4.11: Number of students that considered the various TC in their risk evaluation of packages.

As the final question, we asked if they had considered some other criteria not on the list. The core finding from this question was that the students had manually assessed the owner information by taking into account factors such as the number of other packages published by the owner, the number of downloads those packages had, and whether the owner was an individual or a company. Looking at the individual responses, the participant who considered the most TC reviewed 11 out of the total 15 criteria and did not consider any additional criteria. The two participants considering the least only considered 2 of the TC. However, they also took into account the package owner, similar to trust criterion 15, resulting in a total of 3 criteria that were assessed.

Looking at the results from the student experiment we can see that the two hypotheses can be rejected. Hypothesis 1 stated that the students would provide a risk review that is equal to the AutoTrust review, which we can see from Figure 4.10 is not true. The students generally ranked the package higher than AutoTrust, which is evident in the evaluation of Analytics and PTJK.GenericRepoSpec-Pattern. Hypothesis 2 stated that the students would consider the same amount of trust criteria or more than AutoTrust when evaluating package risk. As we can see from Figure 4.11 this hypothesis was also rejected. Only 6 of the 15 criteria were considered by at least half of the 12 students. This also means that 9 of the 15 criteria were considered by less than half of the students. Furthermore, none of the participants examined and suggested more than two additional trust criteria.

Based on the experiment it seems that the students are less strict in their rating and they consider fewer criteria when evaluating packages. Since there is no ground truth regarding the correct security risk score, it is difficult to know if the score given by the students is more accurate than the AutoTrust ranking. However, the students considered fewer criteria which means they could miss indicators of risk that would influence the security risk of the dependency.

Validity of the Student Experiment

In any experiment, it is important to consider factors that may impact its validity. The four common factors to consider are conclusion-, internal-, external-, and construct validity [58]. Conclusion validity is concerned with the relationship between the treatment and the outcome. This means ensuring that a statistical relationship exists with a given significance. Threats to internal validity refer to the potential influences that may impact the causal relationship between the independent variable and the outcome, without the researcher's awareness. External validity pertains to the generalization of the results. For instance, if a causal relationship exists between the cause and effect of a construct, the question stands if the results can be applied to other contexts. Construct validity refers to whether an experiment is created in a way that allows the results to be applicable and representative of the underlying concept or theory being investigated. There are various threats to construct validity, including those related to the experimental design and social factors.

Conclusion validity might have been affected by the choice of packages. The packages chosen could have influenced the result of the study, as there was a significant variation in their security risk score. This might have led to the participants being able to distinguish the packages easier than if they had been more similar. The number of students who participated in the experiment was limited to only 12, and more participants could have strengthened the statistical power of the experiment. Also, the students were from a relatively heterogeneous group since they all were computer science master's students in their final year. The heterogeneity of the group may have impacted the results, as a more diverse group could have provided a wider range of answers.

The internal validity could have been affected negatively by the issue of maturation. The participants might have been more thorough in their evaluation of the first package than when they evaluated the last package because they get bored during testing.

For external validity, the limited knowledge of NuGet by the students could affect the generalization of the experiment. The students might know less about NuGet and what is possible to consider than developers regularly using the .NET ecosystem, which can have affected how well the assessment fits the real world.

In regards to construct validity, there might have been an issue with how the participants performed in a test environment compared to their normal behavior. We wanted to compare the process of doing manual assessment and automatic assessment before adding a new package to a project, but this experiment primes the participant to be more vigilant than if they would add the package normally. The fact that the candidates knew their actions were being observed and evaluated can have created a Hawthorne effect, leading to a potential modification of their normal behavior during the experiment.

We acknowledge the aforementioned issues and have implemented certain measures to enhance the validity of our study. To increase the construct validity, we changed the package order to allow all the different combinations to occur. This minimizes the risk of participants rating the same last packages based on impressions from rating the first package, as the order is rotating. Another factor this rotation solves is that the experiment is less affected by the participants becoming tired after the two first packages and doing a mediocre assessment for the last. We also made sure that the same information was given to all the participants by having the questionnaire guide them and not explaining the experiment in person.

In order to strengthen the conclusion validity of the study, we aimed to improve the reliability of the measures by remaining physically present near the students. This would allow them to seek clarification in case of confusion and ensure that their responses were based on accurate premises.

The final action we took to improve the external validity was having all the students use the same testing environment. Using the same environment minimizes the likelihood of participants being influenced differently by external factors in their environment.

4.5.2 Testing OpenSSF Scorecard

The second experiment, which involved testing OpenSSF Scorecard, was performed in order to help answer RQ-3. We wanted to compare AutoTrust to multiple similar tools for assessing software packages. However, OSSGadget had an issue with the OSS-risk-calculator[3] and npq is only made for the npm ecosystem. This left OpenSSF Scorecard as the best tool to compare AutoTrust with.

As described in subsection 2.6.3, OpenSSF Scorecard checks open source repositories to assess if the project is following security best practices. It has 18 checks for GitHub and working on supporting checks for GitLab with 13 already being implemented. Based on the checks and weights of these checks it ranks the security of an open source project as a decimal number between 0 and 10, where 10 is the best.

We decided to test the same 100 NuGet packages as described in section B.6. Most of the packages had their source code on GitHub, one package had it on BitBucket, and the rest did not have it linked on the NuGet website. Since none of the packages had their source code hosted on GitLab, the assessment was only done comparing OpenSSF Scorecard's GitHub ranking with AutoTrust's ranking of the NuGet package. The test was conducted by running the latest stable version of OpenSSF Scorecard which was v4.10.5-155-g1dff427. This evaluation was, as the student experiment, an artificial evaluation as it also takes place in an artificial setting. The ranking of all the packages using OpenSSF Scorecard can be found in section B.8.

For the instrumentation step of this experiment, the main instruments used were the OpenSSF Scorecard tool itself, and the documentation for how to install and use the tool. To ensure consistency and eliminate variations, we conducted the testing of all packages on the same day. This approach ensured that each package was evaluated using the same version of the tool, without any updates or modifications introduced between the checks. For guidelines on installing and using the tool, we used the README provided on the OpenSSF Scorecard project's GitHub page[4]. The tool has to be run through the CLI and there are two options for doing this. One option is to install OpenSSF Scorecard as a standalone, and the other option is to use it through a Docker container[5], which we ended up doing.

The resulting package ranking of all 100 packages by using OpenSSF Scorecard is presented in Figure 4.12.

[3] `https://github.com/microsoft/OSSGadget/issues/150`

[4] `https://github.com/ossf/scorecard`

[5] `https://www.docker.com/resources/what-container/`

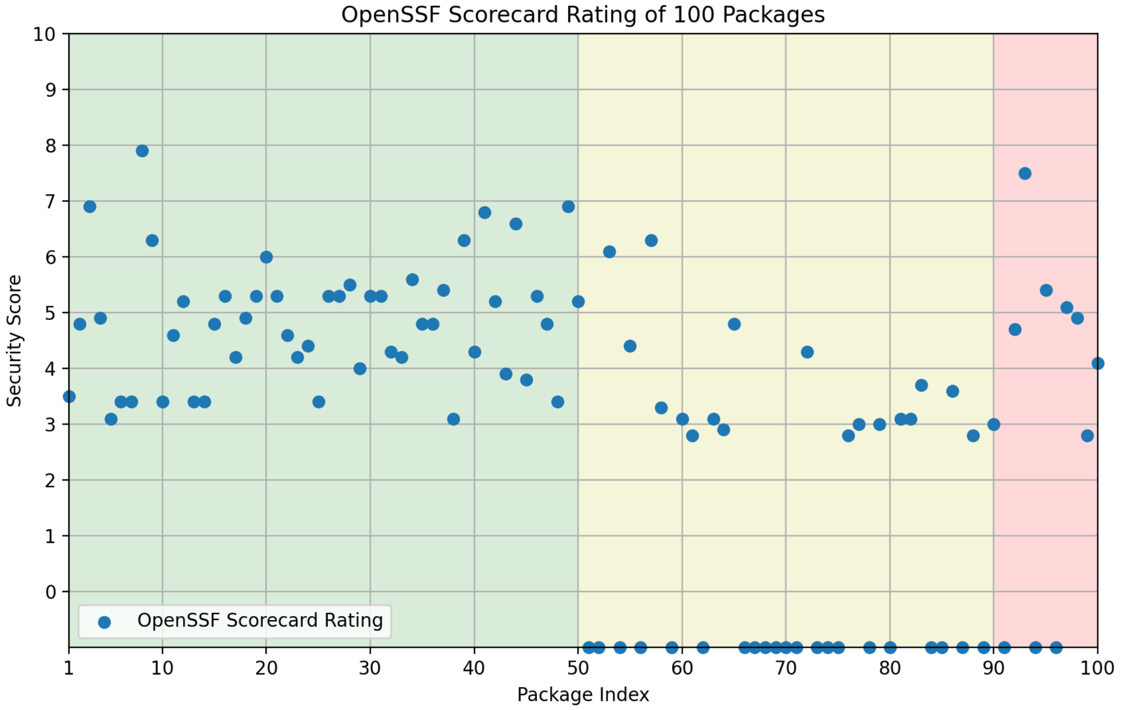


Figure 4.12: OSSF Scorecards rating of the 100 packages. The green area: top 50 packages, yellow area: 40 random packages, red area: 10 problematic packages.

There were 24 packages that could not be ranked by OpenSSF Scorecard since their source code was not on GitHub or GitLab. In Figure 4.12, we gave these packages a ranking of -1, as we still wanted to present them in the graph. The total median and mode score of the 76 remaining ranked packages is shown in Table 4.5, together with the individual rankings of the three categories. As not all packages had provided a GitHub URL to the repository, we could only rank 19 of the 40 random packages and 7 of the 10 packages with issues. Since the sample size for the problematic packages was only 7, none of the packages was rated the same, which led to the mode being the same as the median.

Table 4.5: Median and mode of the package ranking for the different package categories made by OpenSSF Scorecard.

Package Category	Median	Mode
76 Total packages	4.6	5.3
Top 50 packages	4.8	5.3
19 Random packages	3.1	3.1
7 Problematic packages	4.9	4.9

In Figure 4.13 we combined the ranking made by AutoTrust in subsection 4.4.1 and the ranking given by OpenSSF Scorecard. Note that due to the utilization of two distinct ranking scales for packages, with AutoTrust ranging with discrete values from 1 to 5 and the OpenSSF Scorecard ranging from 0 to 10, we have performed a rescaling of the OpenSSF Scorecard rankings to align with AutoTrust. This conversion makes it possible to use the chi-square test of independence and compare the package ratings of the two tools. The conversion is displayed in Table 4.6.

Table 4.6: Combining the package rating from AutoTrust and OpenSSF Scorecard.

Ranking	AutoTrust	OpenSSF Scorecard
5	5	8.1 - 10
4	4	6.1 - 8
3	3	4.1 - 6
2	2	2.1 - 4
1	1	0 - 2

Dotted vertical lines have been used in Figure 4.13 to visually connect the results from both tests pertaining to the same package. Also, the NuGet packages that were not possible to rank with OpenSSF Scorecard are displayed in the graph with a score of 0. Examining Figure 4.13, the orange triangles showing AutoTrust's rating are predominantly distributed between 5 and 4 for the top 50 packages within the green area, while it is mainly distributed between 3 and 2 for the rest. In contrast, for the OpenSSF Scorecard package ratings most of them are distributed between 3 and 2 for all the package ratings, including the top 50. We see that there is a more distinct difference between AutoTrust's ranking of the top 50 most downloaded packages and the other package categories, compared to the ranking made by OpenSSF Scorecard.

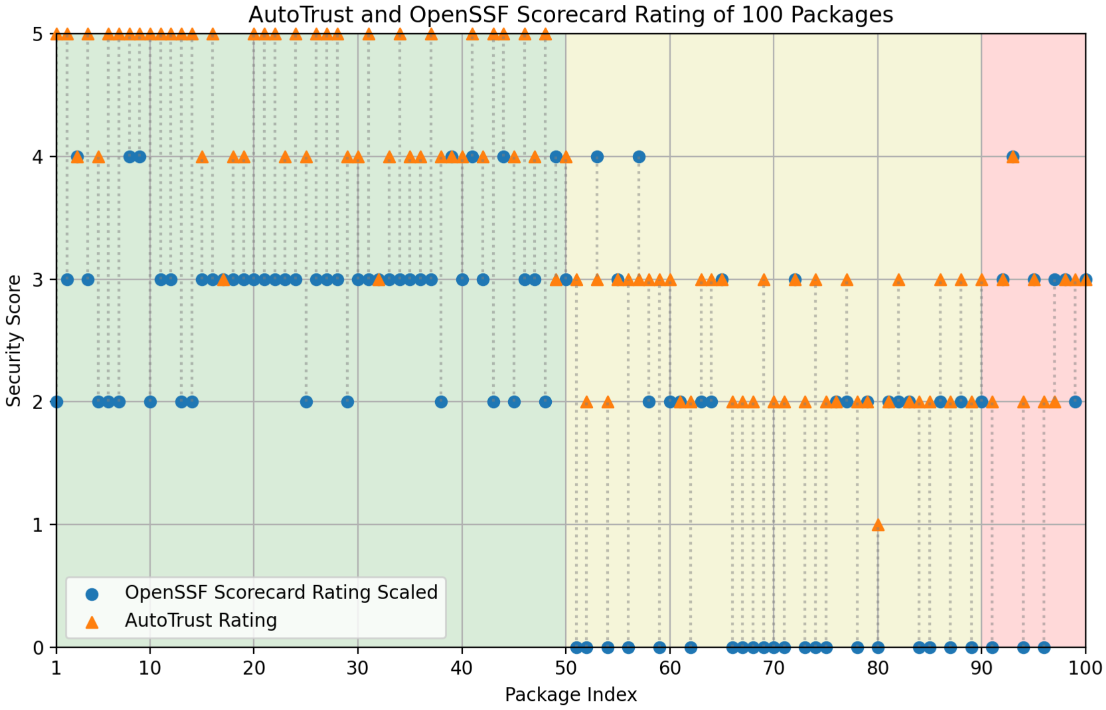


Figure 4.13: Comparison of the AutoTrust and OpenSSF Scorecard rating of the 100 packages. The green area: top 50 packages, yellow area: 40 random packages, red area: 10 problematic packages.

Similar to AutoTrust, the ranking given by OpenSSF Scorecard was highest for the top 50 most downloaded packages, and higher for the 10 problematic packages with issues than the 40 random packages. Out of the 10 problematic packages with issues, the highest-scoring one was Microsoft.EntityFrameworkCore.Tools which was the same one receiving the highest score of the problematic package by AutoTrust.

Chi-square Test of Independence

To decide if the result of the two tools' ranking methods is enough to reject the null hypothesis, and confirm that there is a statistically valid difference in the ratings, we can use the chi-squared test of independence, as described in section 3.4.2. The first step is to define the null hypothesis and alternative hypothesis which we did in subsection 3.2.2:

- **H_0:** The ranking of packages and the tool used to give those rankings are not related given the 100 packages. The proportions of rankings are the same for the two tools.

- $\mathbf{H_a}$: The ranking of packages and the tool used to give those rankings are related given the 100 packages. The proportions of rankings are not the same for the two tools.

The two categorical variables tested in these hypotheses are the ranking of packages, and the two tools used. The next step is to create the contingency table, which is shown in Table 4.7.

Table 4.7: Contingency table showing the resulting ranking from AutoTrust and OpenSSF Scorecard.

Ranking	AutoTrust Count	OpenSSF Scorecard Count	Total
5	28	0	28
4	20	10	30
3	27	38	65
2	24	28	52
1	1	0	1
Sum	100	76	176

From the table we can see that OpenSSF Scorecard was only able to rank 76 of the 100 packages. From this table, we were able to calculate the expected distribution by assuming that the null hypothesis is true. This means that there should be no difference between the two results from the tools.

Table 4.8: Contingency table showing the expected distribution of AutoTrust and OpenSSF Scorecard using percentage.

Ranking	AutoTrust	OpenSSF Scorecard	Total
5	15.91%	15.91%	15.91%
4	17.05%	17.05%	17.05%
3	36.93%	36.93%	36.93%
2	29.55%	29.55%	29.55%
1	0.57%	0.57%	0.57%
Sum	100%	100%	100%

The percentage distribution in Table 4.8 is calculated by looking at the total for each ranking category and dividing it by the 176 total number of results. We are then able to calculate the expected frequency for all the ranking levels for both tools by multiplying the percentage for each cell with the total number of respondents of the corresponding tool. This completes the third step and is shown in Table 4.9.

Table 4.9: Contingency table showing the expected results of AutoTrust and OpenSSF Scorecard.

Ranking	AutoTrust Count	OpenSSF Scorecard Count	Total
5	15.91	12.09	28
4	17.05	12.95	30
3	36.93	28.07	65
2	29.55	22.45	52
1	0.57	0.43	1
Sum	100	76	176

Using the expected frequencies, we are now able to calculate the chi-squared test statistic using Equation 3.1. The calculated chi-squared test statistic is:

$$\chi^2 = 31.822$$

The chi-square test statistic measures the extent to which the observed frequencies differ from the expected frequencies if there is no relationship between the two categorical variables. If the test statistic is large enough we can conclude that the variables are related and that we can reject the null hypothesis. To decide what is large enough we need to compare the test statistic to a critical value for a chi-square distribution. To find this critical chi-square value we need to decide on the degrees of freedom and the significance level.

The degrees of freedom can be calculated directly from counting the columns and rows of the table, excluding the "Total" column, and "Sum" row. The degrees of freedom is $(2-1)\times(5-1) = 1\times4 = 4$. The significance level is set to 0.05 (5%), which is a common threshold for the chi-square test of independence [61]. This significance level represents a threshold for determining whether the observed relationship between variables is statistically significant. By setting a significance level of 0.05, we are essentially stating that we are willing to accept a 5% chance of making a Type I error, which is to reject the null hypothesis when it is actually true [61].

The critical value based on a degree of freedom of 4 and a significance level of 0.05, found in a chi-square critical value table[6], is 9.49.

[6] https://www.scribbr.com/statistics/chi-square-distribution-table/

The final step is to compare the critical value to the chi-squared test statistic. We have a chi-squared test statistic of 31.822 which is larger than the critical value of 9.49. This means that we can reject the null hypothesis. The large difference between the values also shows that there is a significant difference between the results from AutoTrust and the results from OpenSSF Scorecard.

Testing the Package Categories

Further, we also decided to analyze the different categories of packages to see if the null hypothesis would be rejected for the individual categories. We wanted to see if the difference between the tools holds true when comparing the top 50 packages, the 40 random packages, and the 10 packages with issues individually. Following the same procedure as described above, with the same significance level of 0.05, we got the following results:

Table 4.10: Result of performing the chi-square test of independence for the three package categories.

Package Category	**Degrees of Freedom**	**Critical Value**	**Chi-Square Test Statistic (χ^2)**	**Rejected?**
Top 50 packages	3	7.82	77.16	Yes
Random packages	3	7.82	9.41	Yes
Problematic packages	2	5.99	1.31	No

As we can see from Table 4.10 we would be able to reject the null hypothesis by comparing the top 50 packages and the 40 random packages. Despite being able to reject the hypothesis for both categories, the difference between the critical value and the chi-square statistic was much greater for the top 50 packages. For the 10 packages with issues, we could not reject the null hypothesis. However, this result is highly affected by the fact that there were only 7 packages rated by OpenSSF Scorecard and 10 packages by AutoTrust. Having such a low number of packages will affect the statistical power, increasing the risk of making the previously described type I error or type II error, which is a nonrejection of the null hypothesis when it is false [61].

Validity of the OpenSSF Scorecard Experiment

The validity of the experiment may have been influenced by various factors, which can encompass conclusion, internal, construct, or external validity. The conclusion validity of this experiment might have been affected by the fact that NuGet packages without open source code are not tested by OpenSSF Scorecard. This is in contrast to AutoTrust which can still evaluate packages even though a link to

GitHub is not provided. Since OpenSSF scorecard is limited to projects with open source code, this may introduce bias into the data. Packages without open source code could potentially have lower average rankings compared to packages with open source code. Of the 24 packages OpenSSF Scorecard was not able to rank, AutoTrust gave 5 of them three stars, 18 of them two stars, and 1 of them a single star. Since these packages had a tendency to perform poorer on the ranking, this might have affected the observation that AutoTrust had a more distinct ranking between the top 50 most downloaded packages and the random packages than OpenSSF Scorecard.

Another factor that can have affected the reliability of treatment implementation is that AutoTrust evaluates the latest stable package version, while OpenSSF Scorecard evaluates the project and its latest stable commit on the main branch. This could lead to a difference in what is being evaluated by the two tools. An example could be if the license is available on NuGet, while it has been removed on GitHub, which will lead to AutoTrust giving a high score for the license and OpenSSF Scorecard failing that check.

The internal validity of the experiment might also have been affected. We conducted the 100-packages demonstration of AutoTrust on the 21st of Apr 2023, while we did the testing of OpenSSF Scorecard on the 18th of May 2023. Considering the nearly month-long gap between the two experiments, it is possible that changes were made to the packages and source code during that time. This can have led to a disparity between what was evaluated by the two tools leading to differences in the results.

Another factor that might have affected the internal validity was how we collected the data. For OpenSSF Scorecard we did not assess the individual score it gave for all the different checks it performed. As only four of OpenSSF Scorecard's checks are similar to AutoTrust's validators we decided that the data would be hard to compare. However, it is possible that this data would have unveiled some trends that could have strengthened or weakened the conclusions.

The instrumentation of the experiment also affected the internal validity since the OpenSSF Scorecard tool needs a maintainer personal access token to do a proper evaluation of the branch-protection check. This has affected the score of the branch-protection check, however, we think the result is still valid since developers evaluating a potential NuGet package also will not have a maintainer personal access token. Also, since there are 18 checks in total, we believe lacking this token that only affects one check, will not have a big impact on the overall result.

The selection of the 100 packages we decided to test, was based on finding packages that would help demonstrate that the AutoTrust tool worked for packages in the NuGet ecosystem. Since these packages were selected to prove that AutoTrust works for NuGet packages, it might have been better for the experiment to select 100 random packages. Choosing packages with distinct features, such as the top 50 most downloaded packages and the problematic packages with issues, might have affected the internal validity of the experiment, as it is likely that these packages are different than regular packages on NuGet.

The construct validity might have been influenced by the fact that we have used the same 100 packages as in the 100-packages demonstration. This led to a low sample size for the problematic packages and random packages. The low sample size can have decreased the statistical power of this experiment.

Another issue that might have affected the construct validity is the inadequate preoperational explication of constructs, which refers to the insufficiency of defining constructs before they are converted into measures or treatments [58]. As there is no clear definition of what makes a tool better or worse regarding assessing security risk, it is hard to give a comparative analysis based on equivalent criteria. For instance, is the AutoTrust rating better than the OpenSSF Scorecard rating since it considers more criteria, or is OpenSSF Scorecard better since it goes more in-depth into the source code? Questions like these do not have a clear answer, which is related to the problem of having no ground truth as to what makes a package risky.

As OpenSSF Scorecard and AutoTrust used different scales, we decided to rescale the OpenSSF Scorecard ranking to be able to conduct the experiment by converting its scale as described in Table 4.6. This modification of the scales involves changing the measured data, consequently impacting the construct validity of our study. Since this could potentially influence the result of the chi-squared test of independence we calculated and analyzed how the rescaling affected the OpenSSF Scorecard package ranking.

To calculate the increase and decrease in package values, using Equation 4.1, we compared the original OpenSSF Scorecard rankings (R_o) with the values 1, 3, 5, 7, and 9 (V_c). All rankings within 1 above or below each value were converted to that specific value. The total information loss (T) was determined as the aggregated difference (D) between the original ranking and the converted value. The decrease or increase (I) indicated whether the ranking was rounded down or up, respectively. After having converted the numbers to either 1, 3, 5, 7, or 9, we further transformed these values to a scale of 1, 2, 3, 4, and 5.

Assuming that R_o, V_c, D, and I are all single values, the formula can be written as:

$$
\begin{aligned}
V_c &= \{1,3,5,7,9\},\\
D &= |R_o - V_c|,\\
I &= \begin{cases} \text{Increase} & \text{if } R_o < V_c,\\ \text{Decrease} & \text{if } R_o > V_c,\\ \text{None} & \text{if } R_o = V_c \end{cases}
\end{aligned}
\tag{4.1}
$$

In Table 4.11 we have displayed the information loss occurring by rescaling the 0-10 scale to the 1-5 scale, within the context of the original 0-10 scale.

Table 4.11: Table displaying how the rescaling of OpenSSF Scorecard affected the package categories.

Package Category	Increase	Decrease	Total (T)	Total per package
All packages	13.9	15.6	-1.7	-0.0224
Top 50 packages	8.6	12.6	-4.0	-0.08
Random packages	3.8	2.0	1.8	0.0947
Problematic packages	1.5	1.0	0.5	0.0714

As we can see from Table 4.11, the total change added to the packages score was only -1.7, and by having 76 packages, this on average changed the package scores by -0.0224. As the chi-squared test statistic, 31.822, was much larger than the critical value, 9.49, this minor difference in package score would not have had an impact on whether we could reject the null hypothesis or not. Also, by looking at the individual package categories we believe the total per-package differences are not large enough to affect the final outcome.

We have also made some conscious choices to improve the validity of this experiment. We did not only assess the overall difference between the package ranking made by OpenSSF Scorecard and AutoTrust, but we also examined how they ranked the different package categories. Since we could reject the null hypothesis for the top 50 packages and the 40 random it strengthens the conclusion that the evaluation of the packages was in fact different between the two tools. Another positive aspect to consider for the validity is that since both AutoTrust and OpenSSF Scorecard are computer tools they consistently yield the same outcomes, which improve the reliability and reproducibility of the findings. An additional valuable aspect to highlight is that we evaluated the package ranking distributions both visually and using statistics, which increased the chances of spotting unforeseen patterns in the data.

4.5.3 Interviews with Visma

The interviews with Visma were the final method we used for gathering data, and the questions asked can be found in section B.5. The motivation behind doing interviews was to have software developers test out the tool and get their feedback. Unlike the artificial evaluation conducted during the student experiment and OpenSSF Scorecard experiment, the testing carried out by the Visma developers was a naturalistic evaluation [7]. The reason for this is that the developers tested out the tool as part of their daily work, and on real use cases. They tested out packages already used in their projects, more popular packages, and packages they knew had vulnerabilities. To find interview participants we combined multiple approaches. To increase interest in the topic, we posted the earlier mentioned one-pager (section B.9) on Visma's intranet, where we describe the issue of malicious packages and how to prevent downloading them. Through the one-pager, we asked if people interested would take the time to test out the tool. We and our supervisor also sent out multiple emails to Visma employees asking them directly if they could test out the tool and join us for an interview. In addition, we asked the participants if they could ask if their colleagues had time available to test the tool.

Despite the effort put in to get participants, we ended up with only 4 participants, which is considerably less than our goal of 12, which was recommended for interviews [73]. However, their responses were quite extensive and we still got a lot of valuable feedback. The participants exhibited varying levels of experience with NuGet, ranging from 1 year of usage to 10 years. The frequency of how often the participants were adding NuGet packages was varying as it was primarily done at the beginning of new projects. However, most participants reported adding new packages on a monthly basis. When questioned about their practices for adding packages, only two respondents indicated that they conducted any form of evaluation. Their evaluation methods were limited to inspecting packages on NuGet or checking for problems on Stack Overflow[7], which is similar to what the students did in their assessment. Additionally, it is worth noting that none of the participants had prior experience using security tools for this type of assessment, and their lack of awareness about such tools aligns with their limited previous evaluation efforts.

All of the participants used less than 15 minutes to set up and start using the tool. During the setup of the tool, only one participant encountered any issues, and that was to wrap the "dotnet add package" command with the AutoTrust functionality, which is a feature not necessary to test and run AutoTrust. During the participants' usage of AutoTrust for package inspection, only one issue was mentioned. When attempting to install a non-existent package, the tool executed all the checks without providing a warning indicating the package's non-existence. One participant also mentioned that they would have liked to get some feedback on whether a package was actually installed or not when deciding to install an evaluated package. To get additional information about the tool, two of the participants used the verbosity flag, and they were both satisfied with the implementation, and it worked as they would expect.

During the assessment of the validators, most participants expressed appreciation for the information in the README, detailing how the validators functioned. They found this to be valuable, and they felt they could trust the result more when they knew how it was obtained. Two participants mentioned that there was one of the validators they did not understand. One mentioned the analyzers, and one mentioned the open pull requests. To understand analyzers, one would need a deeper knowledge of how NuGet and .NET function so it makes sense that this is harder to grasp. The interviewee that questioned the open pull request validator said he did not understand why we had decided on the chosen thresholds.

[7] `https://stackoverflow.com/`

This coincides with a statement from one of the security engineers that said that multiple factors can affect how pull requests are being handled in a project.

There were also mentionings of the initialization script validator being "a bit weird", contributors feeling a bit unnecessary, and the low thresholds for the age validator. Similar to analyzers, the initialization script validator requires an understanding of how it can be exploited for malicious purposes in order to comprehend the necessity of evaluating it. The contributors validator can be hard to grasp and evaluate as it can be affected by multiple factors such as if the contributor is not a single person, but a whole company. The low threshold for the age was set to three weeks, which might be a bit strict if the project is often updated with new small patches. However, despite these individual observations, the overall consensus was that none of the validators should be removed.

All participants expressed confidence in the overall assessment provided by AutoTrust. They thought the rating was a good summary of how the package had been evaluated for all the validators. While all participants found the rating to be representative of the package's trustworthiness, one participant also expressed a preference for viewing all the metrics used in the calculation instead of a summary. It is worth noting that this individual did not utilize the diagnostic flag or read about the validators in the README. The remaining participants found the star rating to be useful and appreciated the convenience of receiving a summary.

In regard to the approach of analyzing packages, the participants expressed positive feedback. They considered it a good idea to perform pre-analysis checks, noting that it is something they typically do not do themselves. Also, they found the implementation to be non-intrusive in their workflow, which was appreciated. The participants emphasized that AutoTrust is helpful, particularly since developers often have limited time available for thorough assessments. All the participants said that they would continue to use the tool, which is an indicator that they found it useful and valuable.

Validity of the Interviews

Several factors can have influenced the validity of the interviews, including but not limited to the conclusion, internal, external, or construct validity. The conclusion validity was affected negatively since the number of respondents was only four. This was a small sample size and a larger sample size could have provided a better representative and diverse range of perspectives.

The internal validity might have been affected by the interviewees not testing the tool for a long period of time and not putting too much effort into understanding it. We wanted to test the tool in a naturalistic setting, so we allowed the interviewees to read about the tool and try it out as they wanted. This allows for the tool to be tested more similarly to how it would be used in a real-world setting, but it can have led to some of the interviewees not having enough prior knowledge before participating in the interviews. We got this suspicion confirmed when one of the interviewees said they had not read about how the validators worked in AutoTrust's README.

In some of the questions, we asked the respondents to give specific numeric answers, such as asking how long time they used on installing AutoTrust. These kinds of questions have issues with recall bias, as they might have difficulty accurately recalling specific details such as the time they used to install AutoTrust. They can also be affected by social desirability bias. An example that might have happend is that the interviewees felt like they used too much time installing AutoTrust, which led them to report lower time spent than what they actually did. Recall bias and social desirability bias are general problems when conducting research with self-reports and affect the internal validity.

The external validity was affected negatively by how we recruited the interviewees. These interviews are affected by volunteer bias as we only selected people that were willing to try and test AutoTrust. It is very likely that the people that were initially positive about the idea also were the same people that were willing to test AutoTrust and participate in the interviews. Since some of the respondents read the one-pager, described in subsection 4.5.3, they might have had a deeper understanding of the issues concerning malicious packages than developers not reading the article. Some of the interviewees even referenced the article in their replies. The results may also have been influenced by the fact that Visma, being a security-focused company, has employees who are likely to be more positive towards security tools compared to employees in other organizations. All of these factors can have introduced a bias in favor of a tool like AutoTrust and affected the generalizability of the interviews.

Construct validity was influenced by the fact that we designed the interview guide ourselves, it might have been affected by confirmation bias as the questions can have been created to confirm our beliefs. However, we attempted to ask the interviewees to state both positive and negative experiences they had with the tool to get a more nuanced response. The construct validity may also have been affected by courtesy bias. Since we conducted the interviews ourselves the respondents may have not fully stated their unhappiness in an attempt to be polite.

We also took some measures to improve the validity of the interviews. We had respondents from different countries and teams which increased heterogeneity. This decreased the likelihood of cultural differences affecting the results.

To improve the generalizability and how effectively we could measure what we intended to research, we used Davis's technology acceptance model as the basis for the questions regarding usefulness and ease of use [74]. The model is one of the most used models to measure users' acceptance and was helpful in creating questions to accurately measure usefulness and ease of use.

To minimize the potential of collecting inaccurate interview responses, we recorded the interview to be able to correctly write down the replies. We also both participated in the interview to increase inter-rater reliability as we were two people interpreting the replies. After the interview was finished we sent the transcript back to the interviewees to allow them to fix any errors and confirm that we had understood them correctly.

As the sample size was relatively small we took multiple measures to increase it. One measure was to create traction through the one-pager. It got positive responses from multiple employees, however, only one of the interviews came directly from this tactic. We also conducted the interviews on Teams so it would be easier for people to participate. All of our respondents were from abroad and would not have been able to participate without having it online. After finishing the interviews we asked if the interviewees knew any other colleagues that could be interested in trying AutoTrust to increase the sample size, some of the interviewees asked multiple colleagues but it did not lead to more respondents. We also did not want to use developers from other companies as Johannesson and Perjons state that the artifact should not be used by subjects outside of the research context during technical action research [57].

4.5.4 Goals of the Evaluation

To do an overall assessment of the artifact we considered the four evaluation goals presented in the introduction of section 4.5. In this assessment of the evaluation goals, we have taken all the parts of this research project into account. This includes the Visma Questionnaire, the discussion with the two security engineers, the 100-packages demonstration, the time demonstration, the student experiment, and the interviews with Visma employees.

Evaluation Goal 1

> *Evaluate if the artifact is effectively solving the problem.*

The primary objective of the artifact evaluation is to assess the effectiveness of a given artifact in addressing the problem it was designed to solve. AutoTrust needs to solve the problem stated in subsection 4.1.2: "Threat actors can use dependencies as an attack vector to infiltrate software projects and there is no simple way of evaluating the security of these dependencies prior to installing them". Specifically, how can AutoTrust evaluate NuGet packages and prevent users from installing malicious packages that could lead to compromises?

To determine the utility, the first action we did was the 100-packages demonstration to see if the tool worked as intended and to look for bugs or unwanted behavior. The conducted test did not reveal any instances of undesired behavior, and the observed ranking of the different packages was consistent with anticipated outcomes.

Then with the student experiment, we saw that manually assessing packages is a cumbersome process, and the quality of this assessment is varying based on the time available and the experience of the developer. From the interviews, all the participants found the package assessment provided by AutoTrust to be credible and by having a summary and executing quickly, the participants thought it was valuable and not causing any extra work. Another important aspect was that the tool provided an extensive check of packages that had not been performed earlier and one participant said it started a conversation between his colleagues about the issue of malicious packages. These aspects show that AutoTrust definitely helps with increasing security and decreases the chance of developers downloading malicious packages.

Evaluation Goal 2

> *Evaluate the functional and non-functional requirements proposed used on the artifact.*

The second objective is to assess both the functional requirements and non-functional requirements, which we are referring to as quality attributes. Each requirement is presented below together with a justification of how it has been fulfilled, using the results from the demonstration and the evaluation phases.

- **FR-1:** The artifact should evaluate packages before they are installed.
  - The fulfillment of this requirement was taken into consideration from the start of the design process and subsequently verified upon completion of the 100-packages demonstration.
- **FR-2:** The artifact should be possible to run from the CLI.
  - The fulfillment of this requirement was also taken into consideration from the start of the design process and subsequently verified upon completion of the 100-packages demonstration.
- **FR-3:** The artifact should run in the .NET ecosystem and work for all NuGet packages from the central repository.
  - This requirement was tested both through the 100-packages demonstration and by the Visma employees we interviewed. From the 100-packages demonstration, we found no problems with any of the packages. To ensure optimal compatibility within the .NET ecosystem, we developed AutoTrust using the dedicated console app template provided by .NET. From the interviews, we had one participant that tested out the tool on a privately hosted package, but as the tool is not made to work for packages hosted that way, the tool did not work.
- **FR-4:** The artifact should display information to the user if the package passed or failed the thresholds of the trust criteria. In unclear circumstances, the user should get a warning.
  - This was verified by the Visma employees being interviewed. They found the console information about which validators failed, warned, and passed to be clear and liked the red, yellow, and green coloring used.
- **FR-5:** The artifact should fetch package metadata in real-time.
  - To meet this requirement, we ensured that every time the script ran,

we fetched all the data and avoided caching it.

- **FR-6:** The artifact should provide an option for getting a more detailed explanation of the evaluation process.
    - The incorporation of an additional information feature was done by adding the option of a verbosity flag, as described in subsection 4.3.7. This was validated through the 100-packages demonstration as well as the Visma interviews. The feedback from the Visma employees indicated that the additional information was valuable for better understanding what the specific validators did. There were two of the participants that tested out the verbosity flag and they both liked the way it was implemented and the information it provided.
- **QA-1:** Usability
    - The confirmation of the first part, regarding the implementation of flags, was taken into consideration during the design process. Using the .NET standard syntax for flags will contribute to conformance to a standard that the developers are used to, which will make the tool easier to use.
    - The second part of this attribute was verified through the Visma interviews, as all of the interviewees used less than 15 minutes to install and start using the tool. The interviewees expressed that the tool was user-friendly and easy to comprehend, particularly after using it on multiple packages and gaining familiarity with the ranking system. The interviewees found both the README with instructions and an image displaying the output from running the tool useful, and it helped with getting the tool up and running quickly. The participants noted that AutoTrust offered a seamless user experience while also introducing a new practice they believed should be incorporated into their workflow.
- **QA-2:** Portability
    - The compatibility of AutoTrust with the operating systems Linux, macOS, and Windows, was confirmed through the time test. Furthermore, the compatibility of AutoTrust was thoroughly tested during both the 100-packages demonstration and the Visma testing. In the 100-packages demonstration, we utilized all three mentioned operating systems (Windows, macOS, and Linux) to ensure cross-platform functionality. During the interviews, the software was exclusively used on Windows. We did not encounter any issues or complications during the 100-packages demonstration, and there were only a few non-critical problems reported in the interviews.
- **QA-3:** Security
    - Throughout the entire development process, special attention was given to ensure that the tool did not rely on any third-party dependencies.

    No external libraries or dependencies were used in its implementation, mitigating the risk associated with incorporating potentially malicious third-party code into the tool.
  - Additionally, AutoTrust incorporates a prompt that asks the user to decide whether they want to download the package or not, following the evaluation of the package. Through the 100-packages demonstration, we ensured this was working as expected.

- **QA-4:** Performance
  - The validation of the performance attribute was done through the time test, where we found that the installation time did not exceed three times that of the "dotnet add package" command. We did not receive any feedback on lacking performance from the interviews, which indicates that it is performing well enough.

- **QA-5:** Modifiability
  - During the implementation process, adjustments to the design were made to ensure the possibility of adding new validators. Although specific testing of this capability was not conducted after the artifact's completion, it was thoroughly verified during development. All validators were successfully incorporated without any adverse effects on each other, thereby confirming the successful implementation of this attribute.

Evaluation Goal 3

> *Compare the artifact to other similar artifacts that intend to solve the same or a similar problem.*

The third evaluation goal is not only to assess the artifact in isolation but also to compare it with other artifacts that aim to address the same or a similar problem. In this regard, the new artifact must demonstrate advantages over existing artifacts. This evaluation goal was assessed by comparing AutoTrust to the manual assessment of packages tested through the student experiment, and to compare it to OpenSSF Scorecard. We also intended to compare the tool with OSSGadget, however, evaluating risk with OSSGadget through OSS-risk-calculator was not functioning as described by the authors.

This evaluation goal will be discussed further when answering the RQ-3 in section 5.3.

Evaluation Goal 4

> *Investigate the side-effects such as unintended or harmful effects of the artifact.*

A fourth objective of artifact evaluation is to examine the potential side-effects associated with the use of the artifact. Specifically, this involves assessing whether any unintended or adverse consequences may arise from using the artifact.

The relatively short testing period was not optimal for uncovering potential side-effects, as they are often revealed during extended usage and observation. However, through the interviews, we got feedback on both the positive and negative side-effects of AutoTrust. A positive side-effect was that the tool started conversations around supply chain security and malicious third-party dependencies among Visma coworkers. In addition, it introduced a new practice of dealing with security issues, that the participants would want to follow in the future. One potentially negative aspect of using AutoTrust and blindly trusting its results is that it could replace manual assessments, potentially leading to a lack of critical evaluation. However, considering the interviews conducted, it is worth noting that half of the respondents admitted to currently performing no assessment at all when downloading packages. Therefore, while there is a concern about overreliance on AutoTrust, it may not be a significant problem in practice.

Chapter 5

Discussion

This chapter starts by providing answers to the research questions presented in section 1.3, followed by a discussion on the implications of this research, both for the research and the practice. Thereafter, reflections on the limitations of this research and threats to the validity of the project are presented. This chapter concludes with general reflections on the research project, including potential improvements of AutoTrust.

5.1 Research Question 1 - Useful Information

RQ-1 | What information is useful for the assessment of security in third-party software packages prior to their installation?

The trust criteria presented in Table 2.2 is a collection of information that can be valuable in the assessment of third-party software packages' security, without focusing on one specific programming language ecosystem. The validators presented in subsection 4.3.3 were the physical implementation of the TC used in AutoTrust.

5.1.1 Evaluating Trust Criteria and Validators

Not all the TC are testable for all programming languages, such as trust criterion 26, "The component does not contain installation scripts", which will only be relevant if the programming language ecosystem supports installation scripts. The relevance of the various TC will also be based on the available information. As we saw in section 2.5, the package registries provide different information, which will affect how one is able to evaluate the package.

From the Visma Questionnaire, presented in section 4.3.2, we saw that many of the proposed TC were considered valuable. The most valuable TC based on the feedback received include checking if the package or its dependencies are deprecated, identifying security vulnerabilities, ensuring access to the source code and that there are no differences between the source code and package, examining the component's license, assessing any prior harmful effects associated with the component, and evaluating the component's widespread use over time. AutoTrust evaluates all of these except it does not check if there are any differences between the source code and the package. We could have checked the "Author signature" to see if the package was modified since the author signed the package. However, this can be checked with the "dotnet nuget verify"-command, and if the package content has been modified since it was signed, the installation is blocked with error NU3008, so it would not be possible to add the package [71, 75].

The TC considered the least important by the Visma employees were if the package has changed ownership recently, if the maintainers of the component are overloaded, and the size of the repository. If the package has changed ownership recently it might indicate a hostile takeover of the package. However, we agree with the respondents and we do not see this as very important. We believe this would just create false positives since most ownership changes are benign. Evaluating if the maintainers of the component are overloaded is hard as there are many factors outside of the NuGet ecosystem that might affect this. An indicator of being overloaded could be that the package maintainer is responsible for multiple packages, but maintaining multiple packages could also be an indicator that the maintainer has experience in the programming language ecosystem. Including information about the size of the repository would have been easy to add since it is available from the NuGet API. Projects with larger sizes have more code, which means they have a greater attack surface [30]. However, we did decide not to implement this since it was the trust criterion that scored the lowest by the Visma employees and we believe it is hard to set a proper threshold for when the size of a project is considered too large.

From the student experiment in subsection 4.5.1, we saw that many of the participants considered the following validators: popularity, widespread use, age of the package, documentation, and package license. As they conducted a manual assessment, they relied more on metadata that was easily accessible. This might indicate that code hosting platforms should prioritize making more data easily available, and stimulate package maintainers to provide more information.

The students also considered information that AutoTrust does not. They analyzed the characteristics of the package and they used search engines to consider public attention, exposure, and reported problems. This information would be valuable, however, in our perspective, assessing characteristics and public attention automatically poses challenges. Creating a program that accurately emulates a human's ability to form an overall impression of a package, by considering multiple varying factors, is a challenging task. Automatically analyzing such aspects requires a more in-depth assessment, possibly employing natural language processing techniques, rather than simply determining pass or fail based on predefined numerical thresholds. It is worth mentioning that NuGet does offer package owners the ability to add descriptive tags to their packages, which could have been used to analyze the characteristics of the package. However, these tags can often be inadequate if the owner has not included enough relevant keywords and this will not give an equally good overall impression as a manual evaluation.

In the interviews with the Visma employees, we asked them about their opinion on the validators we had implemented in AutoTrust. The validators they found the most useful were information about deprecated packages, known vulnerabilities, popularity, and widespread use. The information about deprecated packages and known vulnerabilities was also ranked high by the Visma employees in the questionnaire. This further strengthens the confidence that these two validators are considered useful. High scores in popularity and widespread use can indicate that software packages are trusted by others. Such information is easily understandable and accessible, and this might also be a reason why they were ranked highly by the Visma employees. One of the interviewees said that they found the information about open issues, open pull requests, and maintainers valuable. These three validators can be used as indicators of how well the project is being maintained. None of the interviewees said they would remove any validators, but one did not understand the analyzers validator and another did not understand the open pull request validator. To make the information provided valuable for the user it is important that they comprehend what is being assessed and understand why it is being evaluated. This finding aligns with the feedback from the interviewees, who expressed a higher level of trust in the results when they had a clear understanding of the process.

It is also worth mentioning that it is hard to reach a consensus on what are good indicators for evaluating if a software package can be trusted, and the thresholds to evaluate these indicators. In the work by McDonald and Goggins, they found that one of the most common measures of an open source project's success was the number of contributors and the contributor growth [76]. In contrast, Zahan et al. proposed too many maintainers and contributors as potential indicators of a package being exposed to a higher risk of a supply chain attack [24]. Zahan et al. also asked 470 npm package developers about their proposed indicators and more than 40% of them disagreed that too many maintainers and too many contributors were indicators of increased risk of a supply chain attack. This shows that there are disagreements between what are actual indicators of heightened risk. Some of the other criteria can also be interpreted as both good and bad in regard to increased risk. Trust criterion 20, "The project does not have reported security vulnerabilities", can be good if the package actually has no vulnerabilities, or it can be bad if the reason is that no one has put effort into detecting and disclosing vulnerabilities in the package.

5.1.2 The Most Prominent Trust Criteria and Validators

Table 5.1 and Table 5.2 display the TC and validators that were highest rated or got the most traction during the different phases of this research. All the trust criteria and validators were found to be valuable but these were added to better visualize which can be considered the most useful to assess in the pre-installation phase for security tools. Table 5.1 presents the TC that got the most attention in the academic papers from the SLR and was considered the most useful in the Visma Questionnaire. There were 2 TC that were mentioned in 5 papers and 5 TC that were mentioned in 4 papers. The ranking of the TC from the Visma Questionnaire is ordered by how valuable the employees considered the TC.

Table 5.1: The ranking given to the TC in the SLR and the Visma Questionnaire.

	SLR (TC)	**Visma Questionnaire (TC)**
1	• The component has been in widespread use for a considerable amount of time. • The component is being developed by an active maintainer domain.	The component is not deprecated.
2		The component does not depend on deprecated packages.
3	• The component has an adequate number of maintainers and contributors. • The maintainers, owners, and suppliers of the component are trustworthy. • The component's maintenance lifecycle is up to date. • The component is not deprecated. • The component does not depend on deprecated packages.	The project does not have reported security vulnerabilities
4		Access to the source code is provided.
5		The component provides a standard or well-written license.
6		There is no history of prior harmful effects associated with the component.
7		The component has been in widespread use for a considerable amount of time.

Table 5.2 contains the ranking of the validators made by the students and the interviewees. The order of the validators for the students is based on how many students said they used each validator as a criterion when manually evaluating packages. The four last validators were considered by the same amount of students, so they are all included. Given that we conducted interviews with only four employees, we have listed all the validators they identified as useful. However, due to the small sample size, these validators have not been arranged in any particular order.

Table 5.2: The ranking given to the validators in the student experiment and interview.

	Student Experiment (Validators)	**Interviews (Validators)**
1	Popularity	• Deprecated package • Known vulnerabilities • Popularity • Widespread use • Open issues • Open pull requests • Contributors
2	Widespread use	
3	Age of package	
4	Documentation	
5	• Number of contributors • Deprecated package • Known Vulnerabilities • License	
6		
7		

5.1.3 Summary of Answer to RQ-1

Based on the feedback received from the two security engineers, the SLR, and the ranking by the Visma employees in the questionnaire and interview, it is clear that several of the proposed TC can be useful when assessing the security risks associated with third-party packages. The TC that were found the most useful in the different evaluation methods were information about deprecation or deprecated dependencies, known vulnerabilities, popularity, widespread use, license, and contributors. However, factors such as the time it takes to evaluate them, the availability of data, the ease of automatic evaluation, programming language-specific differences, and the comprehensibility of the trust criterion might be more influential factors when it comes to how useful the information is both for manual and automatic assessment of software packages.

5.2 Research Question 2 - Advantages and Disadvantages

RQ-2 What are the advantages and disadvantages of the CLI pre-install tool for developers in the process of evaluating third-party software packages?

During the development of AutoTrust, we found some potential advantages and disadvantages of tools with a design similar to AutoTrust.

5.2.1 Advantages

One of the primary advantages is that CLI security tools made for the pre-install phase can help detect potential malicious packages before the user installs them and gets compromised. As pointed out in section 1.1 a real-world example of such malicious packages was reported by JFrog. Since these packages used initialization scripts [6], they would have been detected and reported by AutoTrust. During the interviews, the respondents showed appreciation for this feature that checks for initialization scripts. AutoTrust is not only made to prevent installing malicious software packages but also to detect benign packages that might have vulnerabilities. Malicious packages might infect developers' laptops and vulnerable packages might lead to vulnerabilities in the main application. Both these problems would potentially lead to more work in the future, compared to the proactive evaluation of packages.

During the interviews, one of the main advantages mentioned by all of the respondents was that AutoTrust helped them do an assessment they should be doing, without much extra effort. Using tools like AutoTrust that are extending the functionality of already used commands instead of being a separate tool, prevents the developers from having to take a conscious choice to run the tool. This saves time and reduces the cognitive load.

When asking the Visma employees if they had any guidelines for how they should include new dependencies, many reported that they had no such guidelines. Even if the team members follow guidelines it is easy to miss a step, and as we saw during the student experiment people do not rank packages the same. Tools like AutoTrust can help create a standard for what the team should check before including a software package, and help make sure that all the criteria are evaluated correctly and none of them are forgotten.

As mentioned in the interview, tools like AutoTrust can also be used as part of pull requests. If the person creating the pull request includes an image of the AutoTrust output it can make it easier for people to assess the package. Another option is that the person looking to review and approve the changes proposed in the pull request can scan the packages themselves. This can start a discussion regarding what kinds of dependencies to include, and the need for adding the dependency in the first place. A downside of this is that it can potentially lead to a long discussion about which dependencies to include, especially if developers do not agree if the package failing one specific criterion means they should add it or not. If we had more time to test AutoTrust, it would be interesting to see if the tool would lead to unwanted discussions.

5.2.2 Disadvantages

A general disadvantage of security tools is that they can lead to notification fatigue. Having multiple notifications pop up in their workflow can feel intrusive. If we had tested AutoTrust for a longer time period with more developers, preferably people who were not too fond of the idea, it would be interesting to see if they still would consider the warnings from AutoTrust or just ignore them.

There are also some issues with how the developers decide to interact with the tool. As one of the interviewees stated, these kinds of tools are most efficient when combined with manual assessment. As mentioned, an issue is that developers might become lazy and trust the tool completely without adding their own critical reflection. The tool is based on best effort and cannot catch all issues, this can therefore create a false sense of trustworthiness. Another disadvantage linked to how the developers use the tool is if they understand all the validators and why they are added. As reported in the interview one did not understand the analyzers validator and another did not understand the open pull requests validator. The developers not understanding the criteria might lead them to add packages they should not, or decide to not add packages that they could.

Another issue with security tools is that developers need to know that they exist. AutoTrust might be hard to find for developers and it is likely that many do not know the security risk of adding new packages. The functionality provided by AutoTrust and similar tools should preferably be included as part of the package managers to reach a larger number of developers.

The last disadvantage, that came up during the interviews, is that some of the developers normally use the Visual Studio IDE and its integrated package manager for adding NuGet packages. When there are multiple ways of adding packages it can lead to an imbalance where some packages get assessed and others do not. This can lead the developers to believe all the packages have been evaluated even though some of them have not.

5.2.3 Summary of Answer to RQ-2

Overall, the main advantages are that tools like AutoTrust can lead to early detection of package issues and can be incorporated in a non-intrusive way into the developers' workflow. The main disadvantages of the tool are that developers can misinterpret the results and how they should respond to the results. To prevent this, they need to put effort into understanding what the tool assesses and why the tool does it.

5.3 Research Question 3 - Comparing Assessments

RQ-3 | How does the tool's assessment of third-party software packages' security risk compare to other ways of assessing these packages?

Our objective was to evaluate AutoTrust alongside various other methods of assessing third-party software packages. We conducted tests comparing AutoTrust with OpenSSF Scorecard and a manual evaluation performed by students. Additionally, we intended to evaluate AutoTrust with OSSGadget but encountered difficulties in doing so. Furthermore, we conducted a limited comparison with npq, although the scope was limited to how the tools are checking the packages rather than directly comparing them, as npq is specifically designed for npm. We also compared the functionality of AutoTrust to the .NET commands that are most similar to it.

5.3.1 OpenSSF Scorecard

OpenSSF Scorecard was the only tool we were able to compare directly with AutoTrust. The main difference between the two tools is that OpenSSF Scorecard is made to scan open source projects on code hosting platforms, while AutoTrust is made for packages in the NuGet package registry. From the comparison between AutoTrust and OpenSSF Scorecard in Figure 4.13 we saw that it was a difference in the distribution of the top 50 packages. Most of the OpenSSF Scorecard results were distributed between 3 and 2, and the AutoTrust results were mostly distributed between 4 and 5. This strengthens the credibility of AutoTrust's ability to distinguish high-risk packages from low-risk packages as it is reasonable to assume that the top 50 most downloaded packages are lower risk.

From the comparison of the two tools, we also found that there is a less distinct difference between the top downloaded packages and the random packages using OpenSSF Scorecard. This small difference between the top downloaded packages and the random packages might be due to the NuGet package owners not following the best practices recommended by OpenSSF. This coincides with the findings from Zahan et al. that there is a lack of adoption of the security practices proposed by OpenSSF by the npm and PyPI ecosystems [67]. Another possible explanation for the difference in how AutoTrust and the OpenSSF Scorecard differentiates the top 50 packages from the random packages could be that AutoTrust considers popularity as one of its criteria.

If we look closer at the Newtonsoft.Json package, OpenSSF Scorecard only gave it a rating of 3.5/10, which is very poor considering it is the most downloaded NuGet package. In comparison OpenSSF Scorecard gave the AutoTrust project a score of 5.6/10. We believe it will be hard for developers to trust the ranking when commonly used packages receive a very low score. Developers may find it challenging to make confident decisions regarding the inclusion of a package when considering the relatively smaller difference between the rankings of the top 50 most downloaded packages and random packages by OpenSSF Scorecard, as opposed to AutoTrust.

OpenSSF Scorecard is also not as easily integrated into the developers' workflow as AutoTrust is. AutoTrust can be used to extend commands developers already use while OpenSSF Scorecard is developed to be a distinct scan of projects. We also tested the time it took to run OpenSSF Scorecard on the Newtonsoft.json package on Windows. It used 26.5s compared to 5.127s using AutoTrust. We believe that since OpenSSF Scorecard takes a longer time to run it will be harder for developers to adopt it into their everyday workflow. However, the standalone nature of OpenSSF Scorecard might make it easier to integrate into CI/CD pipelines, which was reported as a possible improvement to AutoTrust in the Visma interviews.

The OpenSSF Scorecard tool checks if the project is implementing many security best practices such as if the project has a license, security file, and dependency update tools such as Github's Dependabot. As this tool is focused on using the information from GitHub and GitLab, the checks of these code hosting platforms are more thorough than the ones performed by AutoTrust. To improve AutoTrust it could be possible to query the OpenSSF Scorecard API to incorporate OpenSSF Scorecard's ranking into AutoTrust.

5.3.2 OSSGadget

We wanted to compare AutoTrust with the OSSGadget's tool OSS-risk-calculator. This tool tries to calculate the risk of using different open source software packages, similar to AutoTrust. It can be used to evaluate many different kinds of open source software packages including NuGet packages. It calculates the risk by using metrics from GitHub and an assessment of the package's characteristics to output a security risk score. However, when trying to evaluate the risk of the 18 most popular packages on NuGet, all of them scored 1/1, where 0 indicates no risk and 1 indicates very high risk, so we understood that something was incorrect. We reported the issue[1], and the maintainers confirmed that there was something wrong. Therefore, we were not able to compare OSS-risk-calculator's risk score with AutoTrust's security risk score.

Nevertheless, we can still compare how the authors of OSSgadget have designed the tool and what they have prioritized to implement. The OSS-risk-calculator is separated into two parts, one finding characteristics for the package and one checking the health of the GitHub repository. Including such information about characteristics would be a potential improvement to AutoTrust. This would have been helpful to better understand how the packages operate and was something the students did during the manual assessment. OSS-risk-calculator also considers health information such as project size, issues, pull requests, contributors, subscribers, forks, stars, number of releases, and issues that contains specific security keyword. This part is more similar to some of the checks considered by AutoTrust.

5.3.3 npq

The tool that is most similar to AutoTrust in its workflow is the npq tool, however, as it is only made for the npm ecosystem it is not possible to do a direct comparison of the same packages using AutoTrust and npq. Nonetheless, it is possible to do an evaluation of what is being evaluated with the tool. As discussed in section 4.3.8, there are three checks npq does that AutoTrust does not. Two of these checks that npq does are associated with the author, which verifies if the package has an author and if the associated email domain is still active. In the NuGet API the information about the author was just an input field that users could set themselves, and is not necessarily linked to the NuGet username of the package owner. We, therefore, deemed it unhelpful to check this, as it is not linked to if the package actually has an owner. The fact that AutoTrust does not check if the associated email domain is still active is linked to NuGet not providing information

[1] https://github.com/microsoft/OSSGadget/issues/150

about the author's email. The third different check that npq does that AutoTrust does not is checking if the project has a repository URL. AutoTrust warns if it does not find a link to a Github page, but we do not have a specific validator checking it.

The extensiveness of the checks is not discussed in section 4.3.8, as a complete comparison of how all the checks work in detail is out of scope for this thesis. However, based on the description given in the README file in the npq project it seems that we do more in-depth analysis for the different validators with AutoTrust. One example is that npq will display a warning if a package does not have a license, while AutoTrust also does an evaluation of whether the license is considered high, medium, or low risk.

5.3.4 .NET Commands

In addition to the prior tools, we also looked into the functionality supported directly by the .NET package manager. The package manager provides the option of checking packages already added to a project and its dependencies for known vulnerabilities and deprecated dependencies. One drawback is that it is not possible to check for vulnerable dependencies and deprecated dependencies at the same time [77]. The checks have to be done separately through the following commands:

```
dotnet list package --vulnerable --include-transitive
```

```
dotnet list package --deprecated --include-transitive
```

These checks are similar to the validators deprecated package, deprecated dependencies, and known vulnerabilities used in AutoTrust. However, these validators are not as extensive in their evaluation. The deprecated package is the same, but for the deprecated dependencies, the .NET package manager checks all the dependencies and does not stop at a depth of two which AutoTrust does. For the known vulnerabilities, AutoTrust only fetches vulnerability data for the main package and not for the dependencies as the .NET package manager does. In addition, both AutoTrust and the .NET package manager use the same data from NuGet, but AutoTrust also uses the OSV database to find vulnerabilities.

One reason for not directly comparing AutoTrust against these .NET commands is that they cannot be executed before the packages are added to a project. This means that it has one of the problems we are trying to solve, which is including the package before assessing it. Also, as these scripts are only similar to the three validators explained above, the direct comparison would not apply to the whole of AutoTrust.

5.3.5 Manual assessment

In the student experiment, discussed in subsection 4.5.1, we did a comparison of how computer science master's students do a manual assessment with AutoTrust's automatic assessment. From the evaluation, we saw that the students had a tendency to rank the packages higher, meaning less risky, than the AutoTrust tool and use fewer criteria in their evaluation.

Some issues with this experiment were that we only used computer science master's students, the experiment was artificial, and few of the students had much experience with the NuGet ecosystem. It would be interesting to see if security experts or senior developers would consider more criteria but based on the responses to the questionnaire and in the interviews it does not seem to be too common to have proper guidelines on what to assess in a manual assessment. In the interviews, the respondents that did assessments of packages prior to installing them reported that they usually used the NuGet website and Stack Overflow similar to what the students did during their manual assessment.

Overall, we saw that using tools gave a more thorough and faster evaluation of available package data attributes, but we still believe that manual assessment is valuable as it is easier to manually process language which can give an overall impression. A manual evaluation is also more flexible. An example is that AutoTrust can only evaluate GitHub repositories, while a manual assessment can examine multiple code hosting platforms like GitLab, BitBucket, etc. However, we do not believe that automatic assessments should completely replace manual assessments. As stated in one of the interviews the optimal approach lies in combining different tools for automatic assessment with human judgment and common sense.

5.3.6 Summary of Answer to RQ-3

In Table 5.3 we have summarized the findings from RQ-3 together with a statement saying if we could do a direct comparison between the tool being described and AutoTrust. We searched for multiple existing tools and methods for evaluating third-party software but were only able to directly compare AutoTrust against OpenSSF Scorecard and a manual student evaluation of packages. The OpenSSF Scorecard tool is similar to AutoTrust in the way it operates, but it is made for GitHub and GitLab, and not NuGet projects. However, OpenSSF Scorecard still considers some of the same criteria as AutoTrust including the package age, contributors, known vulnerabilities, and license. There was a difference in the rankings provided by OpenSSF Scorecard compared to AutoTrust of the top 50 packages, but the ranking of the 40 random packages and the 10 problematic packages were quite similar. Also, OpenSSF Scorecard takes more time to run, making integration into developers' workflows more challenging. From the student evaluation, we found that AutoTrust considered a lot more criteria and performed a more thorough analysis than the students. Further, AutoTrust was not able to do the same evaluation of the overall impression as the students were, indicating that tools like AutoTrust should be used in combination with manual assessments.

Table 5.3: Summary of the findings for RQ-3.

Tool	Comparison
OpenSSF Scorecard	*We did a direct comparison between this tool and AutoTrust, through the OpenSSF Scorecard experiment.* • Had a less distinct difference between the package ratings during the experiment. • Commonly used packages receive a low score, which might make it harder for developers to use. • Provides a more thorough evaluation of data from code hosting platforms. • More of a standalone tool and not as easily integrated into the developers' workflow.
OSSGadget	*We did not do a direct comparison between this tool and AutoTrust, as there was an issue with it.* • Not a distinct tool, but a set of 12 tools for open source projects. • Uses package characteristics and GitHub repository health to evaluate packages. • The authors consider deprecating the OSS-risk-calculator, which is the OSSGadget tool most similar to AutoTrust.
npq	*We did not do a direct comparison between this tool and AutoTrust, as it is made for the npm ecosystem.* • The tool is the most similar to AutoTrust. • Provides a more thorough evaluation of authors than AutoTrust. • Has an individual check to evaluate if the repository URL is included, which AutoTrust does not.
.NET Commands	*We did not do a direct comparison between these commands and AutoTrust, as they only check packages already added to a project.* • Integrated into the .NET tool. • Checks dependencies if they are deprecated and vulnerable. • Also checks the transitive dependencies if they are deprecated and vulnerable.
Manual assessment	*We did a direct comparison between this tool and AutoTrust, through the student experiment.* • The students ranked packages less risky compared to AutoTrust. • Varying how thorough people are in their evaluation. • People check different data attributes and they are inconsistent in what they consider. • A manual assessment might be better to get an overall impression. • Manual assessment can be more flexible than an automatic assessment. • Manual assessment can be used together with other security tools.

5.4 Implications for Research

Throughout our work on this thesis, we have identified certain areas that would benefit from further research and improvements by the research community. There is an issue when one wants to do a proper evaluation of security tools for package managers due to the significant difference in the number of benign packages compared to malicious packages. A collection of malicious packages found on NuGet similar to the npm, PyPI, and RubyGems packages found by Ohm et al. could contribute to better evaluation of tools and help find indicators of risks that tools should alert about [16].

Further assessment of trust criteria in Table 2.2 and their thresholds is needed as it is hard to find a good balance between when to warn and when to pass the criteria. Performing additional testing on a broader range of packages for a longer time would most likely help detect which criteria are useful, which are less useful, and which need their thresholds adjusted. While developing AutoTrust, our primary emphasis was on academic sources for identifying trust criteria. However, exploring security tools during the evaluation process could potentially help discover additional trust criteria. This could be done as further research into TC.

AutoTrust is an example of a shift-left solution that gives the developers more responsibility and understanding of the dependencies they add to a project. Researchers can further develop security tools with a shift-left approach, such as artifacts specifically designed to educate developers about the risks associated with software supply chain attacks.

More research into the measures taken by different companies to secure their software supply chain can also contribute to a better understanding of what is needed to improve the software supply chain security. As discussed in section 2.6, there are multiple tools that can be used, and finding out what is being used and how it is being used can contribute to better guidelines and recommendations for companies. In addition, researching company practices might lead to discoveries of internal tools or practices used that are not publicly known.

In section 2.5, we compared different package managers and what information they provided on their websites. A more thorough analysis of what data is available through the websites and API could be useful. It could contribute to making it easier to create tools that use this data and give the registries information about things they can improve.

More research on this artifact could also be done. The testing period for this tool was quite short and we believe it would be valuable to see the effects of using this tool over time. It could potentially lead to positive side effects such as new policies in the teams regarding what to do when including new software dependencies, or negative side effects such as notification fatigue. Testing it over a longer time period would also most likely contribute to a better assessment of the thresholds used in this tool.

The TC that were not implemented were presented in subsection 4.3.3, more research into how these TC can be implemented would be valuable. There are also other aspects of the tool that could be changed which will be discussed in subsection 5.7.1.

5.5 Implications for Practice

During the course of our research for this thesis, we have recognized specific areas that have the potential to enhance the software industry and its practices. Throughout the interview process with the Visma employees, a significant number of respondents emphasized the time constraints faced by developers and expressed appreciation for any tools that can effectively alleviate their workload. They also said that they think AutoTrust could make their assessment of NuGet packages more efficient compared to doing manual assessments.

Visma is a security-focused firm, yet a considerable number of respondents from the questionnaire and interviews expressed a lack of specific plans or guidelines for evaluating the security of new dependencies. We think that companies should do some reflections on the potential consequences of adding software dependencies without a planned approach and they should have some guidelines or tools for this process. If the company is working in the NuGet ecosystem they can use AutoTrust as it is not too intrusive in the developers' workflow as it only affects the download of packages process. If they are working in a different programming language ecosystem they can use other tools such as npq or OpenSSF Scorecard or they can use the TC proposed in Table 2.2 for a manual assessment.

The TC presented in Table 2.2 can be used as the basis for new security tools, or be used to improve existing tools. As discussed in section 4.3.8, there are some differences between AutoTrust, npq, OSSGadget, and OpenSSF Scorecard. Owners of existing security tools or creators of new tools can examine how AutoTrust, npq, OSSGadget, and OpenSSF Scorecard operate and use this knowledge to create new functionality. AutoTrust incorporates several validators that are not present in other tools, including checking if the package or its dependencies are deprecated, the number of direct and transitive dependencies, if the namespace is protected with a verified prefix, and popularity over time. Some of these differences between the tools might be due to the information not being available in the other programming language ecosystems, which also could be improved.

As mentioned in section 2.5, the package registries, linked to these programming language ecosystems, vary in the information they provide. We believe it would be advantageous for all package registries to include additional information such as popularity metrics, tools for package inspection, and a security evaluation. The availability of consistent information across registries would facilitate the development of tools and best practices that can cater to multiple programming language ecosystems.

In addition, we suggest that integrating the functionality offered by AutoTrust directly into either the "dotnet add package" command as a new flag or a new dedicated .NET command would yield notable advantages. This integration would increase the likelihood of developers utilizing the tool, thereby enhancing its overall effectiveness and impact within the software development process.

5.6 Limitations and Threats to Validity

In section 4.5 we presented the threats to the validity of the different evaluations we conducted. There were also some limitations that might have affected the validity of this whole research.

As we did the research in collaboration with Visma, a limitation is that we did not get insight from developers working at other companies. We could have sent out a questionnaire to other companies, included them in the interviews, or had some developers test the tool and collected their opinions. Visma is a large security-focused firm and respondents from both smaller companies and companies with other specializations could have provided a broader range of perspectives. There were also a relatively small number of respondents for the questionnaire and the interviews and including employees from other companies could have increased those numbers and strengthened the credibility of the results.

Even though we collaborated with Visma we did not stay physically in their offices. If we had stayed in their offices we believe it could have given us better insight into the developer's daily work by observing them. By observing them we could have seen how they add dependencies instead of relying on them telling us through the questionnaire. We think this also would have led to more respondents and less of a volunteer bias, as we believe it would have been easier to recruit participants that were more skeptical of the tool if they saw it first.

Another limitation of this research is that we were not able to do a direct comparison with a tool made for the NuGet ecosystem. It would have been beneficial to compare the tool we had made to other tools targeting the NuGet ecosystem instead of OpenSSF Scorecard that uses GitHub and GitLab data to assess projects.

The time demonstration of the artifact could have been improved by running a full Linux operating system and not just a WSL instance as this perhaps could have affected the results. AutoTrust relies on conducting multiple API queries, making it more susceptible to performance issues caused by a poor internet connection compared to the .NET command. Consequently, a slow or unreliable internet connection may introduce delays that can adversely impact the overall user experience. A limitation is that we did not consider how QA-4, performance, would be affected by increasing the number of validators or adding more extensive tests to the existing ones. This could have strongly affected the general perception the interviewees had regarding AutoTrust.

During the 100-packages demonstration, the selection of random packages might have been affected by the fact that we sorted based on relevance. The relevance ranking used in the NuGet search might have affected the 40 random packages from being appropriately randomized. In the 100-packages demonstration, we chose to include the top 50 most downloaded packages, 40 random packages, and 10 problematic packages. As we are using the popularity and widespread use as validators, it might be that the large difference in the ranking between the top packages and the random packages is attributed to the fact that the top 50 most downloaded packages are more popular. Another limitation of the 100-packages demonstration was that the 10 problematic packages were chosen based on issues we found. We probably ended up finding more commonly used packages that were better maintained than other poorly maintained packages, which we found in the random packages. This might be why one of the random packages got a low ranking of one star, while one of the problematic packages got a high ranking of four stars.

There are some limitations with AutoTrust as well. As mentioned, AutoTrust does not execute when developers add NuGet packages through the user interface within the Visual Studio IDE. Developers can then get affected by malicious NuGet packages if they add them using the Visual Studio package manager. Another critique of AutoTrust and this way of doing pre-install assessments of packages is that it is affected by the available data related to NuGet packages to do good assessments.

5.7 General Reflections

This section deals with general reflections that we discovered during the work on the thesis, that were not used to answer the RQs or does not necessarily fit under the standardized implications sections. This includes reflections around specific TC and how we found these, other use cases of AutoTrust, implementing AI, our thoughts about reducing attacks, and improvements to AutoTrust.

Not all of the TC are easy to implement. Some of them, such as trust criterion 8, "The component has detailed documentation", are hard to automate since they can be interpreted in multiple ways. With this example, what is considered detailed is hard to predefine. The fact that it is hard to predefine evaluations and that not all data was easily available through NuGet and GitHub led to a difference between the proposed TC and the implemented validators. We do believe it is valuable to have a list of TC that are programming language agnostic, but it has made it harder to evaluate their usefulness.

The TC presented in Table 2.2 were primarily found using academic sources. We could perhaps have found extra criteria if we had contacted more people working in the software industry. We asked people working at Visma if they had some more potential criteria both during the questionnaire and the interview, however, many of the respondents had not previously contemplated this aspect and consequently provided limited responses. As this is a more general question we could also have asked more developers either using a questionnaire or asking questions on internet forums for programmers to increase the possibility of finding new criteria.

Some of the Visma employees mentioned during the interviews other ways they thought AutoTrust could be useful. They mentioned that it could be incorporated into the CI/CD pipeline and that it could be used to analyze pull requests. Another way of using the tool is that the code contributor can include an image of the ranking in a pull request. If we had time to let the Visma employees use AutoTrust for a longer duration we would probably have discovered more ways it could have been useful.

As discussed in subsection 2.6.4 new AI tools such as ChatGPT can be incorporated into more software supply chain security tools. It will be interesting to see how this might affect the security landscape. AI is a double-edged sword as it can be used by both good and bad actors. It will make it easier for threat actors to create new attacks, but it can also be used to strengthen security. AI tools can also be used to assess source code like Socket AI does, or allow developers to query packages in a conversational manner like DroidGPT. LLMs like ChatGPT can be used to get a more in-depth and nuanced analysis of the checks used in the validators, and by doing this enhance the validators. We believe these kinds of tools have the potential to improve the assessment of licenses and documentation, if the developers are using good coding standards, if the code reviews of the project are of high quality, and be used for the assessment of installation scripts and source code.

AI is a double-edged sword as it can be used by both good and bad actors. It will make it easier for threat actors to create new attacks, but it can also be used to strengthen security.

Security tools play a crucial role in mitigating the success of attacks by making it more challenging for attackers to achieve their objectives. As a result, attackers experience a diminished return on investment, rendering their malicious activities less lucrative. This observation is evident in reports indicating that automated identification and removal of malicious packages using typosquatting techniques by package registries have compelled attackers to shift away from using typosquatting as an attack vector [32]. We strongly believe that widespread adoption of security tools like AutoTrust, OpenSSF Scorecard, npq, and OSSGadget within the development community would significantly reduce the profitability of software supply chain attacks, consequently leading to a decline in their occurrence. It is also worth noting that if the functionality these tools provide were directly incorporated into the package managers, a greater number of individuals would become aware of its significance and usefulness. This, in turn, would further diminish the attackers' return on investment and contribute to a decreased prevalence of such attacks.

5.7.1 Improving the Tool

Based on the interviews, the responses to the questionnaire, and personal reflections we have discovered some potential improvements for the AutoTrust tool. One of them is the option to have user-defined configurations of the thresholds. As the risk appetite might be different for different users it could have been valuable to allow users to define which validators they want to use, what thresholds they see fit, and which weight they considered the best for the validators. An option to add more customizable trust criteria could also be useful, such as trust criterion 4, "The company you are working for is already using the package in another project". We believe a general ranking of such a criterion might be hard to make, but if users are allowed to add their own it could be incorporated.

Another potential improvement for AutoTrust is to support multiple programming languages, as OSSGadget does. Putting effort into supporting multiple programming languages would most likely lead to greater interest in the tool. Supporting multiple programming languages in AutoTrust would address the diverse needs and preferences of developers working with various language ecosystems. It would allow developers to apply AutoTrust's functionality and security assessments to a wider range of projects, regardless of the programming language used. However, supporting multiple programming languages faces the challenge of inconsistent information from package registries, and relying solely on code hosting platforms like OSSGadget would limit data availability and lead to a less comprehensive package assessment.

During the interview some of the developers tried to use AutoTrust in new ways we did not expect. They saw the potential of including it into CI/CD pipelines or when reviewing pull requests. Another suggestion from the interviews was to add a dry-run flag that made AutoTrust only return the score and evaluation of the criteria, without prompting the user if they would like to add the package. Adding a dry-run flag could have made AutoTrust more flexible and such a solution would be more similar to OpenSSF Scorecard as it is made more as a standalone solution. Even though AutoTrust was intended to be used as an extension of the "dotnet add package" command it is interesting to see new potential use cases.

Three of the interviewees commented that AutoTrust runs as normal when they mistyped the package name and no package with that name existed. AutoTrust should just return an error if it does not find the package name on NuGet, as this would make the user experience better. It is, however, interesting to see how common mistyping is, which again is an advantage of using a tool such as AutoTrust as it will prevent the user from being exposed to a typosquatting attack when they mistype.

User behavior might also be a risk factor, not only the packages. When looking at third-party security and the process of adding new dependencies using the CLI there are some bad practices users might do that can increase the security risk. Users should not run package manager commands as superuser (sudo) as it grants unnecessary access to the filesystem [22]. When adding new dependencies the user might download an older version, and this older version might lack important security patches. Users might also be downgrading packages unintentionally during updates. AutoTrust could potentially warn about all of these undesirable user behaviors. However, it should be noted that explicitly specifying the version flag is required when downloading a version other than the latest stable release or when downgrading to an older version. We think that downgrading is often an unintentional action and should, at the very least, trigger a warning to alert the user.

The current solution of AutoTrust fails the known vulnerability validator if it finds a known vulnerability for the package version it checks. If it finds a vulnerability in a previous version that has been fixed it informs the developer about it. We do believe that if vulnerabilities are disclosed and fixed it can indicate a higher trust than having no recorded vulnerabilities. This would indicate that the developers are prioritizing their security. The existing version of AutoTrust lacks support for assigning positive scores. However, introducing the capability for positive scores could be a valuable enhancement, as it would enable a more nuanced ranking system. Based on the source code of the OSS-risk-calculator we saw that it gave a slightly worse score to packages that had no disclosed security vulnerabilities than packages that had disclosed and fixed them. However, we think it would be better to increase the security risk score for reported and fixed vulnerabilities since this would not punish packages that have not had any incidents.

OpenSSF Scorecard ranks its criteria on a scale from 0-10 and multiplies it by the weight. The fact that OpenSSF Scorecard ranks their criteria instead of just passing, warning, or failing means that the value given can be more nuanced. AutoTrust does not differentiate the security risk score based on how severely a criterion fails. For instance, AutoTrust does not assign a lower score to a NuGet package with five deprecated dependencies compared to a package with only a single deprecated dependency, even though the former may be considered worse. This difference in approach highlights the varying levels of nuance and granularity employed by the different systems in evaluating security risks and is something that could be improved with AutoTrust.

AutoTrust is relying on the NuGet API and OSV for data about vulnerabilities. An option to improve AutoTrust would be to allow more vulnerability databases or have the users choose the one they typically rely on to potentially detect more reported vulnerabilities. If AutoTrust supported more vulnerability databases, an aspect to consider would be that the same vulnerability could be reported multiple times. As discussed in subsection 2.6.2, a CVE id can be used to compare if the reported vulnerabilities from different vulnerability databases are in fact the same. To calculate the associated risk of the vulnerability, the CVSS string returned from the OSV database could have been used. AutoTrust could then differentiate the response based on the severity of the vulnerability.

Another improvement is to include the OpenSSF Scorecard ranking provided in their API. They allow users to query pre-calculated OpenSSF Scorecard rankings of large open-source projects on Github and Gitlab. This information could be used to give a more detailed assessment of the project they provide pre-calculated scores for. Not all projects are provided by the API, but it could still be valuable information for the NuGet packages when the information is available.

As discussed in subsection 5.3.4, the .NET package manager supports scanning all dependencies for vulnerabilities. An enhancement to AutoTrust would involve extending the known vulnerabilities validator to also scan all dependencies. OSV supports fetching vulnerability data for multiple packages, through its "/v1/querybatch" API endpoint. Using this endpoint would enable AutoTrust to scan the dependencies of the main package with just a single additional API call.

One additional API call would probably not affect the performance significantly.

Although the installation process received positive feedback from all individuals, there is room for further improvement to make it even more convenient. A possible improvement is to add AutoTrust as a NuGet package to make the installation process easier. By doing so, users would no longer need to manually clone and pack the project themselves.

One of the stated QAs for AutoTrust is security. We ran the OpenSSF Scorecard tool on the AutoTrust repository and got a ranking of 5.6/10. We should improve AutoTrust by making the changes recommended by the OpenSSF Scorecard tool. This is important as a tool for enhancing security should follow the best security practices and not be a security risk.

Chapter 6

Conclusion

This chapter concludes this thesis by summarizing the main contributions and findings from this work. It also presents some other relevant contributions, not directly related to AutoTrust, and our future work. The goal and the RQs used to guide the work on this thesis are:

Goal | Develop and evaluate a CLI tool to improve the pre-install assessment of third-party software packages.

RQ-1 | What information is useful for the assessment of security in third-party software packages prior to their installation?

RQ-2 | What are the advantages and disadvantages of the CLI pre-install tool for developers in the process of evaluating third-party software packages?

RQ-3 | How does the tool's assessment of third-party software packages' security risk compare to other ways of assessing these packages?

In this thesis, we presented the CLI security tool AutoTrust. The goal of this thesis was to develop and evaluate a CLI tool to improve the pre-install assessment of third-party software packages. AutoTurst is such a tool, made for the NuGet ecosystem. We designed it and tested it in collaboration with Visma.

The novelty of this research was the scientific development of a security tool for the pre-install phase with a specific focus on research and testing. We also focused more on the NuGet ecosystem than other scientific papers which have mainly focused on npm and PyPI [10]. Another novelty of this research is the presentation, use, and evaluation of new trust criteria, to help answer RQ-1. We demonstrated the feasibility of AutoTrust on 100 NuGet packages and measured the time using three common operating systems. After the demonstration, we evaluated AutoTrust by comparing it to manual assessment from computer science master's students and OpenSSF Scorecard, to answer RQ-3. The final part of the evaluation was having four employees at Visma try the tool and interview them, this was done to acquire their feedback on AutoTrust and to find advantages and disadvantages which helped answer RQ-2.

For RQ-1, we found that several of the TC and validators proposed in this thesis were well received and people working with software security believed them to be useful when assessing the security risks associated with third-party components. The information that got the most attention was to evaluate if the package was deprecated or used deprecated dependencies, had known vulnerabilities, was popular, had widespread use, had a license, or had multiple contributors.

When answering RQ-2, the primary advantages found were that tools like AutoTrust can lead to early detection of problematic packages and can be incorporated in a non-intrusive way into the developers' workflow. The main disadvantages are linked to the potential for developers to misinterpret the results of such tools and know how to respond to them. To mitigate this, developers must invest effort in comprehending the tool's assessment criteria and rationale.

To answer RQ-3, we did a direct comparison between AutoTrust and OpenSSF Scorecard by evaluating 100 NuGet packages. We found that AutoTrust had a more distinct ranking of the NuGet packages which helped better differentiate them. We also directly compared AutoTrust with the manual evaluation done by computer science students in their final year, where we found that AutoTrust considers more criteria and performs a more thorough analysis than the student did. However, tools like AutoTrust should be used in combination with manual assessment, which we and the study participants believed to be the optimal approach.

This work highlights the significance of conducting pre-installation evaluations of software packages with AutoTrust, which have proven to be useful for developers. We also found it preferred to combine the use of security tools with manual assessment for optimal results. Additionally, we have identified numerous valuable trust criteria and evaluated their usability and importance. There are still opportunities for further research to explore these trust criteria, as well as to investigate the long-time effectiveness of security tools for the pre-install phase.

6.1 Other Contributions

During our work on this thesis, we made some contributions other than creating and testing AutoTrust. As discussed in subsection 2.5.2, the PyPI package registry used to have the count of open pull requests and open issues aggregated into a single number since GitHub provided the information in that manner. We reported that we think it would be more insightful for developers to have this count split into two numbers[1]. The maintainers agreed and asked us to make the code changes. We created a pull request that got approved[2] and the information

[1] https://github.com/pypi/warehouse/issues/12971

[2] https://github.com/pypi/warehouse/pull/12985

regarding open issues and open pull requests is now split on the PyPI website.

As mentioned in subsection 4.5.2, we experienced issues when trying the OSSGadget tool. We tested the 18 most popular NuGet packages and all received a score of 1 indicating very high risk. The tool's inability to differentiate the packages renders it useless. We analyzed the code and reported where we believe the issue originated[3]. The maintainers of OSSGadget agreed that they believed the issue was originating from where we reported it, however, the last comments on the issue indicate that they might consider deprecating the tool altogether in favor of OpenSSF Scorecard.

When exploring previous incidents of malicious packages in the Maven package registry, discussed in subsection 2.5.3, we noticed that some packages, that should have been removed, were still possible to find. We reported this to the maintainers of Maven[4]. After investigating the issue they reported that only the option to download the packages was blocked, not searching for them in the registry. Upon our recommendation, they conveyed the suggestion to their development team to include an explicit warning for malicious packages, similar to the practice followed by npm.

6.2 Future Work

In section 5.4 we discussed areas that would benefit from further research and improvements by the research community. Then in subsection 5.7.1 we discussed further improvements to AutoTrust. In this section, we will present what work we will be doing with the tool and this thesis in the near future.

To generate interest in the tool we will write a Medium[5] article about the problem around third-party dependencies. We will explain the discoveries in this thesis about important trust criteria and validators, and finish by presenting how AutoTrust works. We will also file issues on AutoTrust's GitHub page for the suggested improvements mentioned in subsection 5.7.1, to make it possible for other developers to help us out. Finally, we will continue using the tool in our daily development practices to disclose further improvements and actively encourage other developers to use it as well.

[3] https://github.com/microsoft/OSSGadget/issues/150

[4] https://issues.sonatype.org/browse/MVNCENTRAL-7836

[5] https://medium.com/

Bibliography

[1] M. J. Hossain Faruk, M. Tasnim, H. Shahriar, M. Valero, A. Rahman, and F. Wu, “Investigating novel approaches to defend software supply chain attacks,” in *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2022, pp. 283–288. DOI: 10.1109/ISSREW55968.2022.00081.

[2] S. Raponi, M. Caprolu, and R. Di Pietro, “Beyond solarwinds: The systemic risks of critical infrastructures, state of play, and future directions,” in *ITASEC ’21: Italian Conference on Cyber Security*, 2021. [Online]. Available: http://star.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-2940/paper33.pdf.

[3] L. Sterle and S. Bhunia, “On solarwinds orion platform security breach,” in *2021 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/IOP/SCI)*, 2021, pp. 636–641. DOI: 10.1109/SWC50871.2021.00094.

[4] J. Yang, Y. Lee, and A. McDonald, “Solarwinds software supply chain security: Better protection with enforced policies and technologies,” in Jan. 2022, pp. 43–58, ISBN: 978-3-030-92316-7. DOI: 10.1007/978-3-030-92317-4_4.

[5] L. Tal, *Alert: peacenotwar module sabotages npm developers in the node-ipc package to protest the invasion of*, Accessed on 2023-03-22., Apr. 2022. [Online]. Available: https://snyk.io/blog/peacenotwar-malicious-npm-node-ipc-package-vulnerability/.

[6] N. Nehorai and B. Moussalli, *Attackers are starting to target .NET developers with malicious-code NuGet packages*, Accessed on 2023-03-22., Mar. 2023. [Online]. Available: https://jfrog.com/blog/attackers-are-starting-to-target-net-developers-with-malicious-code-nuget-packages/.

[7] P. Johannesson and E. Perjons, *An Introduction to Design Science*. Springer, Sep. 2014, vol. 1, pp. 1–197, ISBN: 978-3-319-10631-1. DOI: 10.1007/978-3-319-10632-8. [Online]. Available: https://link.springer.com/book/10.1007/978-3-319-10632-8.

[8] NTNU, *Structure in a empirical thesis*, Accessed on 2023-06-03., 2023. [Online]. Available: https://i.ntnu.no/academic-writing/strukture-in-a-empirical-thesis.

[9] S. Du, T. Lu, L. Zhao, B. Xu, X. Guo, and H. Yang, "Towards an analysis of software supply chain risk management," in *Proceedings of the World Congress on Engineering and Computer Science*, vol. 1, 2013.

[10] H. M. Morstøl and S. Rynning-Tønnesen, "Software supply chain security a systematic literature review," *Appendix A*, p. 78, 2022.

[11] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "Sok: Taxonomy of attacks on open-source software supply chains," in *2023 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 167–184. DOI: 10.1109/SP46215.2023.00010. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00010.

[12] J. D. Linton, S. Boyson, and J. Aje, "The challenge of cyber supply chain security to research and practice – an introduction," *Technovation*, vol. 34, no. 7, pp. 339–341, 2014, ISSN: 0166-4972. DOI: https://doi.org/10.1016/j.technovation.2014.05.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166497214000522.

[13] W. Axelrod, "Malware, "weakware," and the security of software supply chains," *CrossTalk, The Journal of Defense Software Engineering*, vol. 27, pp. 20–24, Mar. 2014.

[14] A. S. Markov and I. A. Sheremet, "Enhancement of confidence in software in the context of international security," in *CEUR Workshop Proceedings*, vol. 2603, 2019, pp. 88–92.

[15] T. Preston-Werner, *Semantic Versioning 2.0.0*, Accessed on 2023-02-03. [Online]. Available: https://semver.org/.

[16] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," *CoRR*, vol. abs/2005.09535, 2020. [Online]. Available: https://arxiv.org/abs/2005.09535.

[17] W. Enck and L. Williams, "Top five challenges in software supply chain security: Observations from 30 industry and government organizations," *IEEE Security & Privacy*, vol. 20, no. 2, pp. 96–100, 2022. DOI: 10.1109/MSEC.2022.3142338.

[18] N. P. Tschacher, "Typosquatting in programming language package managers," M.S. thesis, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln-Straße 30, 22527 Hamburg, Germany, Jun. 2016. [Online]. Available: https://incolumitas.com/2016/06/08/typosquatting-package-managers/.

[19] R. K. Vaidya, L. D. Carli, D. Davidson, and V. Rastogi, “Security issues in language-based sofware ecosystems,” *CoRR*, vol. abs/1903.02613, 2019. [Online]. Available: `http://arxiv.org/abs/1903.02613`.

[20] B. A., *Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies*, Accessed on 2023-03-22., Dec. 2021. [Online]. Available: `https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610`.

[21] A. Sharma, *Researcher hacks over 35 tech firms in novel supply chain attack*, Accessed on 2023-03-22., Feb. 2021. [Online]. Available: `https://www.bleepingcomputer.com/news/security/researcher-hacks-over-35-tech-firms-in-novel-supply-chain-attack/`.

[22] R. Kestilä, “Acknowledging the risks of open source dependencies to software supply chain security,” M.S. thesis, Tampereen yliopisto, Faculty of Information Technology and Communication Sciences, Kalevantie 4, 33100 Tampere, Finland, Jun. 2022. [Online]. Available: `https://trepo.tuni.fi/handle/10024/141165`.

[23] V. Mills and S. Butakov, “Security evaluation criteria of open-source libraries,” in *Computational Science and Its Applications – ICCSA 2022 Workshops*, Malaga, Spain: Springer-Verlag, 2022, pp. 422–435, ISBN: 978-3-031-10547-0. DOI: `10.1007/978-3-031-10548-7_31`. [Online]. Available: `https://doi.org/10.1007/978-3-031-10548-7_31`.

[24] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, “What are weak links in the npm supply chain?” In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022, pp. 331–340. DOI: `10.1145/3510457.3513044`.

[25] S. Dashevskyi, A. D. Brucker, and F. Massacci, “On the security cost of using a free and open source component in a proprietary product,” in *Engineering Secure Software and Systems*, J. Caballero, E. Bodden, and E. Athanasopoulos, Eds., Cham: Springer International Publishing, 2016, pp. 190–206, ISBN: 978-3-319-30806-7.

[26] M. Naedele and T. E. Koch, “Trust and tamper-proof software delivery,” ser. SESS ’06, Shanghai, China: Association for Computing Machinery, 2006, pp. 51–58, ISBN: 1595934111. DOI: `10.1145/1137627.1137636`. [Online]. Available: `https://doi.org/10.1145/1137627.1137636`.

[27] R. A. Martin, “Visibility & control: Addressing supply chain challenges to trustworthy software-enabled things,” in *2020 IEEE Systems Security Symposium (SSS)*, 2020, pp. 1–4. DOI: `10.1109/SSS47320.2020.9174365`.

[28] S. Lawrence Pfleeger, M. Libicki, and M. Webber, “I’ll buy that! cybersecurity in the internet marketplace,” *IEEE Security & Privacy*, vol. 5, no. 3, pp. 25–31, 2007. DOI: `10.1109/MSP.2007.64`.

[29] D. Yan, Y. Niu, K. Liu, Z. Liu, Z. Liu, and T. F. Bissyandé, "Estimating the attack surface from residual vulnerabilities in open source software supply chain," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 493–502. DOI: 10.1109/QRS54544.2021.00060.

[30] C. Thompson and D. Wagner, "A large-scale study of modern code review and security in open source projects," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE, Toronto, Canada: Association for Computing Machinery, 2017, pp. 83–92, ISBN: 9781450353052. DOI: 10.1145/3127005.3127014. [Online]. Available: https://doi.org/10.1145/3127005.3127014.

[31] N. Imtiaz and L. Williams, *Are your dependencies code reviewed?: Measuring code review coverage in dependency updates*, Accessed on 2023-03-22., 2022. [Online]. Available: https://arxiv.org/abs/2206.09422.

[32] L. Williams, "Trusting trust: Humans in the software supply chain loop," *IEEE Security & Privacy*, vol. 20, no. 5, pp. 7–10, 2022. DOI: 10.1109/MSEC.2022.3173123.

[33] G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, "Containing malicious package updates in npm with a lightweight permission system," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1334–1346. DOI: 10.1109/ICSE43902.2021.00121.

[34] A. Gkortzis, D. Feitosa, and D. Spinellis, "Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities," *Journal of Systems and Software*, vol. 172, p. 110 653, 2021, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110653. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301199.

[35] V. Jarukitpipat, K. Chhun, W. Wanprasert, C. Ragkhitwetsagul, M. Choetkiertikul, T. Sunetnanta, R. G. Kula, B. Chinthanet, T. Ishio, and K. Matsumoto, "V-achilles: An interactive visualization of transitive security vulnerabilities," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, Rochester, MI, USA: Association for Computing Machinery, 2023, ISBN: 9781450394758. DOI: 10.1145/3551349.3559526. [Online]. Available: https://doi.org/10.1145/3551349.3559526.

[36] M. Ohm, A. Sykosch, and M. Meier, "Towards detection of software supply chain attacks by forensic artifacts," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ser. ARES '20, Virtual Event, Ireland: Association for Computing Machinery, 2020, ISBN: 9781450388337. DOI: 10.1145/3407023.3409183. [Online]. Available: https://doi.org/10.1145/3407023.3409183.

[37] D.-L. Vu, F. Massacci, I. Pashchenko, H. Plate, and A. 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*, ser. ESEC/FSE 2021, Athens, Greece: Association for Computing Machinery, 2021, pp. 780–792, ISBN: 9781450385626. DOI: 10.1145/3468264.3468592. [Online]. Available: https://doi.org/10.1145/3468264.3468592.

[38] D. Jacobson, *NuGet Package Identity and Trust*, Accessed on 2023-04-18., Apr. 2017. [Online]. Available: https://devblogs.microsoft.com/nuget/package-identity-and-trust/.

[39] D. Chan, *Sunsetting Mercurial support in Bitbucket*, Accessed on 2023-03-22., Aug. 2020. [Online]. Available: https://bitbucket.org/blog/sunsetting-mercurial-support-in-bitbucket.

[40] G. Alamer and S. Alyahya, "Open source software hosting platforms: A collaborative perspective's review," *Journal of Software*, vol. 12, pp. 274–291, Apr. 2017. DOI: 10.17706/jsw.12.4.274-291.

[41] B. Cloud, *[BCLOUD-18541] Provide ability to favorite repositories and projects*, Accessed on 2023-03-22., Sep. 2019. [Online]. Available: https://jira.atlassian.com/browse/BCLOUD-18541.

[42] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," 2020.

[43] S. S. Team, *Sonatype Stops Software Supply Chain Attack Aimed at the Java Developer Community*, Accessed on 2023-03-22., Jan. 2021. [Online]. Available: https://blog.sonatype.com/malware-removed-from-maven-central.

[44] Microsoft, *Microsoft acquires GitHub - Stories*, Accessed on 2023-04-18., Jun. 2018. [Online]. Available: https://news.microsoft.com/announcement/microsoft-acquires-github/.

[45] Microsoft, *What is NuGet and what does it do?* Accessed on 2023-04-18., Dec. 2022. [Online]. Available: https://learn.microsoft.com/en-us/nuget/what-is-nuget.

[46] Lirantal, *GitHub - lirantal/npq: safely install packages with npm or yarn by auditing them as part of your install process*, Accessed on 2023-03-22., Dec. 2017. [Online]. Available: https://github.com/lirantal/npq.

[47] Microsoft, *GitHub - microsoft/OSSGadget: Collection of tools for analyzing open source packages.* Accessed on 2023-03-22., Mar. 2020. [Online]. Available: https://github.com/microsoft/OSSGadget.

[48] Microsoft, *GitHub - microsoft/ApplicationInspector: A source code analyzer built for surfacing features of interest and other characteristics to answer the question 'What's in the code?'* Accessed on 2023-03-22., Jan. 2020. [Online]. Available: https://github.com/Microsoft/ApplicationInspector.

[49] M. Scovetta, *OSS Gadget: Using oss-download*, Accessed on 2023-03-22., Dec. 2022. [Online]. Available: https://dev.to/scovetta/oss-gadget-using-oss-download-1gi8.

[50] O. S. S. Foundation, *GitHub - ossf/scorecard: OpenSSF Scorecard - Security health metrics for Open Source*, Accessed on 2023-05-16., Oct. 2020. [Online]. Available: https://github.com/ossf/scorecard.

[51] M. Lysenko, *Introducing Socket AI – ChatGPT-Powered Threat Analysis - Socket*, Accessed on 2023-04-18., Mar. 2023. [Online]. Available: https://socket.dev/blog/introducing-socket-ai-chatgpt-powered-threat-analysis.

[52] E. Labs, *DroidGPT - AI-Assisted open source software selection*, Accessed on 2023-04-19., 2023. [Online]. Available: https://www.endorlabs.com/droidgpt.

[53] P. Inglesant and A. Sasse, "The true cost of unusable password policies," vol. 1, Apr. 2010, pp. 383–392. DOI: 10.1145/1753326.1753384.

[54] L. Zhang-Kennedy, S. Chiasson, and P. van Oorschot, "Revisiting password rules: Facilitating human management of passwords," in *2016 APWG Symposium on Electronic Crime Research (eCrime)*, 2016, pp. 1–10. DOI: 10.1109/ECRIME.2016.7487945.

[55] C. Herley, "So long, and no thanks for the externalities: The rational rejection of security advice by users," in *Proceedings of the 2009 Workshop on New Security Paradigms Workshop*, ser. NSPW '09, Oxford, United Kingdom: Association for Computing Machinery, 2009, pp. 133–144, ISBN: 9781605588452. DOI: 10.1145/1719030.1719050. [Online]. Available: https://doi.org/10.1145/1719030.1719050.

[56] T. M. S. do Amaral and J. J. C. Gondim, "Integrating zero trust in the cyber supply chain security," in *2021 Workshop on Communication Networks and Power Systems (WCNPS)*, 2021, pp. 1–6. DOI: 10.1109/WCNPS53648.2021.9626299.

[57] R. J. Wieringa, *Design science methodology for information systems and software engineering*. Springer, May 2014, pp. 1–332, ISBN: 978-3-662-43838-1. DOI: 10.1007/978-3-662-43839-8. [Online]. Available: https://link.springer.com/book/10.1007/978-3-662-43839-8.

[58] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[59] U. E. Chigbu, "Visually hypothesising in scientific paper writing: Confirming and refuting qualitative research hypotheses using diagrams," *Publications*, vol. 7, no. 1, 2019. [Online]. Available: `https://www.mdpi.com/2304-6775/7/1/22`.

[60] O. Gelo, D. Braakmann, and G. Benetka, "Quantitative and qualitative research: Beyond the debate," *Integrative psychological and behavioral science*, vol. 42, pp. 266–290, 2008.

[61] R. Walpole, R. Myers, S. Myers, and K. Ye, *Probability and Statistics for Engineers and Scientists*, 9th ed. Pearson Education, 2011, ISBN: 9780321629111.

[62] B. J. Oates, M. Griffiths, and R. McLean, *Researching information systems and computing*. Sage, 2022.

[63] N. U. of Science and Technology, *Collection of personal data for research projects - Kunnskapsbasen - NTNU*, Accessed on 2023-03-06. [Online]. Available: `https://i.ntnu.no/wiki/-/wiki/English/Collection+of+personal+data+for+research+projects`.

[64] *Lov om behandling av personopplysninger (personopplysningsloven)*, Accessed on 2023-03-06., 2018. [Online]. Available: `https://lovdata.no/dokument/NL/lov/2018-06-15-38`.

[65] J. Harush, *How 140k NuGet, NPM, and PyPi Packages Were Used to Spread Phishing Links*, Accessed on 2023-03-22., Dec. 2022. [Online]. Available: `https://checkmarx.com/blog/how-140k-nuget-npm-and-pypi-packages-were-used-to-spread-phishing-links/`.

[66] Sonatype, *Sonatype Finds 700% Average Increase in Open Source Supply Chain Attacks*, Accessed on 2023-03-22., Sep. 2022. [Online]. Available: `https://www.sonatype.com/press-releases/sonatype-finds-700-average-increase-in-open-source-supply-chain-attacks`.

[67] N. Zahan, P. Kanakiya, B. Hambleton, S. Shohan, and L. Williams, "Openssf scorecard: On the path toward ecosystem-wide automated security metrics," Jan. 2023.

[68] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley, Sep. 2012, vol. 5. [Online]. Available: `https://dl.acm.org/doi/book/10.5555/2392670`.

[69] SPDX, *SPDX License List*, Accessed on 2023-04-27., Feb. 2023. [Online]. Available: `https://spdx.org/licenses/`.

[70] S. E. Team, *Top open source licenses and legal risk for developers*, Accessed on 2023-04-27., Jul. 2022. [Online]. Available: `https://www.synopsys.com/blogs/software-security/top-open-source-licenses/`.

[71] Microsoft, *NuGet.org starts repo-signing packages*, Accessed on 2023-04-26., Aug. 2018. [Online]. Available: `https://devblogs.microsoft.com/nuget/introducing-repository-signatures/`.

[72] Microsoft, *Install and use a NuGet package in Visual Studio (Windows only)*, Accessed on 2023-06-05., Feb. 2023. [Online]. Available: `https://learn.microsoft.com/en-us/nuget/quickstart/install-and-use-a-package-in-visual-studio`.

[73] G. Guest, A. Bunce, and L. Johnson, "How many interviews are enough? an experiment with data saturation and variability," *Field methods*, vol. 18, no. 1, pp. 59–82, 2006.

[74] F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," *MIS Quarterly*, vol. 13, no. 3, pp. 319–340, 1989, ISSN: 02767783. [Online]. Available: `http://www.jstor.org/stable/249008`.

[75] Microsoft, *Manage package trust boundaries*, Accessed on 2023-05-16., Jan. 2021. [Online]. Available: `https://learn.microsoft.com/en-us/nuget/consume-packages/installing-signed-packages`.

[76] N. McDonald and S. Goggins, "Performance and participation in open source software on github," in *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '13, Paris, France: Association for Computing Machinery, 2013, pp. 139–144, ISBN: 9781450319522. DOI: `10.1145/2468356.2468382`. [Online]. Available: `https://doi.org/10.1145/2468356.2468382`.

[77] Microsoft, *dotnet list package*, Accessed on 2023-06-03., Feb. 2023. [Online]. Available: `https://learn.microsoft.com/en-us/dotnet/core/tools/dotnet-list-package`.

Appendix A

Sikt Information

Sikt Notification Form, Sikt's assessment of processing of personal data, and Information letter about consent.

A.1 Sikt Notification Form

This section contains the notification form that was sent to Sikt for approval. The notification form contains information about the project and information about what data the project is intended to collect.

Sikt

Notification form / Automatic software dependency auditing using trust criteria / Export

Notification Form

Reference number
622074

Which personal data will be processed?

- Email address, IP address or other online identifier
- Photographs or video recordings of people
- Sound recordings of people

Project information

Project title

Automatic software dependency auditing using trust criteria

Project description

Vi skal som en del av masteroppgaven vår på NTNU utvikle en løsning for å bedre sjekke sikkerheten på forsyningskjeden til software. Etter at denne er utviklet skal vi så teste ut løsningen på ulike teams hos bedriften Visma og intervjue dem angående deres inntrykk av vår løsning. Vi vil først vise frem løsningen til de ulike teamene og vil deretter la dem selv teste ut verktøyet. Etter at dette er gjennomført vil vi avholde intervjuer, trolig digitalt, hvor vi innhenter informasjon om deres inntrykk av løsningen.

Explain why it is necessary to process personal data in the project

Personopplysningene vil bli brukt for å samle inn informasjonen om hvordan intervjuobjektene brukte verktøyet og deres inntrykk av det. Dataen vil samles og anonymiseres i masteroppgaven. Informasjonen vil kun lagres for å kunne ettergå informasjonen fra intervjuene.

External funding
Ikke utfyllt
Type of project
Student project, Master’s thesis

Contact information, student
Hallvard Molin Morstøl, hallvarm@stud.ntnu.no, tlf: +4794155709

Data controller

Data controller (institution responsible for the project)
Norges teknisk-naturvitenskapelige universitet / Fakultet for informasjonsteknologi og elektroteknikk (IE) / Institutt for datateknologi og informatikk

Project leader (academic employee/supervisor or PhD candidate)
Daniela Soares Cruzes, daniela.s.cruzes@ntnu.no, tlf: +4794249891

Will the responsibility of the data controller be shared with other institutions (joint data controllers)?
No

Sample 1

Describe the sample

Utvalget vil bestå av ansatte som jobber med programmering hos Visma AS

Describe how you will recruit or select the sample

Visma AS vil internt velge de teamene som de anser som best egnet til å teste verktøyet.

Age
21 - 70

Personal data relating to sample 1

- Email address, IP address or other online identifier
- Photographs or video recordings of people
- Sound recordings of people

How will you collect data relating to sample 1?

Personal interview

Attachment

Personlig Intervju.docx

Legal basis for processing general categories of personal data
Consent (General Data Protection Regulation art. 6 nr. 1 a)

Online survey

Attachment

Elektronisk spørreskjema.docx

Legal basis for processing general categories of personal data
Consent (General Data Protection Regulation art. 6 nr. 1 a)

Information for sample 1

Will you inform the sample about the processing of their personal data?
Yes

How?
Written information (on paper or electronically)

Information letter

information_letter_about_consent.docx

Third Persons

Will you be processing data relating to third persons?
No

Documentation

How will consent be documented?

- Electronically (email, e-form, digital signature)

How can consent be withdrawn?

Ved å sende epost eller ved å kontakte studentene eller veileder på telefon.

How can data subjects get access to their personal data or have their personal data corrected or deleted?

Ved å sende epost eller ved å kontakte studentene eller veileder på telefon.

Total number of data subjects in the project
1-99

Approvals

Will you obtain any of the following approvals or permits for the project?

Ikke utfyllt

Processing

Where will the personal data be processed?

- Computer belonging to the data controller

Who will be processing/have access to the collected personal data?

- Project leader
- Student (student project)

Will the collected personal data be transferred/made available to a third country or international organisation outside the EU/EEA?

No

Information Security

Will directly identifiable data be stored separately from the rest of the collected data (e.g. in a scrambling key)?

Yes

Which technical and practical measures will be used to secure the personal data?

- Record of changes
- Multi-factor authentication
- Restricted access
- Personal data will be stored in encrypted form

Duration of processing

Project period

01.01.2023 - 31.08.2023

What happens to the data at the end of the project?

Personal data will be anonymised (deleting or rewriting identifiable data)

Which anonymization measures will be taken?

- The identification key will be deleted
- Personally identifiable information will be removed, re-written or categorized
- Any sound or video recordings will be deleted

Will the data subjects be identifiable (directly or indirectly) in the thesis/publications from the project?

No

Additional information

A.2 Sikt's Assessment of Processing of Personal Data

The assessment of the research project provided by Sikt.

Sikt

Notification form / Automatic software dependency auditing using trust criteria / Assessment

Assessment of processing of personal data

Reference number	Assessment type	Date
622074	Automatic ?	28.01.2023

Project title
Automatic software dependency auditing using trust criteria

Data controller (institution responsible for the project)
Norges teknisk-naturvitenskapelige universitet / Fakultet for informasjonsteknologi og elektroteknikk (IE) / Institutt for datateknologi og informatikk

Project leader
Daniela Soares Cruzes

Student
Hallvard Molin Morstøl

Project period
01.01.2023 - 31.08.2023

Categories of personal data
General

Legal basis
Consent (General Data Protection Regulation art. 6 nr. 1 a)

The processing of personal data is lawful, so long as it is carried out as stated in the notification form. The legal basis is valid until 31.08.2023.

Notification Form

Basis for automatic assessment
The notification form has received an automatic assessment. This means that the assessment has been automatically generated based on the information registered in the notification form. Only processing of personal data with low risk for data subjects receive an automatic assessment. Key criteria are:

- Data subjects are over the age of 15
- Processing does not include special categories of personal data;
  - Racial or ethnic origin
  - Political, religious or philosophical beliefs
  - Trade union membership
  - Genetic data
  - Biometric data to uniquely identify an individual
  - Health data
  - Sex life or sexual orientation
- Processing does not include personal data about criminal convictions and offences
- Personal data shall not be processed outside the EU/EEA, and no one located outside the EU/EEA shall have access to the personal data
- Data subjects will receive information in advance about the processing of their personal data.

Information provided to data subjects (samples) must include

- The identity and contact details of the data controller
- Contact details of the data protection officer (if relevant)
- The purpose for processing personal data
- The scientific purpose of the project
- The legal basis for processing personal data
- What type of personal data will be processed and how it will be collected, or from where it will be obtained
- Who will have access to the personal data (categories of recipients)

- How long the personal data will be processed
- The right to withdraw consent and other rights

We recommend using our template for the information letter.

Information security
You must process the personal data in accordance with the storage guide and information security guidelines of the data controller. The institution is responsible for ensuring that the conditions of Article 5(1)(d) accuracy and 5(1)(f) integrity and confidentiality, as well as Article 32 security, are met.

A.3 Data Management Plan

The data management plan (in Norwegian) is used as a guide to control that the data collected during the research is handled in a correct manner.

DATAHÅNDTERINGSPLAN

Automatic software dependency auditing using trust criteria

Vi skal som en del av masteroppgaven vår på NTNU utvikle en løsning for å bedre sjekke sikkerheten på forsyningskjeden til software. Etter at denne er utviklet skal vi så teste ut løsningen på ulike teams hos bedriften Visma og intervjue dem angående deres inntrykk av vår løsning. Vi vil først vise frem løsningen til de ulike teamene og vil deretter la dem selv teste ut verktøyet. Etter at dette er gjennomført vil vi avholde intervjuer, trolig digitalt, hvor vi innhenter informasjon om deres inntrykk av løsningen.

Fagfelt
Teknologi

Forskningsansvarlig institusjon
Norges teknisk-naturvitenskapelige universitet / Fakultet for informasjonsteknologi og elektroteknikk (IE) / Institutt for datateknologi og informatikk

Prosjektvarighet
01.01.2023 — 31.08.2023

Formål
Ettersom vi har utvikling en løsning for å sikre forsyningskjeden til software ønsker vi å evaluere nyttigheten av denne.

Nytteverdi
Programvareutviklere og andre som jobber med datasikkerhet.

Etiske retningslinjer

- Generelle forskningsetiske retningslinjer
- Naturvitenskap og teknologi

Personlig intervju

Beskrivelse
Svar på intervju der deltakere har brukt løsningen som ble utviklet som en del av prosjektet.

Datatype
Video, Lyd

Språk
Norsk, Engelsk

Nøkkelord
Sikkerhet, Datateknologi, Tredjepartskode, Security , Software security, Dependencies

Data om personer
Ja

Er det noen andre grunner til at dataene dine trenger ekstra beskyttelse?
Nei

Kategorier av personopplysninger
Alminnelige

Utvalgets størrelse
30

Konfidensialitetsklassifisering
Intern

Innsamlingsperiode
01.02.2023 — 31.05.2023

Innsamlingsenheter

- 1. NTNU Zoom

- 2. NTNU Microsoft Teams

Datakvalitet

Bruker løsningenes opptaksfunksjonalitet med god internettdekning.

Metode

Intervju

Størrelse

10000 MB

Kommentar

Cirka 30 videointervjuer på 30 minutter.

Format

mp4

Lagring

- 02. NTNU Personlig hjemmeområde (M:-disk)
- 05. NTNU Office 365 (SharePoint, Teams, Onedrive)

Overføring

- 1. NTNU e-post med AIP
- 2. Office 365 (SharePoint, Teams, Onedrive)
- 3. Unit FileSender

Arkivering

Nei

Elektronisk spørreskjema

Beskrivelse

Svar på elektronisk spørreskjema der deltakere har brukt løsningen som ble utviklet som en del av prosjektet.

Datatype

Tekst

Språk

Norsk, Engelsk

Nøkkelord

Sikkerhet, Software secuirty, Security, Tredjepartskode, Third party dependencies

Data om personer

Ja

Er det noen andre grunner til at dataene dine trenger ekstra beskyttelse?

Nei

Kategorier av personopplysninger

Alminnelige

Utvalgets størrelse

30

Konfidensialitetsklassifisering

Intern

Innsamlingsperiode

01.01.2023 — 31.05.2023

Innsamlingsenheter

- 6. Annen innsamlingsmåte

Datakvalitet

Nettskjema ved NTNU

Metode

Selvadministrerende spørreskjema

Beskrivelse

Nettskjema med spørsmål for å svare på.

Størrelse

100 MB

Kommentar

30 pdfer

Format

pdf

Lagring

- 02. NTNU Personlig hjemmeområde (M:-disk)
- 05. NTNU Office 365 (SharePoint, Teams, Onedrive)

Overføring

- 1. NTNU e-post med AIP
- 2. Office 365 (SharePoint, Teams, Onedrive)

Arkivering

Nei

Appendix B

Data Gathering

This section contains the forms used for data gathering, some of the results, and a one-pager we wrote to increase the number of participants in the interviews.

B.1 Information Letter About Consent

This letter contains information given to the research participants about the data collection and the participants' rights. It is used for the participants to give consent before the interview.

Interview consent form

Purpose of the project
You are invited to participate in a research project where the main purpose is to evaluate a security tool to assess third-party software.

The project is a master's thesis made by two students at NTNU to evaluate trust criteria used to evaluate third-party software dependencies. You will be using a Command Line Interface tool we have made in your daily work, and we would like to ask some questions about your experience using this tool.

Which institution is responsible for the research project?
The Norwegian University of Science and Technology is responsible for the project (the data controller) and it is done in collaboration with Visma AS.

Why are you being asked to participate?
You are working at Visma and are using the relevant programming language and package manager in your daily work.

We have obtained your contact information from our contact at Visma that is helping us with the project.

What does participation involve for you?

You will participate in an interview. The information that will be collected is your answers. We will use a live transcription tool and record the sound and video. The transcription from the tool and the recording will be used to create a more complete transcription of the interview. The video and sound recording will be deleted and the complete transcription will be sent to the participants for validation.

The interview will take approximately 30 minutes.

Participation is voluntary
Participation in the project is voluntary. If you chose to participate, you can withdraw your consent at any time without giving a reason. There will be no negative consequences for you if you chose not to participate or later decide to withdraw.

Your personal privacy – how we will store and use your personal data
We will only use your personal data for the purpose(s) specified here and we will process your personal data in accordance with data protection legislation (the GDPR).

Only the institute responsible will have access to the data.
The data will be stored on NTNU's encrypted servers.
The participants will not be recognizable in publications and the data will be anonymized.

What will happen to your personal data at the end of the research project?
The planned end date of the project is 31.08.2023. The personal data, including any digital recordings, will be deleted before the end of the project.

Your rights

So long as you can be identified in the collected data, you have the right to:

- access the personal data that is being processed about you
- request that your personal data is deleted
- request that incorrect personal data about you is corrected/rectified
- receive a copy of your personal data (data portability), and
- send a complaint to the Norwegian Data Protection Authority regarding the processing of your personal data

What gives us the right to process your personal data?

We will process your personal data based on your consent.

Based on an agreement with the Norwegian University of Science and Technology, The Data Protection Services of Sikt – Norwegian Agency for Shared Services in Education and Research has assessed that the processing of personal data in this project meets requirements in data protection legislation.

Where can I find out more?

If you have questions about the project, or want to exercise your rights, contact:

- NTNU IDI via Professor Daniela Soares Cruzes (daniela.s.cruzes@ntnu.no)
- Our Data Protection Officer: Thomas Helgesen (thomas.helgesen@ntnu.no)

If you have questions about how data protection has been assessed in this project by Sikt, contact:

- email: (personverntjenester@sikt.no) or by telephone: +47 73 98 40 40.

Yours sincerely,

Project Leader	Students
Daniela Soares Cruzes	Sverre Rynning-Tønnesen & Hallvard Molin Morstøl

Consent form

I have received and understood information about the project Automatic software dependency auditing using trust criteria and have been given the opportunity to ask questions. I give consent to participate in the interview.
I give consent for my personal data to be processed until the end of the project.

(Signed by participant, date)

B.2 Questionnaire to Visma Employees

This section contains the questions that were asked in the questionnaire to the security experts at Visma.

Trust Criteria Evaluation

You are invited to participate in a research project where the main purpose is to evaluate a security tool to assess third party software.
The project is a master’s thesis made by two students at NTNU to evaluate trust criteria used to evaluate third-party software dependencies.
The students:

- Hallvard Molin Morstøl (hallvarm@stud.ntnu.no)
- Sverre Rynning-Tønnesen (sverrery@stud.ntnu.no)

Supervisors:

- Daniela Soares Cruzes (daniela.soares.cruzes@visma.com)
- Monica Iovan (monica.iovan@visma.com)

The estimated time for filling out this form is 10 minutes.

Information about consent

Which institution is responsible for the research project?
Norwegian University of Science and Technology is responsible for the project (data controller), and is done collaborating with Visma AS.

Why are you being asked to participate?
You are working at Visma and have programming experience.
We have obtained your contact information from our contacts at Visma that is helping us with the project.

What does participation involve for you?
You will participate in an online-survey. The information that will be collected are your answers to the questions.

Participation is voluntary
Participation in the project is voluntary. All information about you will be made anonymous. There will be no negative consequences for you if you choose not to participate.
The planned end date of the project is 31.08.2023.

What gives us the right to process your data?
We will process your data based on your consent.
Based on an agreement with the Norwegian University of Science and Technology, The Data Protection Services of Sikt – Norwegian Agency for Shared Services in Education and Research has assessed that the processing of data in this project meets requirements in data protection legislation.

Where can I find out more?
If you have questions about the project, contact:

- NTNU IDI via Professor Daniela Soares Cruzes (daniela.soares.cruzes@visma.com)
- Our Data Protection Officer: Thomas Helgesen (thomas.helgesen@ntnu.no)

If you have questions about how data protection has been assessed in this project by Sikt, contact:

- Email: (personverntjenester@sikt.no) or by telephone: +47 73 98 40 40.

I have received and understood information about the project Automatic software dependency auditing using trust criteria and consent to participate:

Consent [ ]
Do not consent [ ]

Programming Experience

Which package managers and related programming language(s) do you have experience with using?
NuGet, working in .NET with C# or F# [ ]
npm, working with node.js writing either JS or TS [ ]
pip, installing packages from PyPI and writing in python [ ]
Maven, Gradle, or Ivy for Java [ ]
RubyGems, writing in Ruby [ ]
Other: ____ [ ]

What part of the software stack do you mainly work on?

Frontend [ ]
Backend [ ]
Frontend & Backend [ ]
Other: ____ [ ]

For have many years have you been programming as part of work?
0-4 years [ ]
5-9 years [ ]
10-14 years [ ]
15+ years [ ]
Other: ____ [ ]

Do you host specific versions of dependencies yourself/internally at Visma or do you use the central package registry (like npmjs, NuGet, maven, PyPI, etc...)?
Yourself/internally [ ]
Central package manager [ ]
Both [ ]

How are you and your team currently dealing with finding dependencies to trust and adding them to the project? (Please check all the boxes that apply)
Add dependencies when they are needed without doing too much research [ ]
Adding dependencies after doing an assessment on your own without guidelines [ ]
Adding dependencies after doing an assessment on your own but following given guidelines [ ]
Discuss the security of the package with one or multiple team members and getting consensus [ ]
Proposing a package and have an security expert evaluate it before using it [ ]
I have never added a new dependency [ ]
There are no guidelines for how to add dependencies [ ]
Other: ____ [ ]

Trust Criteria evaluation

Below we have listed 29 trust criteria used to decide whether to trust a package or not. Please rank how valuable you think they are in determining if you should include a third-party dependency in your project (1-5) based on your first impression?

The ranking system describes how important you think the trust criteria is for evaluating the security of a dependency:
1 - Does not matter
2 - Not necessary
3 - Not necessary but insightful
4 – Valuable
5 - Crucial

NOTE: The complexity of the different trust criteria are quite varying but do not take this into account when evaluating them.

Time and Human Relations

	1-Does not matter	2-Not necessary	3-Not necessary but insightful	4 -Valuable	5-Crucial
1. The component has been in widespread use for a considerable amount of time					
2. The component is widely used/popular					
3. Satisfactory time since the latest update was published					
4. The company you are working for is already using the package in another project					

Licensing and documentation

	1-Does not matter	2-Not necessary	3-Not necessary but insightful	4 -Valuable	5-Crucial
5. The component has a software certification from a certified provider					
6. The component provides a hash or signature that can be used to make sure that the software has not been tampered with					
7. The component provides a standard or well-written license					

8. The component has good documentation					

Maintainers

	1-Does not matter	2-Not necessary	3-Not necessary but insightful	4 -Valuable	5-Crucial
9. The component has an adequate number of maintainers and/or contributors					
10. The component is being developed by an active maintainer domain					
11. The maintainers of the component are not overloaded					
12. The maintainers of the component are using a programming language that they are familiar with					
13. No maintainer accounts are associated with an expired email domain					
14. The package has not changed ownership recently					
15. The maintainers and suppliers of the component are trustworthy (maintainers use their real identity, email, country, former projects, etc.)					

Maintenance lifecycle

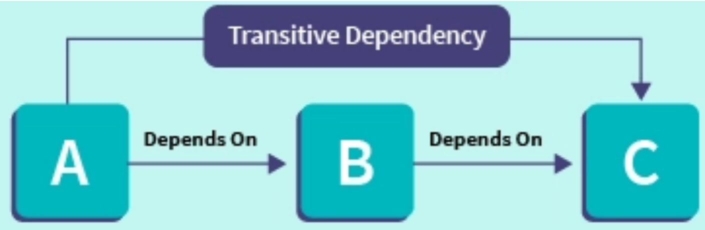
	1-Does not matter	2-Not necessary	3-Not necessary but insightful	4 -Valuable	5-Crucial
16. The component's maintenance lifecycle is up to date (uses tools for automated dependency updates, the number of open issues and pull requests are relatively low, etc.)					

17. There are a small number of open issues, and they are not very old					
18. The developers of the component are using automated code analysis to review the code					
19. The code reviews in the project are of high quality					

Reported problem & deprecation

	1-Does not matter	2-Not necessary	3-Not necessary but insightful	4 -Valuable	5-Crucial
20. The project does not have reported security vulnerabilities					
21. There is no history of prior harmful effects associated with the component					
22. The component is not deprecated					
23. The component does not depend on deprecated packages					

Other



	1-Does not matter	2-Not necessary	3-Not necessary but insightful	4 -Valuable	5-Crucial
24. The size of the repository (files or bytes)					
25. The project has few direct and transitive dependencies					
26. The component does not contain installation scripts					

27. The component has a name that does not resemble that of a popular package					
28. No difference between the source code and the package					
29. Access to the source code is provided					

Final Questions

Are there any trust criteria that you use or think are valuable, that were not mentioned?

Answer: ____

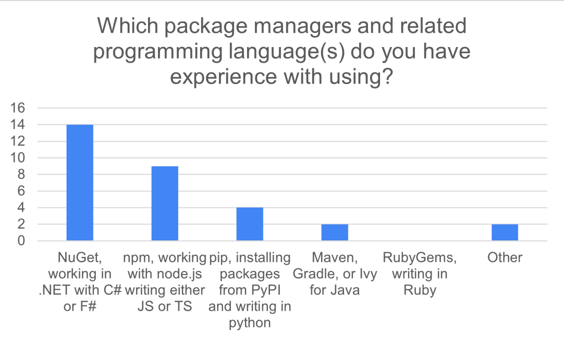
Any other thoughts (about the proposed trust criteria, this questionnaire, etc.)?

Answer: ____

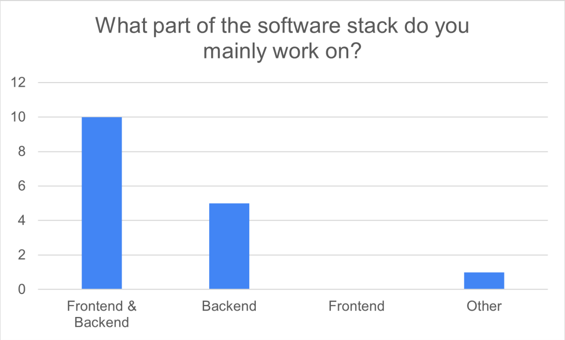
B.3 Replies to the Questionnaire from Visma Employees

These are the result of the 15 replies we got from the questionnaire sent to the Visma employees. For the TC it shows the distribution, the median, and the mode.

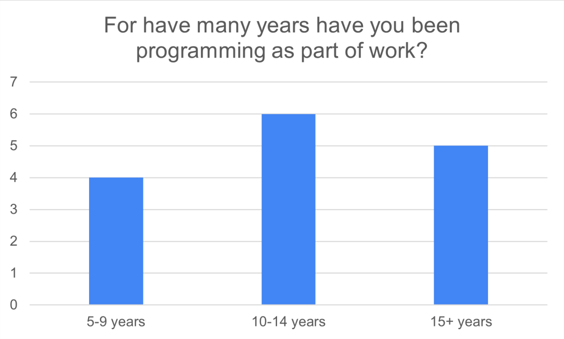
Intro Questions



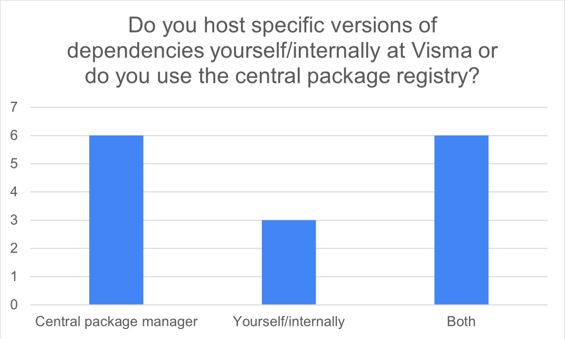
(a) Question 1



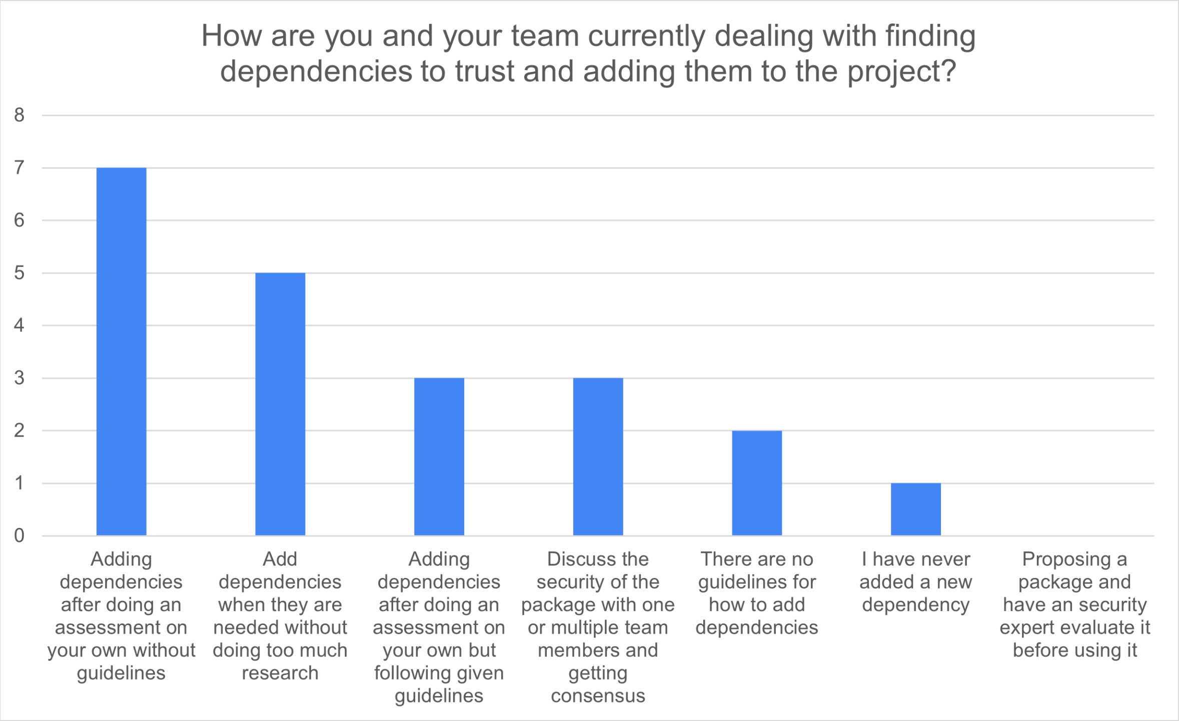
(b) Question 2



(c) Question 3

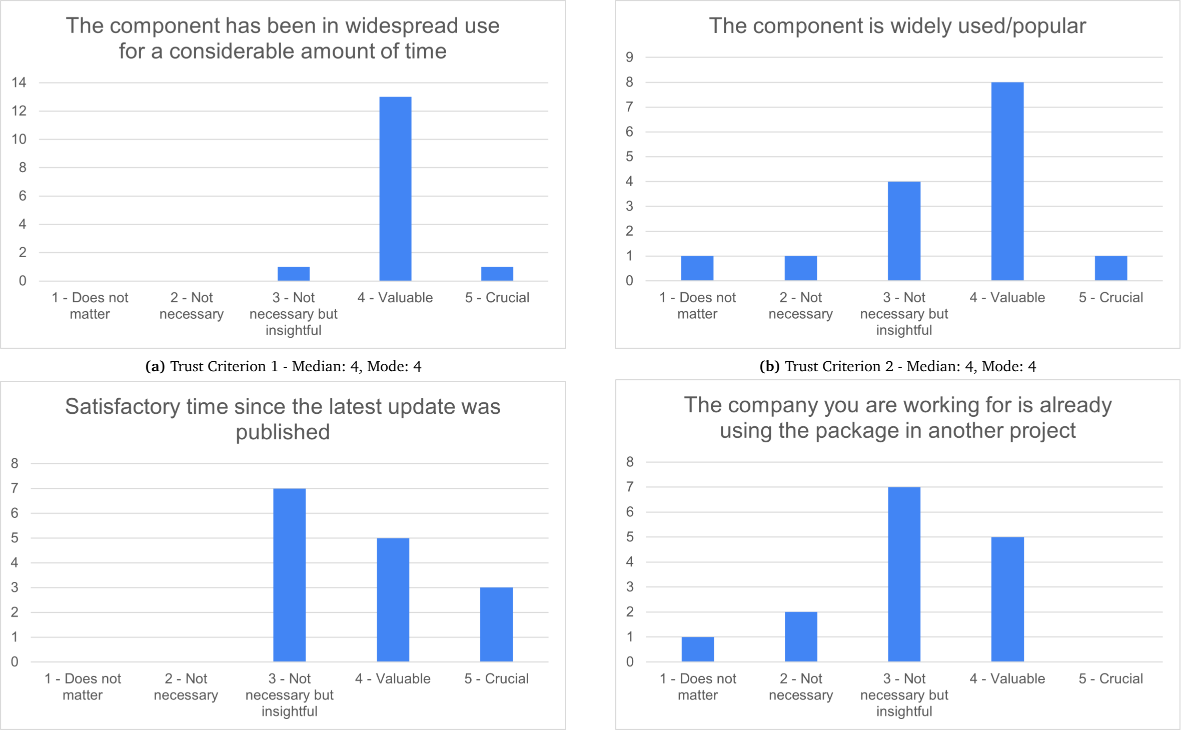


(d) Question 4



(e) Question 5

Time and Human Relations



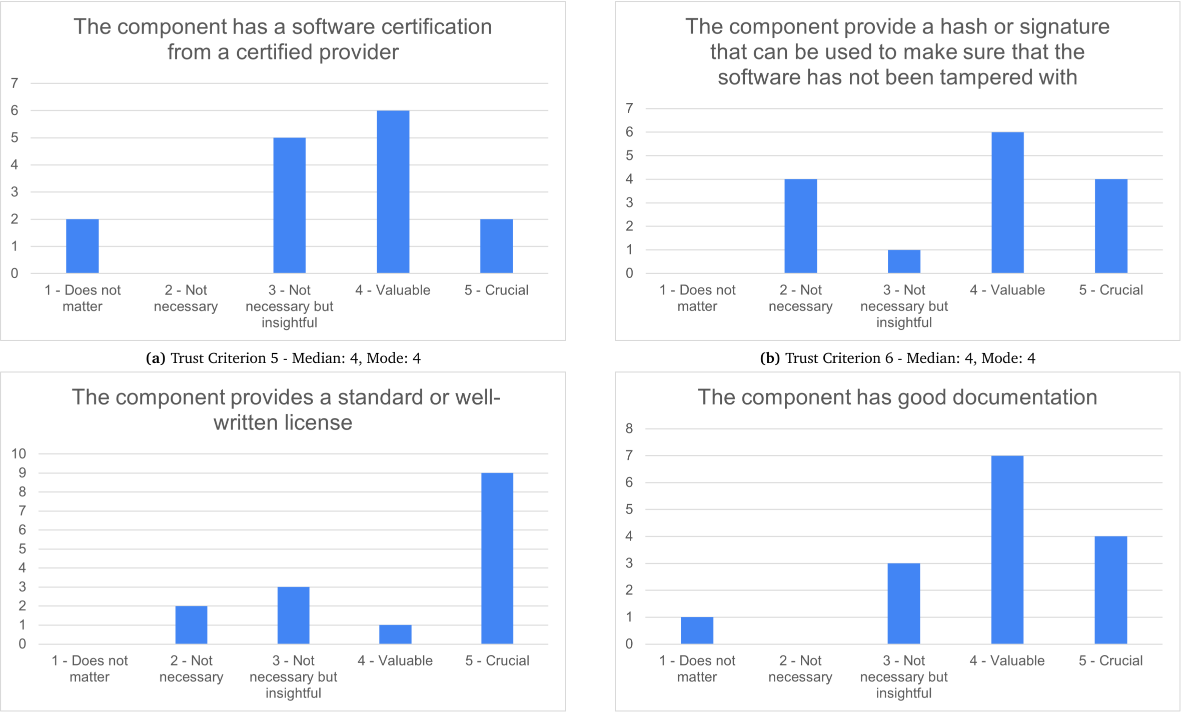
(a) Trust Criterion 1 - Median: 4, Mode: 4

(b) Trust Criterion 2 - Median: 4, Mode: 4

(c) Trust Criterion 3 - Median: 4, Mode: 3

(d) Trust Criterion 4 - Median: 3, Mode: 3

Licensing and Documentation



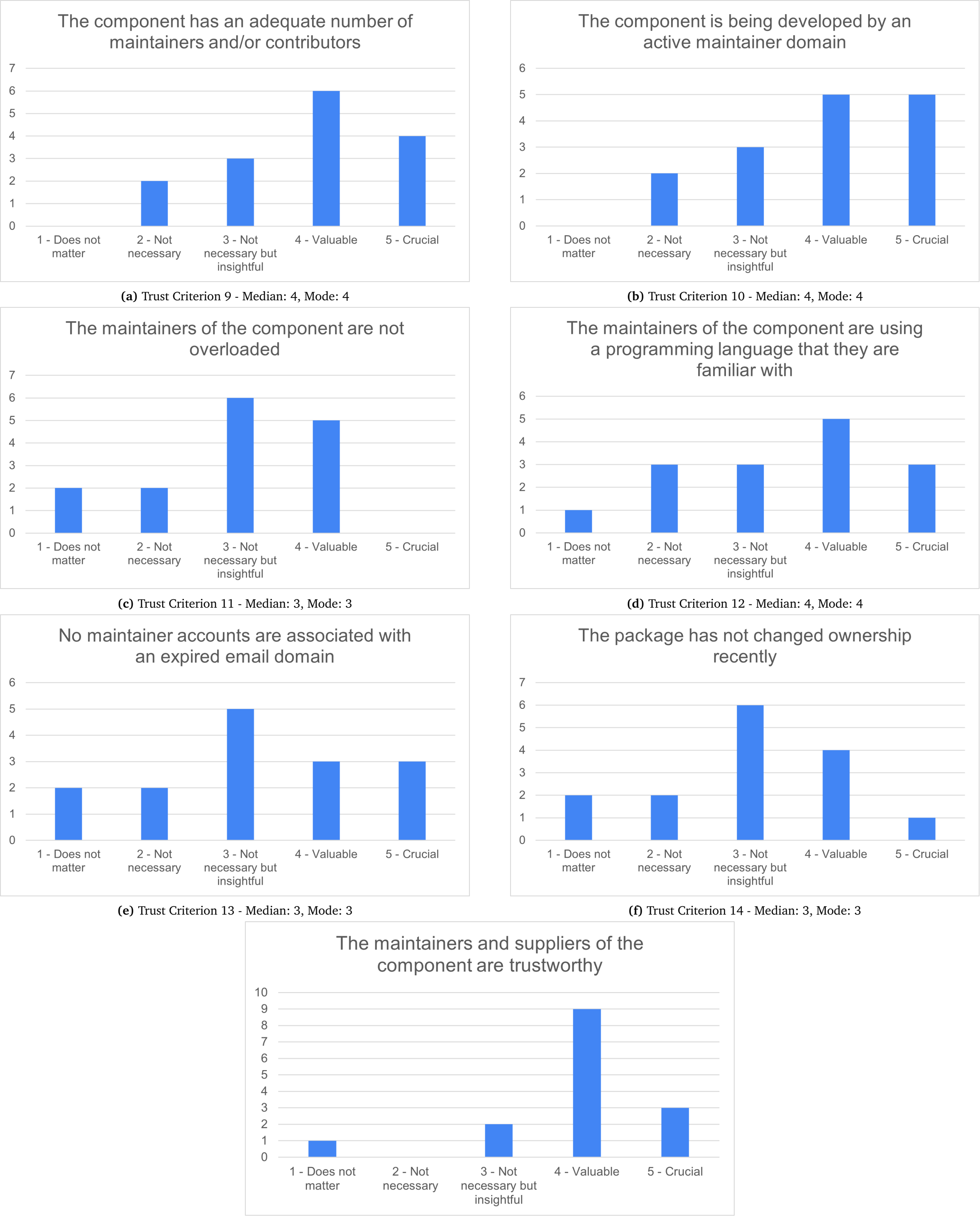
(a) Trust Criterion 5 - Median: 4, Mode: 4

(b) Trust Criterion 6 - Median: 4, Mode: 4

(c) Trust Criterion 7 - Median: 5, Mode: 5

(d) Trust Criterion 8 - Median: 4, Mode: 4

Maintainers



(a) Trust Criterion 9 - Median: 4, Mode: 4

(b) Trust Criterion 10 - Median: 4, Mode: 4

(c) Trust Criterion 11 - Median: 3, Mode: 3

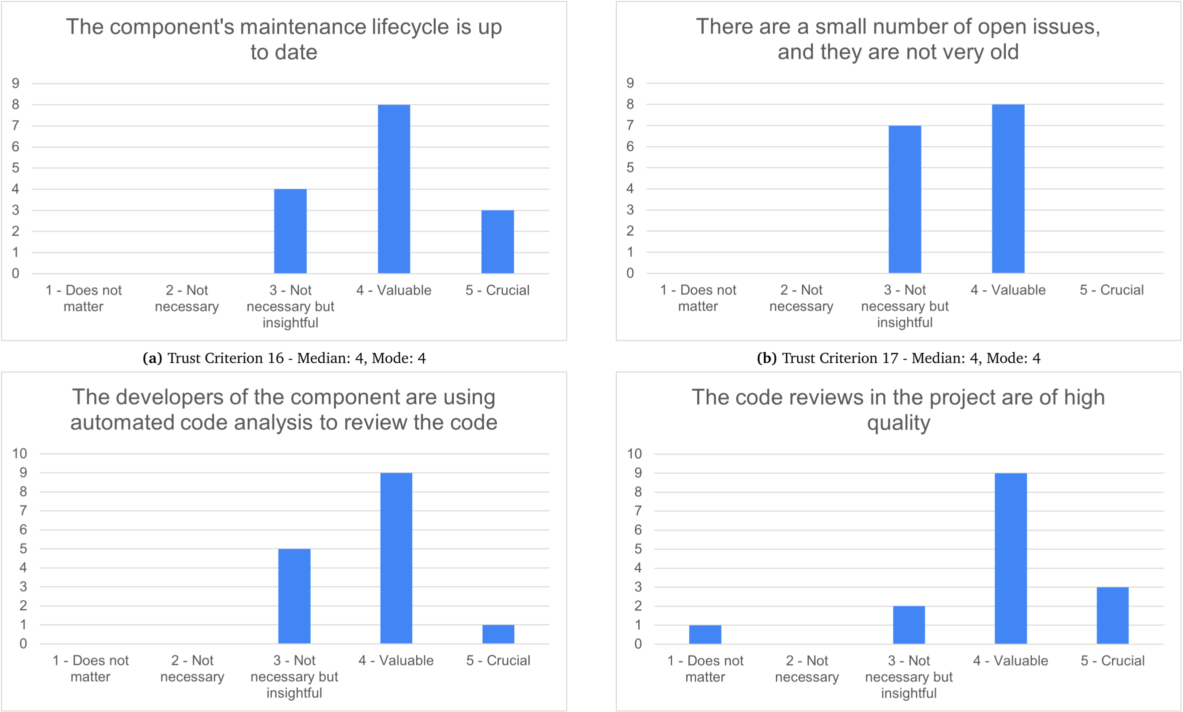
(d) Trust Criterion 12 - Median: 4, Mode: 4

(e) Trust Criterion 13 - Median: 3, Mode: 3

(f) Trust Criterion 14 - Median: 3, Mode: 3

(g) Trust Criterion 15 - Median: 4, Mode: 4

Maintenance Lifecycle



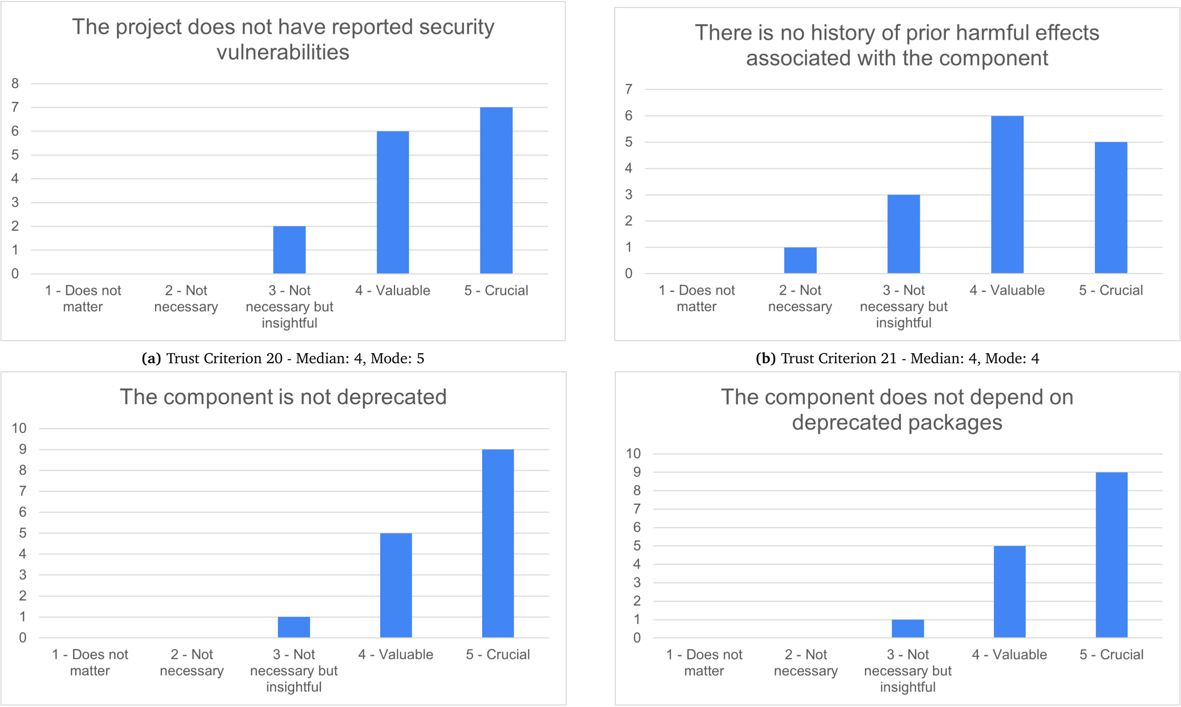
(a) Trust Criterion 16 - Median: 4, Mode: 4

(b) Trust Criterion 17 - Median: 4, Mode: 4

(c) Trust Criterion 18 - Median: 4, Mode: 4

(d) Trust Criterion 19 - Median: 4, Mode: 4

Reported Problems & Deprecation



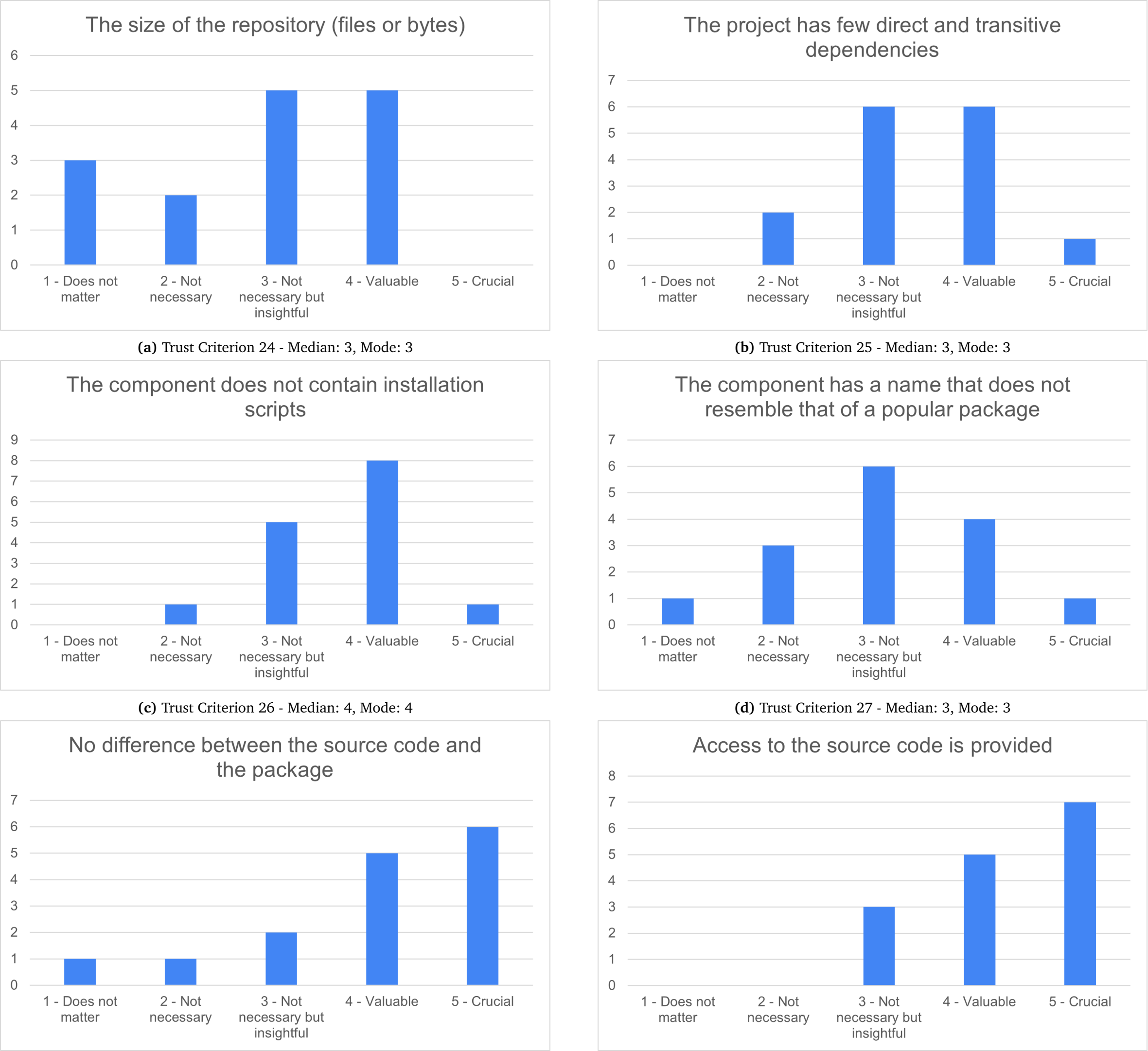
(a) Trust Criterion 20 - Median: 4, Mode: 5

(b) Trust Criterion 21 - Median: 4, Mode: 4

(c) Trust Criterion 22 - Median: 5, Mode: 5

(d) Trust Criterion 23 - Median: 5, Mode: 5

Other



(a) Trust Criterion 24 - Median: 3, Mode: 3

(b) Trust Criterion 25 - Median: 3, Mode: 3

(c) Trust Criterion 26 - Median: 4, Mode: 4

(d) Trust Criterion 27 - Median: 3, Mode: 3

(e) Trust Criterion 28 - Median: 4, Mode: 5

(f) Trust Criterion 29 - Median: 4, Mode: 5

Final Questions

Are there any trust criteria that you use or think are valuable, that were not mentioned?

Answer 1: Previous experience of the dependency use in other products, inside our product group, inside Visma. Package costs. Existing licenses from Visma. Corporate decisions to go with specific vendor solution.

Any other thoughts (about the proposed trust criteria, this questionnaire, etc.)?

Answer 1: Most of those criteria doesn't work with closed source dependencies

Answer 2: Criteria 28 "No difference between the source code and the package" is rather silly criteria to have as in most cases the content of the package is the compiled version of the source code. The few cases where it makes sense to do so are for languages where one doesn't usually do any optimization, like Python.

Answer 3: As with everything it depends, some of the answers could fluctuate by 1 depending on the dependency in question. I.E. for security vulnerabilities that are not applicable to your situation (unused features for example). In that case it wouldn't hurt to still install it if it has an otherwise proven track record and actively maintained.

Answer 4: If the package is matured, it is not developed with new functionality or has old bugs that can be handled. It does the work - it is OK. Often it is not feasible to switch to something newer. If in the package used functionality area does not have Security vulnerabilities it is OK. It can have security vulnerabilities in the functionality area that are not used

B.4 Questionnaire to Computer Science Students

This section contains the questions that were asked in the questionnaire to the computer science students. The order of the questions about the Serilog, Analytics, and PTJK.GenericRepoSpecPattern packages were changed to all the different combinations.

NuGet Package Evaluation

You are invited to participate in a research project where the main purpose is to evaluate three NuGet packages manually.

This questionnaire is part of a master's thesis made by two students at NTNU to create a tool that will use a set of criteria to evaluate third-party software dependencies automatically.

The students:
- Hallvard Molin Morstøl (hallvarm@stud.ntnu.no)
- Sverre Rynning-Tønnesen (sverrery@stud.ntnu.no)

Supervisors:
- Daniela Soares Cruzes (daniela.soares.cruzes@visma.com)

The estimated time for filling out this form is 15 minutes.

Part 1- Package evaluation

In this part we ask you to do a critical risk evaluation of NuGet packages.

Scenario:
- You are working on a software project and you have found three software packages you think could be useful in the project.
- Spend as much time evaluating the packages as you normally would when analysing packages. If you normally don't evaluate packages, then spend as much time as you think is reasonable.
- You have been asked to rank the packages associated risk on a scale from 1-5 (1-high risk and 5-low risk) and also include some information about how you reached this conclusion.
- You can use any website or tool you like to help you with doing the assessment.

What would you rank Serilog?

	1	2	3	4	5	
High risk						Low risk

Why did you give Serilog that score?

Answer: ____

What would you rank Analytics?

	1	2	3	4	5	
High risk						Low risk

Why did you give Analytics that score?

Answer: ____

What would you rank PTJK.GenericRepoSpecPattern?

	1	2	3	4	5	
High risk						Low risk

Why did you give PTJK.GenericRepoSpecPattern that score?

Answer: ____

Part 2- Trust criteria used

In this part we want to find out what criteria you used for evaluating the NuGet packages. Please do NOT go back to Part 1 and change your answers.

Please answer "Yes" for all the criteria you considered, and "Yes, but unable to evaluate" if you thought of the trust criteria but could not use the information for evaluating the risk of the package being problematic.

Trust criteria:
- Age of package: How old is the package
- Package having analyzers: Does the package file contain analyzers (analyzers can execute code automatically)
- Number of Contributors: The number of contributors to the package
- Deprecated package: If the package is deprecated
- Deprecated dependencies: If the package uses any deprecated dependencies
- Direct and transitive dependencies: The amount of direct and transitive dependencies
- Documentation: How is the documentation?
- Package has Initialization script(s): Does the package file contain initalization scripts (initalization scripts can execute code automatically)
- Known vulnerabilities: Is there any known vulnerabilities associated with the package
- License: Do the package have a license, and if yes, what license
- Number of Open issues: The amount of open issues
- Number of Open pull requests: The amount of open pull requests
- Popularity (downloads): How popular is the package
- Package has a Verified prefix: The package name has been verified by NuGet, and therefore have a verified prefix
- Widespread use (popularity over time): Have the package been popular over a period of time

Did you consider these criteria when evaluating the packages in the previous section?

	No	Yes	Yes, but unable to evaluate	Unsure
1. Age of package				
2. Package having analyzers				
3. Number of Contributors				
4. Deprecated package				
5. Deprecated dependencies				
6. Direct and transitive dependencies				
7. Documentation				
8. Package has Initialization script(s)				
9. Known vulnerabilities				
10. License				
11. Number of Open issues				
12. Number of Open pull requests				
13. Popularity (downloads)				
14. Package has a Verified prefix				
15. Widespread use (popularity over time)				

Did you consider some other criteria not listed in the previous list? If yes, please elaborate.

Answer: ____

B.5 Interview Questions

This section contains the questions that were asked during the interviews with Visma employees after they had tested the artifact.

Questions to Visma Employees

Background / Prior habits

- For how long have you used the .NET ecosystem?
- How often do you normally download packages from NuGet?
- Which operating system did you use when testing the tool?
- How did you assess the risk/security of NuGet packages before?
    - Have you used any tools in this assessment?
    - Do you know about any tools that could be used in such an assessment?

Installing / Using

- How was your experience with installing and using the tool?
- How much time would you say you used to download it, and have it run on your machine?
- What kind of packages did you test?
- Did you experience any issues when testing packages?
- Did you use the option flags to get more information?

Usefulness

- Do you find AutoTrust useful in your job?
- Does using AutoTrust make your job easier?
- How do you think using AutoTrust affects your efficiency and performance?

Ease of use

- Would you say it was easy to use AutoTrust?
- Was your interaction with AutoTrust clear and understandable?

- Was it easier to use AutoTrust over time?

Trust Criteria

- Did you look into how each trust criteria operates and what it checks?
- Do you have any feedback on the particular trust criteria and their thresholds? (https://github.com/HallvardMM/AutoTrust)

1. Age
2. Analyzers
3. Contributors
4. Deprecated Package
5. Deprecated Dependencies
6. Direct and Transitive Dependencies
7. Documentation
8. Initialization Script
9. Known Vulnerabilities
10. License
11. Open Issues
12. Open Pull Requests
13. Popularity
14. Verified Prefix
15. Widespread Use

- Were there any trust criteria you did not understand?
- Were there any trust criteria you found more useful than others?
- Were there any criteria you thought were unnecessary?
- Were there any trust criteria you felt were lacking?

    - Or anything else that could have been interesting to consider when evaluating the risk?

Rating

- Did you find the assessment credible?
- Did you feel like the overall star rating represented the result of the various trust criteria?

Overall idea

- What do you think about this way of evaluating the risk of packages?
- Is this something you would continue using? If not, why?

Advantages and disadvantages

- What did you like?
- What did you, not like?
- Is there something you would change?

Finishing Question

- What do you think should be the next steps in improving this tool?
- Anything else you want to add?

B.6 100 NuGet Packages Test

The results after testing 100 NuGet packages. We used the top 50 most downloaded packages, 40 random packages, and 10 problematic packages due to them having some issues. 0 indicates fail, 1 indicates a warning, and 2 means pass. The results can also be found here: `https://github.com/HallvardMM/Master-s-Thesis-AutoTrust/tree/main/100-package-test`

	Package Name	Version	Category	Able to fetch Github data	Overall rating	Age	Analyzers	Contributors	Deprecated Package	Deprecated Dependencies	Direct and Transitive Dependencies	Documentation	Initialization Script	Known Vulnerabilities	License	Open Issues	Open Pull Requests	Popularity	Verified Prefix	Widespread Use	Note
1	newtonsoft.json	13.0.3	Top 50	Y	5	2	2	1	2	2	2	2	2	2	2	0	2	2	2	2	
2	serilog	2.12.0	Top 50	Y	5	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
3	awssdk.core	3.7.106.23	Top 50	Y	4	0	2	2	2	2	2	2	2	2	1	2	2	2	2	2	
4	castle.core	5.1.1	Top 50	Y	5	2	2	2	2	2	2	2	2	2	2	2	0	2	2	2	
5	newtonsoft.json.bson	1.0.2	Top 50	Y	4	0	2	1	2	2	2	2	2	2	2	1	2	2	2	1	
6	swashbuckle.aspnetcore.swagger	6.5.0	Top 50	Y	5	2	2	2	2	2	2	2	2	2	2	2	2	2	1	2	
7	swashbuckle.aspnetcore.swaggergen	6.5.0	Top 50	Y	5	2	2	2	2	2	2	2	2	2	2	2	2	2	1	2	
8	polly	7.2.3	Top 50	Y	5	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
9	automapper	12.0.1	Top 50	Y	5	2	2	2	2	2	2	2	2	2	2	0	2	2	2	2	
10	swashbuckle.aspnetcore.swaggerui	6.5.0	Top 50	Y	5	2	2	2	2	2	2	2	2	2	2	2	2	2	1	2	
11	serilog.sinks.file	5.0.0	Top 50	Y	5	0	2	1	2	2	2	2	2	2	2	2	2	2	2	2	
12	moq	4.18.4	Top 50	Y	5	2	2	2	2	2	2	2	2	2	1	2	2	2	2	2	
13	swashbuckle.aspnetcore	6.5.0	Top 50	Y	5	2	2	2	2	2	2	2	2	2	2	2	2	2	1	2	
14	serilog.sinks.console	4.1.0	Top 50	Y	5	2	2	2	2	2	2	2	2	2	2	2	2	2	2	1	
15	humanizer.core	2.14.1	Top 50	Y	4	0	2	0	2	2	2	2	2	2	2	0	2	2	2	2	
16	xunit.extensibility.core	2.4.2	Top 50	Y	5	2	2	1	2	2	2	2	2	2	2	0	2	2	2	1	
17	serilog.extensions.logging	3.1.0	Top 50	Y	3	0	2	2	2	0	2	2	2	2	2	0	2	2	2	2	
18	fluentvalidation	11.5.2	Top 50	Y	4	0	2	2	2	2	2	2	2	2	2	0	0	2	2	2	
19	xunit.abstractions	2.0.3	Top 50	Y	4	0	2	1	2	2	2	2	2	2	1	0	2	2	2	1	
20	google.protobuf	3.22.3	Top 50	Y	5	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
21	xunit.extensibility.execution	2.4.2	Top 50	Y	5	2	2	1	2	2	2	2	2	2	2	2	2	2	2	1	
22	coverlet.collector	3.2.0	Top 50	Y	5	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
23	serilog.settings.configuration	3.4.0	Top 50	Y	4	2	2	2	2	0	2	2	0	2	2	2	2	2	2	2	
24	stackexchange.redis	2.6.104	Top 50	Y	5	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
25	serilog.formatting.compact	1.1.0	Top 50	Y	4	0	2	0	2	2	2	2	2	2	2	0	0	2	2	1	
26	xunit.core	2.4.2	Top 50	Y	5	2	2	1	2	2	2	2	2	2	2	0	2	2	2	1	
27	xunit.assert	2.4.2	Top 50	Y	5	2	2	1	2	2	2	2	2	2	2	0	2	2	2	1	
28	xunit.runner.visualstudio	2.4.5	Top 50	Y	5	2	2	2	2	2	2	2	2	2	2	0	2	2	2	2	
29	pipelines.sockets.unofficial	2.2.2	Top 50	Y	4	0	2	0	2	2	2	2	2	2	2	0	2	2	1	2	
30	xunit	2.4.2	Top 50	Y	4	2	0	1	2	2	2	2	0	2	2	0	2	2	2	2	
31	grpc.core.api	2.52.0	Top 50	Y	5	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
32	xunit.analyzers	1.1.0	Top 50	Y	3	2	0	1	2	2	2	2	0	2	2	0	0	2	2	0	
33	serilog.extensions.hosting	5.0.1	Top 50	Y	4	2	2	2	0	2	2	2	2	2	2	2	0	2	2	1	
34	npgsql	7.0.2	Top 50	Y	5	2	0	2	2	2	2	2	2	2	1	2	2	2	2	2	
35	portable.bouncycastle	1.9.0	Top 50	Y	4	0	2	2	2	2	2	2	2	2	1	1	2	2	2	2	
36	serilog.aspnetcore	6.1.0	Top 50	Y	4	2	2	1	2	0	2	2	2	2	2	2	2	2	2	2	
37	fluentassertions	6.11.0	Top 50	Y	5	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	
38	serilog.sinks.debug	2.0.0	Top 50	Y	4	0	2	0	2	2	2	2	2	2	2	0	0	2	2	1	
39	identitymodel	6.0.0	Top 50	Y	4	0	0	2	2	2	2	0	2	2	2	2	2	2	2	2	
40	dapper	2.0.123	Top 50	Y	4	0	2	0	2	2	2	2	2	2	2	0	2	2	1	2	
41	restsharp	110.2.0	Top 50	Y	5	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	
42	nunit	3.13.3	Top 50	Y	4	0	2	2	2	2	2	2	2	2	1	0	2	2	2	2	
43	jetbrains.annotations	2022.3.1	Top 50	Y	5	2	2	1	2	2	2	2	2	2	2	1	0	2	2	2	
44	nlog	5.1.3	Top 50	Y	5	2	2	2	2	2	1	2	2	2	2	2	2	2	2	2	
45	automapper.extensions.microsoft.dependencyinjection	12.0.1	Top 50	Y	4	0	2	1	2	2	2	2	2	2	2	0	0	2	2	2	
46	grpc.net.common	2.52.0	Top 50	Y	5	2	2	2	2	2	2	2	2	2	2	2	2	1	2	2	
47	autofac	7.0.1	Top 50	Y	4	0	2	2	2	2	2	2	2	2	2	2	0	2	1	2	
48	serilog.sinks.periodicbatching	3.1.0	Top 50	Y	5	2	2	1	2	2	2	2	2	2	2	1	0	2	2	1	
49	awssdk.s3	3.7.104.1	Top 50	Y	3	0	0	2	2	2	2	2	0	2	1	2	2	2	2	0	
50	mongodb.bson	2.19.1	Top 50	Y	4	2	2	2	2	2	2	2	2	2	1	0	0	2	2	2	
	Median				**5**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	
	Mode				**5**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	
51	Yoko.Tool	2.5.3	Random	N	3	2	2	0	2	2	2	2	2	2	0	0	0	0	1	0	
52	bar	1.0.0	Random	N	2	0	2	0	2	2	2	1	2	2	1	0	0	0	1	0	
53	Ardalis.GuardClauses	4.0.1	Random	Y	3	0	2	2	2	2	2	2	2	2	0	2	2	0	1	2	
54	Kolabs.Repository.Dapper	1.2.0	Random	N	2	0	2	0	2	2	2	0	2	2	0	0	0	0	1	0	
55	Feign	1.4.1	Random	Y	3	2	2	2	2	2	2	2	2	2	1	0	0	1	1	0	
56	MvmNet.ActiveDirectory	2.2.3	Random	N	3	2	2	0	2	2	2	0	2	2	2	0	0	0	1	0	
57	Mcs	5.8.0	Random	Y	3	0	2	2	2	2	2	2	2	2	1	0	2	0	2	0	
58	xEdi.EdiEngine	1.6.0	Random	Y	3	0	2	0	2	2	2	2	2	2	2	1	2	1	1	0	
59	HiNetCloud.Microsoft.Samples.Debugging.MdbgEngine	23.4.20.10	Random	N	3	0	2	0	2	2	2	1	2	2	2	0	0	0	1	0	
60	OdeToCode.AddFeatureFolders	2.0.3	Random	Y	3	0	2	0	2	0	2	2	2	2	1	1	2	1	1	0	
61	MemBus	4.0.1	Random	Y	2	0	2	0	2	2	2	0	2	2	1	0	0	0	1	0	
62	IronSourceHyprMXSDK	5.1.2	Random	N	2	0	2	1	2	2	2	1	2	2	1	0	0	0	1	0	
63	SqExpress	0.4.1	Random	Y	3	2	2	1	2	2	2	2	0	2	2	0	0	0	1	0	
64	HdrHistogram	2.5.0	Random	Y	3	0	2	0	2	2	2	2	2	2	1	0	0	1	1	1	
65	VaultSharp	1.13.0.1	Random	Y	3	0	0	2	2	2	2	2	2	2	2	2	0	1	1	2	
66	Hco.Base.DataAccess.Ef	0.6.5.5	Random	N	2	0	2	0	2	0	2	0	2	2	0	0	0	0	1	0	
67	Yebo.Net.Utility	1.0.0	Random	N	2	2	2	0	0	2	2	0	2	2	0	0	0	0	1	0	
68	Model2Form	1.0.0	Random	N	2	0	2	0	2	2	2	1	2	2	0	0	0	0	1	0	
69	MR.Gestures	3.0.0	Random	N	3	2	2	0	2	2	2	1	2	2	0	0	0	0	1	2	
70	IrwanUnitTestFramework	1.0.0	Random	N	2	0	2	0	2	2	2	0	2	2	0	0	0	0	1	0	
71	Mtn.Library	4.0.0.83	Random	N	2	0	2	0	2	2	2	1	0	2	1	0	0	0	1	0	
72	StockSharp.IEX	5.0.139	Random	Y	3	0	2	2	2	2	2	2	2	2	1	0	2	1	2	0	
73	HMEFramework	2.1.33.38	Random	N	2	0	2	0	2	2	2	0	0	2	0	0	0	0	1	0	
74	MailBee.NET	12.3.1	Random	N	3	2	2	0	2	2	1	1	2	2	1	0	0	0	1	2	
75	aqw001.QAppLogger	1.0.1	Random	N	2	0	2	0	2	2	2	0	2	2	0	0	0	0	1	0	
76	RegexPatterns	1.1.0	Random	Y	2	0	2	1	2	2	2	1	2	2	0	0	0	0	1	0	
77	jieba.NET	0.42.2	Random	Y	3	0	2	0	2	2	2	2	2	2	1	0	0	1	1	0	
78	Aeolin.AwoBot	1.5.3	Random	N	2	2	0	0	2	0	2	0	2	2	0	0	0	0	1	0	
79	Abbyy.CloudSdk.V2.Client	1.0.6	Random	Y	2	0	2	0	2	0	2	2	0	2	1	0	0	1	1	1	
80	PTJK.GenericRepoSpecPattern	1.0.1	Random	N	1	0	0	0	2	0	2	0	2	2	0	0	0	0	1	0	
81	NSKeyedUnarchiver	1.1.3	Random	Y	2	0	2	0	2	2	2	0	2	2	1	0	0	0	1	0	
82	NLanguageTag	3.2.0	Random	Y	3	0	2	0	2	2	2	2	2	2	2	0	0	1	1	0	
83	Bve5_Parsing	1.0.1	Random	Y	2	0	2	0	2	0	2	2	2	2	2	0	2	0	1	0	
84	ietws	0.2.12	Random	N	2	2	2	0	2	2	2	0	0	2	0	0	0	0	1	0	
85	Universal.BureauOfMeteorology	1.1.0	Random	N	2	0	2	0	2	2	2	0	2	2	1	0	0	0	1	0	
86	fmdev.xlftool	0.1.5	Random	Y	3	0	2	0	2	2	2	2	2	2	1	1	1	1	1	0	
87	RJGF.EasyAbp.Abp.Trees.EntityFrameworkCore	1.0.3	Random	N	2	0	2	0	2	2	2	0	2	2	0	0	0	0	1	0	
88	NPOI.HWPF	2.3.0	Random	Y	3	0	2	0	2	2	2	2	2	2	1	0	0	1	1	0	
89	NHibernateProfiler.Appender	6.0.6040	Random	N	2	2	0	0	2	0	1	1	0	2	1	0	0	0	2	2	
90	Auios.QuadTree	1.1.1	Random	Y	3	0	2	0	2	2	2	2	2	2	1	0	0	0	1	0	
	Median				**2**	**0**	**2**	**0**	**2**	**2**	**2**	**1**	**2**	**2**	**1**	**0**	**0**	**0**	**1**	**0**	
	Mode				**2**	**0**	**2**	**0**	**2**	**2**	**2**	**2**	**2**	**2**	**1**	**0**	**0**	**0**	**1**	**0**	
91	AsyncBridge.Net35	0.3.1	Issue	Y	2	0	2	0	0	2	2	0	2	2	1	0	0	0	1	0	Deprecated
92	Microsoft.ChakraCore	1.11.24	Issue	Y	3	0	2	0	2	2	2	0	2	0	1	0	0	0	2	0	Known Vulnerability
93	Microsoft.EntityFrameworkCore.Tools	7.0.5	Issue	Y	4	0	2	2	2	2	2	2	0	2	2	2	2	2	2	2	Has init.ps1 and install.ps1
94	EntityFramework.MappingAPI	6.2.1	Issue	N	2	0	2	0	0	2	2	1	0	2	1	0	0	0	1	0	Deprecated
95	MPlayerControl	1.6.0	Issue	Y	3	0	2	2	2	2	2	2	2	2	0	1	2	1	1	0	High-risk license
96	jquery.cookie	1.4.1	Issue	N	2	0	2	0	2	2	2	1	2	0	1	0	0	0	1	1	Known Vulnerability
97	Masuit.Tools.Core	2.6.0.2	Issue	Y	2	0	0	2	2	0	2	2	2	2	1	0	0	1	1	0	Had a known vulnerability
98	Analytics	3.8.1	Issue	Y	3	2	2	2	2	0	1	2	2	2	2	0	0	1	1	0	Deprecated dependency
99	elFinder.NetCore	1.3.6	Issue	Y	3	0	2	0	2	2	2	2	2	2	0	0	0	1	1	0	Missing license
100	SSCMS	7.2.0	Issue	Y	3	2	2	1	2	0	2	2	2	2	0	0	2	1	1	0	High-risk license
	Median				**3**	**0**	**2**	**0.5**	**2**	**2**	**2**	**2**	**2**	**2**	**1**	**0**	**0**	**1**	**1**	**0**	
	Mode				**3**	**0**	**2**	**0**	**2**	**2**	**2**	**2**	**2**	**2**	**1**	**0**	**0**	**1**	**1**	**0**	
	Total Median				**3**	**0**	**2**	**1**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**0**	**0**	**1.5**	**1**	**1**	
	Total Mode				**5**	**0**	**2**	**0**	**2**	**2**	**2**	**2**	**2**	**2**	**2**	**0**	**0**	**2**	**1**	**2**	

B.7 Time Test

The results after testing 10 Nuget packages and comparing the time with running the "dotnet add package" command on Windows, Mac, and Linux. The results can also be found here: `https://github.com/HallvardMM/Master-s-Thesis-AutoTrust/tree/main/Time-test`
Specifications:

- **Windows**
  - **Operating system:** Windows 11 Pro 22H2 22621.1555
  - **System:** LENOVO_MT_20NX_BU_Think_FM_ThinkPad T490s
  - **Processor:** Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz
  - **RAM:** 16 GB
  - **.NET version:** 7.0.3
- **MacOS**
  - **Operating system:** MacOS Ventura, Version 13.1
  - **System:** MacBook Pro, 13-inch, 2020, Four Thunderbolt 3 ports
  - **Processor:** 2 GHz Quad-Core Intel Core i5
  - **RAM:** 16 GB
  - **.NET version:** 7.0.200
- **Linux**
  - **Operating system:** Ubuntu 20.04.6 LTS with kernel 5.15.90.1-microsoft-standard-WSL2
  - **System:** LENOVO_MT_20NX_BU_Think_FM_ThinkPad T490s
  - **Processor:** Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz
  - **RAM:** 16 GB
  - **.NET version:** 7.0.203

Package Name	Version	Autotrust time (Windows)	.NET time (Windows)	Ratio Windows	Autotrust time (Mac)	.NET time (Mac)	Ratio Mac	Autotrust time (Linux/WSL2)	.NET time (Linux/WSL2)	Ratio Linux/WSL2
newtonsoft.json	13.0.3	5.13	1.964	2.6104888	5.656	2.462	2.297	4.626	2.37	1.952
serilog	2.12.0	5.59	2.032	2.74852362	5.073	4.021	1.262	4.551	3.722	1.223
awssdk.core	3.7.106.22	5.61	2.181	2.5740486	6.29	5.154	1.22	5.089	4.665	1.091
castle.core	5.1.1	4.75	2.088	2.276341	5.784	3.502	1.652	5.157	2.456	2.1
newtonsoft.json.bson	1.0.2	5.72	1.969	2.90248857	6.216	2.834	2.193	4.838	2.384	2.029
swashbuckle.aspnetcore.swagger	6.5.0	5.48	2.042	2.68315377	5.133	2.911	1.763	4.937	2.447	2.018
swashbuckle.aspnetcore.swaggergen	6.5.0	5.91	2.084	2.83637236	5.854	3.502	1.672	5.44	2.422	2.246
polly	7.2.3	5.41	2.088	2.59099617	5.891	3.134	1.88	4.206	2.377	1.769
automapper	12.0.1	5.31	2.044	2.59637965	5.459	3.492	1.563	5.332	3.13	1.704
swashbuckle.aspnetcore.swaggerui	6.5.0	5.44	2.313	2.34976221	5.251	2.833	1.854	4.723	2.724	1.734
Average		**5.4336**	**2.0805**	**2.616855475**	**5.6607**	**3.3845**	**1.7356**	**4.8899**	**2.8697**	**1.7866**

Note: The time is in seconds

B.8 100 Packages Test OpenSSF Scorecard

The results after testing 100 NuGet packages with the OpenSSF Scorecard tool. We believe that the package marked with red is most likely not the correct source code even though it is linked on the Nuget webpage for the project. The results can also be found here: `https://github.com/HallvardMM/Master-s-Thesis-AutoTrust/tree/main/100-package-test`

Package Name	Category	Able to fetch Github data	Aggregate score (/10)
newtonsoft.json	Top 50	Y	3.5
serilog	Top 50	Y	4.8
awssdk.core	Top 50	Y	6.9
castle.core	Top 50	Y	4.9
newtonsoft.json.bson	Top 50	Y	3.1
swashbuckle.aspnetcore.swagger	Top 50	Y	3.4
swashbuckle.aspnetcore.swaggergen	Top 50	Y	3.4
polly	Top 50	Y	7.9
automapper	Top 50	Y	6.3
swashbuckle.aspnetcore.swaggerui	Top 50	Y	3.4
serilog.sinks.file	Top 50	Y	4.6
moq	Top 50	Y	5.2
swashbuckle.aspnetcore	Top 50	Y	3.4
serilog.sinks.console	Top 50	Y	3.4
humanizer.core	Top 50	Y	4.8
xunit.extensibility.core	Top 50	Y	5.3
serilog.extensions.logging	Top 50	Y	4.2
fluentvalidation	Top 50	Y	4.9
xunit.abstractions	Top 50	Y	5.3
google.protobuf	Top 50	Y	6
xunit.extensibility.execution	Top 50	Y	5.3
coverlet.collector	Top 50	Y	4.6
serilog.settings.configuration	Top 50	Y	4.2
stackexchange.redis	Top 50	Y	4.4
serilog.formatting.compact	Top 50	Y	3.4
xunit.core	Top 50	Y	5.3
xunit.assert	Top 50	Y	5.3
xunit.runner.visualstudio	Top 50	Y	5.5
pipelines.sockets.unofficial	Top 50	Y	4
xunit	Top 50	Y	5.3
grpc.core.api	Top 50	Y	5.3
xunit.analyzers	Top 50	Y	4.3
serilog.extensions.hosting	Top 50	Y	4.2
npgsql	Top 50	Y	5.6
portable.bouncycastle	Top 50	Y	4.8
serilog.aspnetcore	Top 50	Y	4.8
fluentassertions	Top 50	Y	5.4
serilog.sinks.debug	Top 50	Y	3.1
identitymodel	Top 50	Y	6.3
dapper	Top 50	Y	4.3
restsharp	Top 50	Y	6.8
nunit	Top 50	Y	5.2
jetbrains.annotations	Top 50	Y	3.9
nlog	Top 50	Y	6.6
automapper.extensions.microsoft.dependencyinjection	Top 50	Y	3.8
grpc.net.common	Top 50	Y	5.3
autofac	Top 50	Y	4.8
serilog.sinks.periodicbatching	Top 50	Y	3.4
awssdk.s3	Top 50	Y	6.9
mongodb.bson	Top 50	Y	5.2
Median			**4.8**
Mode			**5.3**
Yoko.Tool	Random	N	
bar	Random	N	
Ardalis.GuardClauses	Random	Y	6.1
Kolabs.Repository.Dapper	Random	N	
Feign	Random	Y	4.4
MvmNet.ActiveDirectory	Random	N	
Mcs	Random	Y	6.3
xEdi.EdiEngine	Random	Y	3.3
HiNetCloud.Microsoft.Samples.Debugging.MdbgEngine	Random	N	
OdeToCode.AddFeatureFolders	Random	Y	3.1
MemBus	Random	Y	2.8
IronSourceHyprMXSDK	Random	N	
SqExpress	Random	Y	3.1
HdrHistogram	Random	Y	2.9
VaultSharp	Random	Y	4.8
Hco.Base.DataAccess.Ef	Random	N	
Yebo.Net.Utility	Random	N	
Model2Form	Random	N	
MR.Gestures	Random	N	
IrwanUnitTestFramework	Random	N	
Mtn.Library	Random	N	
StockSharp.IEX	Random	Y	4.3
HMEFramework	Random	N	
MailBee.NET	Random	N	
aqw001.QAppLogger	Random	N	
RegexPatterns	Random	Y	2.8
jieba.NET	Random	Y	3
Aeolin.AwoBot	Random	N	
Abbyy.CloudSdk.V2.Client	Random	Y	3
PTJK.GenericRepoSpecPattern	Random	N	
NSKeyedUnarchiver	Random	Y	3.1
NLanguageTag	Random	Y	3.1
Bve5_Parsing	Random	Y	3.7
ietws	Random	N	
Universal.BureauOfMeteorology	Random	N	
fmdev.xlftool	Random	Y	3.6
RJGF.EasyAbp.Abp.Trees.EntityFrameworkCore	Random	N	
NPOI.HWPF	Random	Y	2.8
NHibernateProfiler.Appender	Random	N	
Auios.QuadTree	Random	Y	3
Median			**3.1**
Mode			**3.1**
AsyncBridge.Net35	Issue	N	
Microsoft.ChakraCore	Issue	Y	4.7
Microsoft.EntityFrameworkCore.Tools	Issue	Y	7.5
EntityFramework.MappingAPI	Issue	N	
MPlayerControl	Issue	Y	5.4
jquery.cookie	Issue	N	
Masuit.Tools.Core	Issue	Y	5.1
Analytics	Issue	Y	4.9
elFinder.NetCore	Issue	Y	2.8
SSCMS	Issue	Y	4.1
Median			**4.9**
Mode			**4.9**
Total Median			**4.6**
Total Mode			**5.3**

B.9 One-pager

This short text was published internally at Visma to spark some interest in the topic and to get feedback from some of the developers. The text was based on the news article from JFrog about malicious NuGet packages [6].

Nicoleta Botosan 3 minutes ago

VISMA SECURITY PROGRAM - RESEARCH AND DEVELOPMENT

8 min read

Caught in the NET 🕸️: The Rising Threat of Malicious NuGet Packages Targeting .NET Developers

👨‍🎓 Hallvard Molin Morstøl and Sverre Rynning-Tønnesen - two Master's students from the Norwegian University of Science and Technology (NTNU) want to present you some things about a recent attack targeting .NET developers through NuGet. Want to know how this affects you and maybe also how to prevent this attack? Then I recommend you check out this post which is entirely written by them! 👇

One week ago, the JFrog 🐸 Security Research team released a blog post describing a recent attack targeting .NET developers through NuGet. The attack was performed by spreading malicious NuGet packages that were downloaded more than 150K times. This specific attack used typosquatting techniques, but this is just one out of multiple possible attack vectors. Quite interesting! So let's dive in! 🤔

🏞️ What does the current threat landscape look like?

Package registries like npm, PyPI, Maven Central, and NuGet can all be used for spreading malicious packages, and one should not blindly trust and download packages. The number of attackers using package registries to spread malicious packages is on the rise and while countermeasures are being worked on this problem will probably be around for a long time.

These kinds of attacks are also closely related to another form of supply chain attack where attackers are targeting weaknesses in third-party code. Here attackers can target dependencies (packages) that are used by other dependencies. Infiltrating one package, that is used by another package, which is then used by you, would compromise your program as well as all other programs having the vulnerable package in their dependency tree. As you see this means that one malicious package could make a lot of harm. To see a recent example of a big supply chain attack, just search for the 2020 SolarWinds☀️ hack.

🕶️ Which packages should we look out for?

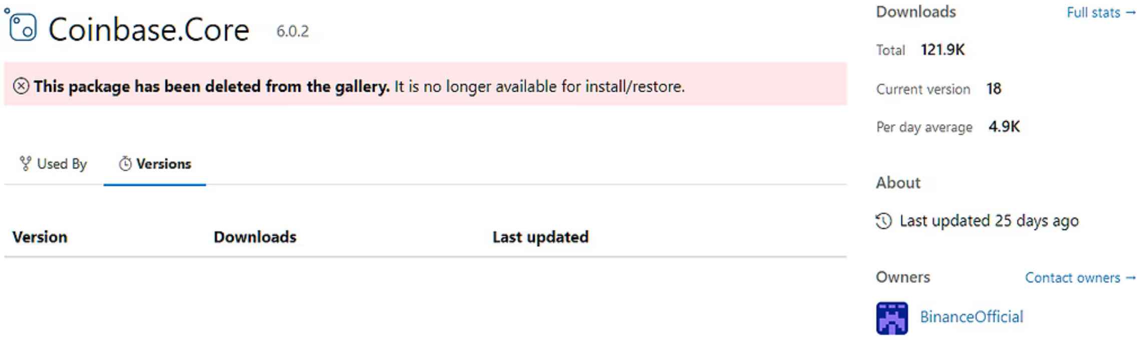
One large security vulnerability with .NET packages is that it facilitates executing code immediately upon package installation using “init.ps1”-files. Visual Studio will automatically run these “init” files without giving any warning to the user. This means that we need to make sure the malicious packages are never downloaded in the first place.

There are all kinds of tricks 🪄 used to make malicious packages look legit. JFrog lists some indicators for malicious indicators in their post, but these are just a few of the many possibilities.

Some common indicators are:

💣 Typosquatting

Typosquatting is a technique where you use a package name that is similar to a legitimate, and often popular, package. One example of such a package is "Coinbase.Core" which mimics the package "Coinbase".



Coinbase.Core 6.0.2

This package has been deleted from the gallery. It is no longer available for install/restore.

Used By | Versions

Version	Downloads	Last updated

Downloads — Full stats →

Total 121.9K

Current version 18

Per day average 4.9K

About

Last updated 25 days ago

Owners — Contact owners →

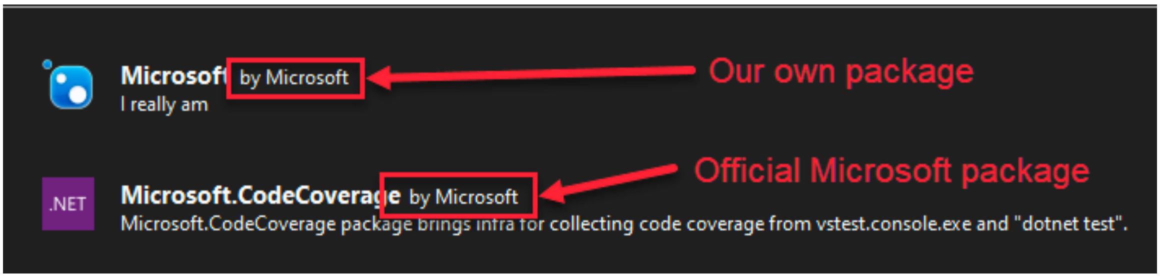
BinanceOfficial

💣 Using package owner names that increase trust or are similar to known users

Another approach used is to use profile names that make the package seem more legitimate. For example "NuGetDev" or "OfficialDevelopmentTeam". One can also use names that are similar to the names of known users. Joel Verhagen is a Microsoft developer working on NuGet with the username "joelverhagen". A fake profile that tries to pretend to be Verhagen is "joeIverhagen", where the lowercase "L" is swapped with an uppercase "i".

💣 Inserting misleading metadata

The package metadata in the ".nuspec" contains various information about the package. Here the author field does not go through a verification process and the attacker could make it look like a package was released by "Microsoft" or another trusted source. This could be followed up by a well-written description (which is not the case in this example🤡) to make it look trustworthy.



Microsoft by Microsoft
I really am

Our own package

Microsoft.CodeCoverage by Microsoft
Microsoft.CodeCoverage package brings infra for collecting code coverage from vstest.console.exe and "dotnet test".

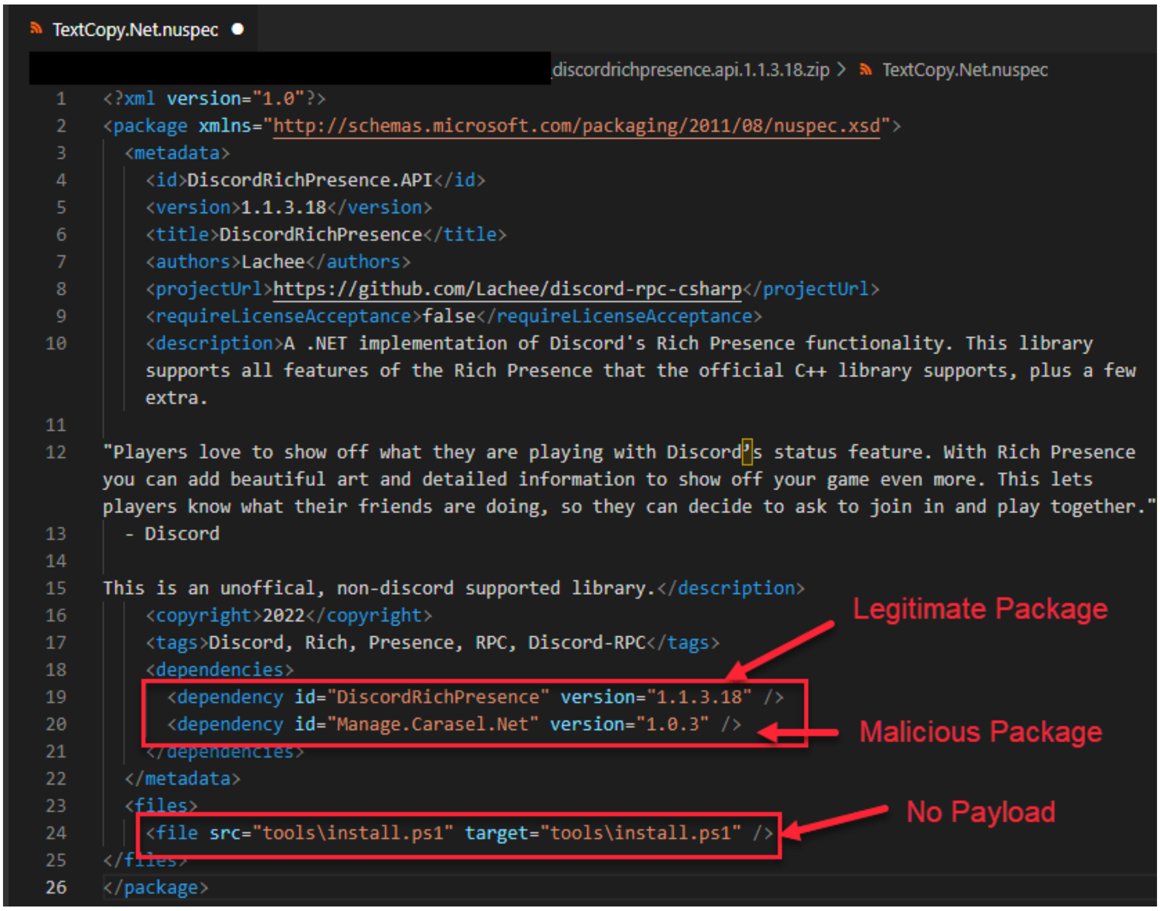
Official Microsoft package

💣 Bumping the total number of downloads

As already mentioned the malicious "Coinbase.Core" package found by JFrog was downloaded 150K times. This is more than the actual "Coinbase" package which is downloaded 118K times. The attackers have most likely artificially bumped the download count using bots to boost the package’s trustworthiness.

💣 Using malicious dependencies

Another thing to look out for are the dependencies included. As mentioned, the packages can be used in a supply chain attack where the code in the package you download is fine but the dependency used can be malicious.


TextCopy.Net.nuspec
[redacted] discordrichpresence.api.1.1.3.18.zip > TextCopy.Net.nuspec

```
<?xml version="1.0"?>
<package xmlns="http://schemas.microsoft.com/packaging/2011/08/nuspec.xsd">
  <metadata>
    <id>DiscordRichPresence.API</id>
    <version>1.1.3.18</version>
    <title>DiscordRichPresence</title>
    <authors>Lachee</authors>
    <projectUrl>https://github.com/Lachee/discord-rpc-csharp</projectUrl>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>A .NET implementation of Discord's Rich Presence functionality. This library supports all features of the Rich Presence that the official C++ library supports, plus a few extra.

"Players love to show off what they are playing with Discord's status feature. With Rich Presence you can add beautiful art and detailed information to show off your game even more. This lets players know what their friends are doing, so they can decide to ask to join in and play together."
  - Discord

This is an unoffical, non-discord supported library.</description>
    <copyright>2022</copyright>
    <tags>Discord, Rich, Presence, RPC, Discord-RPC</tags>
    <dependencies>
      <dependency id="DiscordRichPresence" version="1.1.3.18" />
      <dependency id="Manage.Carasel.Net" version="1.0.3" />
    </dependencies>
  </metadata>
  <files>
    <file src="tools\install.ps1" target="tools\install.ps1" />
</files>
</package>
```

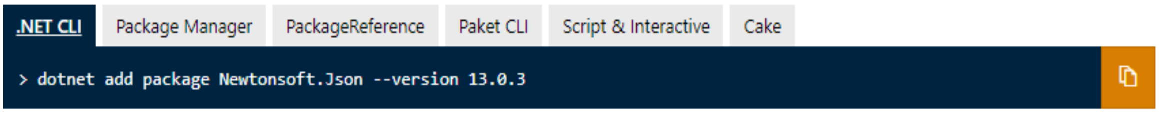
Legitimate Package

Malicious Package

No Payload

Sooo, how should we protect ourselves?

All developers should do an assessment of packages before adding them to their projects. If it is possible, it is good to have a second pair of eyes to help during the evaluation of the package. Since many programming languages allow the packages to execute code on installation, such as NuGet and npm, the developer should make sure they trust the code before downloading it. To prevent being a target of a typosquatting attack you should copy the download command from the official package registry site.



```
> dotnet add package Newtonsoft.Json --version 13.0.3
```

When adding a dependency you should make it clear in the pull request why you have added the dependency. It can also be valuable to have guidelines in the group regarding what information should be included in a pull request when adding new dependencies. After dependencies are added to a project you should use some tool to scan the project regularly for vulnerabilities using dependency scanners such as Github Dependabot, Snyk, etc.

If you want to take some extra steps for increasing security, it is possible to scan the packages before adding them using OSS-gadgets. OSS-Gadgets is a set of Microsoft-developed tools that can help find obfuscated strings, identify potential backdoors, calculate a metric for the risk of using a package, etc. They claim it is still in public preview and that it is not ready for production use, so one should still be critical when adding packages. After the packages are assessed you can choose to self-host the package with a set version in a private package registry. Then you can have a policy to only use packages from the private package registry. When updating packages to newer versions it is then important to rescan the package before adding it to the private package registry. But note that these tools are no guarantee for making sure packages are safe to use.

💪 Luckily we're working on fixing it! But we need your help!!!

We are currently writing a master's thesis about software supply chain security, where we focus on NuGet. We plan to create a tool, to wrap the "dotnet add package"-command which first fetches relevant information about the package and then automatically assesses the package using a set of trust criteria and thresholds. The information will be displayed to the developers and they are prompted if they would still like to add the package.

We are currently working on figuring out which trust criteria are valuable and the thresholds that make sense to use. Finding good trust criteria that differentiate benign and malicious packages using the available data from NuGet and other sites is hard. But by combining different trust criteria it will be harder for attackers to make the malicious packages seem trustworthy.

If you would like to help with assessing trust criteria, we would appreciate it if you answered this survey regarding trust criteria https://forms.gle/WnTEJxSqze2gReot7.

Soon we will also reach out to some of you to run user tests on this tool we are really appreciating all your help! So thank you!

About us:

The students:
- Hallvard Molin Morstøl (hallvarm@stud.ntnu.no) LinkedIn profile

- Sverre Rynning-Tønnesen (sverrery@stud.ntnu.no) LinkedIn profile

Supervisors:
- Daniela Soares Cruzes (daniela.soares.cruzes@visma.com)
- Monica Iovan (monica.iovan@visma.com)

We really hope that you found this interesting. Curious to find more about what they are planning to do? Then don't hesitate to invite @Daniela Cruzes and/or @Monica Iovan to a virtual ☕ - and they will give you more details if you want to onboard and give a helping hand to this promising master thesis!

🌻 Happy Spring!

#nuget #security #securityawareness #securityresearch

Appendix C

Project Thesis

The project thesis was done as preparatory research during the last semester. It was a SLR about software supply chain security.

Software Supply Chain Security

A Systematic Literature Review

Hallvard Molin Morstøl
Sverre Rynning-Tønnesen

Autumn 2022

Supervisor: Daniela Soares Cruzes

NTNU
Norwegian University of
Science and Technology

Contents

Figures

Tables

Acronyms

2FA Two-factor authentication. 31, 51

BSIMM Building Security In Maturity Model. 45, 48, 51

CAPEC Common Attack Pattern Enumeration and Classification. 6, 17, 27, 45, 47

CC Common Criteria. 14, 34, 48, 49

CD Continuous Deployment. 39, 40, 42, 49, 55

CI Continuous Integration. 39–42, 49, 55

COTS Commercial off-the-shelf. 33, 34, 53

CSPN La Certification de Sécurité de Premier Niveau. 14, 49, 58

CVE Common Vulnerabilities and Exposures. 6, 17, 27, 45, 47

CVSS Common Vulnerability Scoring System. 4

CWE Common Weakness Enumeration. 6, 17, 27, 44, 45, 47, 50

FOSS Free and Open-Source Software. 34

GDPR General Data Protection Regulation. 4

GOTS Government off-the-shelf. 33

IaaS Infrastructure as a service. 6

IDS Intrusion Detection System. 20, 30

IEC International Electrotechnical Commission. 47–49

IEEE Institute of Electrical and Electronics Engineers. 47, 48

IoT Internet of Things. 7, 21

IPS Intrusion Prevention System. 30

ISO International Organization for Standardization. 6, 27, 45, 47–49

NIST National Institute of Standards and Technology. 6, 27, 45–47, 52

OWASP Open Web Application Security Project. 6, 27, 45, 48

RQ Research Question. i, 8–10, 12, 14, 15, 21, 24, 25, 32, 35, 38, 42, 43, 45, 52, 53, 55

SBOM Software bill of materials. 21, 27, 32, 39, 51, 53, 57

SLR Systematic Literature Review. 2, 3, 7, 8, 15, 16, 42, 43, 56, 58–60

Chapter 1

Introduction

The modern world is getting more dependent on technology each year. Modern technology creates many opportunities and helps us solve multiple challenges. Technology and software have become an integrated part of our daily lives and are used in almost every job sector.

To adapt to the increased dependency and need for technology and software, there has been a shift from the waterfall methodology to the agile methodology. Agile development practices are based on continuous and iterative changes to the product, with frequent releases of new products to satisfy the customers [1]. In this strive to deliver new and improved software at a great pace, it has become common practice to use third-party solutions. The software field has become more modular, where companies, groups, and institutions focus on their products and allow other companies to use them. Almost all software used today have third-party dependencies to help speed up the development process.

Using third-party dependencies is generally considered favorable, but it introduces some security risks. Potential security vulnerabilities in the dependencies might affect all components using the dependency. In a large software project, there might be many dependencies, and it is very demanding to ensure that the dependencies are safe. These vulnerabilities introduced intentionally or unintentionally from third parties can lead to supply chain attacks. Cyber or software supply chain attacks are cyber-attacks that try to damage an organization by targeting less secure software dependencies.

Software supply chain security is focused on different countermeasures and techniques used to prevent and minimize consequences from software supply chain attacks. Since the software supply chain is large and complex there are multiple attack vectors that need to be defended. Code dependencies, container images, third-party services, network devices, and people are all examples of software supply chain parts that can be used in an attack.

Scientific literature about software supply chain attacks is mostly focused on open-source software. Open-source software is computer software that is released with code that is accessible. Depending on the licensing, users can use, modify, examine, and distribute the code. Closed-source software (also known as proprietary software or non-free software) is software where the source code is not openly available. When the code is not available it is more challenging to evaluate if it can be trusted. Even though more scientific literature has chosen to focus on open-source software, it does not mean that closed-source software cannot introduce security vulnerabilities to a project. There is a need to further investigate how software supply chains can be protected from security risks introduced by closed-source software as well as open-source software.

There are countermeasures developers can use to secure their software supply chain. Some examples of techniques and tools introduced to handle the risk introduced by third-party software are: tools to scan open-source dependencies [2–4], frameworks to aid the development process [5–8], and other countermeasures such as intrusion detection systems [9, 10], version control [11, 12], and authentication [5, 13, 14]. To combat the problem there is a need for multiple different solutions that consider the development methodology used by the developers, their experience level, and the size of the company applying them.

This is the first Systematic Literature Review (SLR) on the software supply chain as a whole, that we are aware of. This paper's main contributions are:

- A collection of supply chain attacks discussed in scientific literature.
- A presentation of current countermeasures used to prevent supply chain attacks.
- An evaluation of the main differences when trusting closed-source instead of open-source software in regards to supply chain security.
- An assessment of trust criteria used when deciding which third-party solutions are to be trusted in the supply chain.
- A collection of tools used to detect vulnerabilities related to supply chain attacks.

In chapter 2 software supply chain security and its relevant concepts will be introduced. After that, chapter 3 will present how we conducted the SLR to find our results. The next section, chapter 4, will show the results from the SLR. The second to last chapter, chapter 5, is where the results will be discussed. The last chapter 6 will conclude the paper.

Chapter 2

Background and Related Work

In this section, the different concepts that are relevant for this SLR are explained. First, a short introduction to software security is given, then a description of open- and closed-source and their differences are given. After that examples of real-world attacks and countermeasure techniques are presented. Finally, additional SLRs related to the area of software supply chain security will be presented.

2.1 Software security

A definition of software security is "software security is a specific concept within the overall domain of information security that deals with securing the foundational programmatic logic of the underlying software" [15]. Secure software helps both business and information security. Not using secure software, e.g. outdated and deprecated software, leaves both organizations and individuals exposed to threats like malicious hackers, viruses, malware, and worms. If malicious software is inserted into your computer or program the consequences can potentially be devastating. IBM releases a yearly report showing the cost of data breaches for the given year, and in 2022 the report stated that 83% of the organizations studied had a data breach and the average cost for a data breach was $4.45 million [16].

Security is a non-functional requirement since it serves as a constraint on the system design. A non-functional requirement is a specification that outlines standards that may be used to evaluate the system's quality as opposed to particular behaviors. Security is one of these measures of quality. Other examples are modifiability, maintainability, and scalability. Since these requirements are concerned with the system's quality and not actions or behaviors like functional requirements are, it is harder to measure and observe that they are implemented just by using the software. Because security is a non-functional requirement it is easy to downplay its importance and rather prioritize new functionality instead of security during software development.

Neglecting the importance of good software security could have dire consequences for the company developing the software. If a company gets compromised it would hurt its reputation and perceived trustworthiness. The General Data

Protection Regulation (GDPR) introduced by the European Union has introduced liability to the software publishers and the need to compensate victims if their personal data gets stolen by attackers [17].

2.1.1 Software supply chain security

Modern software is usually relying on multiple dependencies to function as expected. However, these dependencies might introduce security risks to a project. As mentioned it can be challenging to evaluate the security of software just by using it without extra information from the provider. This issue of properly evaluating security is what leads to the challenges of defending the software supply chain. The security risk might be introduced intentionally as malware, but it could also be introduced unintentionally as a consequence of careless security practices of the third party. Software that unintentionally introduces security risks is also known as "weakware" [18]. Both are dangerous since the software, in general, is only as strong as its weakest link [9].

There are multiple parts of the software supply chain that can be attacked and need to be secured. Software supply chain security is a complex topic as it is affected by multiple disciplines such as supply chain and operations management, security, cryptography, telecommunications, computer science, information systems, e-commerce, insurance, and risk analysis [19]. The third-party dependencies that are used also have their own software supply chain that can be compromised. These elongated software supply chains that lack transparency are part of what makes securing the supply chain such a complicated issue.

A relatively recent example of vulnerabilities introduced by third-party dependencies is the Log4Shell zero-day vulnerability in the Java logging framework Log4j, which was reported on December 9, 2021. Log4Shell allows a remote attacker to take over an internet-connected device if the target is running a particular version of Log4j [20]. Many companies were using Log4j directly or using third-party software that used Log4j. The Apache Software Foundation gave Log4Shell the highest severity rating on the Common Vulnerability Scoring System (CVSS) [21]. The Log4Shell vulnerability is an example of how a vulnerability in one software solution can lead to security risks in many projects using it.

2.2 Open-source and closed-source software

When writing new software or starting new projects it is uncommon to build everything from scratch. Therefore programmers often use third-party software. Third-party software can be either open-source or closed-source. With open-source software, you not only have access to use the code or program, but you have free access to view, edit and share the source code as well [22]. The open-source philosophy is also a way of thinking and collaborating based on intellectual freedom and core principles like transparency, inclusion, and community. With open-source software, it is common to rely on collaboration and feedback from other programmers who are part of the online community.

Looking at the security aspect, having multiple contributors to a project can be both positive and negative. Having more people look over the code can increase the chance of finding vulnerabilities, but at the same time having multiple people writing code increases the chance of someone adding vulnerable code either intentionally or unintentionally. The source code from open-source projects is often shared on code hosting platforms such as GitHub, GitLab, and SourceForge. The code can be distributed with package managers such as npm, PyPi, and apt or it can be downloaded directly from distribution websites.

Looking at closed-source software, also known as proprietary software, the difference from open-source is that closed-source comes with restrictions. The biggest restriction is that you do not have access to the source code and as a consequence, you have no way of checking out the code for yourself [22]. This can make it harder to evaluate the security level and see which, if any, countermeasures have been used. However, as you have full control over the code it is common for companies to write in-house software that is only accessible to other programmers in the same company.

The scientific literature is more focused on open-source software than closed-source. This is a natural consequence of open-source software being open, as there is more to write about and discuss. However, they are both important to look at especially from a security standpoint, and closed-source software has recently been in the searchlight as a potential attack vector [23].

2.3 Real world attacks

Supply chain attacks are not just a theoretical issue, there are multiple examples of real-world attacks. The recent SolarWinds hack in 2020 is probably the most known attack. The attack was used to upload the Solarigate malware, also known as Sunburst or UNC2452, to SolarWinds' customers' servers [10, 23, 24]. SolarWinds Orion Platform is a network management system created to aid customers to manage their IT resources. SolarWinds had around 33000 customers at the time of the hack, which might have downloaded the compromised version of the Orion platform that contained backdoor malware that allowed the hackers to operate in the customers' network without any restrictions [10]. Big tech companies such as Microsoft, Nvidia, Intel, Cisco, Belkin, and VMware were affected, and their compromise might lead to new hacks in the future since the hackers might have obtained confidential information that can be used in new attacks [10, 23].

Another example is the Target breach that happened in December 2013. In this attack, hackers got access to 40 million credit and debit card accounts by stealing credentials of Target's heating, ventilation, and air conditioning vendor [25]. The hackers then used the credentials to access Target's vendor-dedicated web services [26]. This breach cost Target 202 million dollars [27] and it shows how intricate these attacks can be, but also that the potential gains for malicious actors can be huge.

It is easy to believe that software supply chain attacks are only a recent concern due to the increased reliance on third-party software. This is, however, not the case as already back in 1984 Ken Thompson demonstrated how a software supply chain attack could be carried out with a C compiler modified to be a Trojan [27–30]. The attack is known as a "Trusting Trust attack", and the is done by modifying the compiler so that when it is compiling its own binary, it keeps the Trojan and when it compiles some other code it adds a backdoor.

One example showing the relevance of supply chain attacks is the recent spread of the SocGholish malware, also known as FakeUpdates, through multiple media websites. A compromised media company, serving content to multiple large news organizations, has been used by hackers to serve malicious JavaScript code. More than 250 American newspaper websites, relying on the compromised company, have been affected [31, 32].

2.4 Countermeasures

Even though securing the software supply chain is a complex task there are multiple measures developers can take at each step of the supply chain to strengthen it. There are many parts of the software supply chain that needs to be considered for protection such as third-party dependencies, container images, assessments of the Infrastructure as a Service (IaaS) provider, etc. When using third-party dependencies developers can perform different kinds of analysis on them and evaluate their trustworthiness [33]. Some tools are linked to specific package managers such as npm audit for npm packages and Eclipse Steady for Java dependencies. Other tools are for general open-source such as Snyk and Github Dependabot.

To help secure the software supply chain there are also multiple frameworks that can provide knowledge to developers about software security and software supply chain risk. Some organizations that publish such frameworks are the National Institute of Standards and Technology (NIST), the Open Web Application Security Project (OWASP), and the International Organization for Standardization (ISO). Developers can also use lists of vulnerabilities and weaknesses such as Common Vulnerabilities and Exposures (CVE), Common Weakness Enumeration (CWE), and Common Attack Pattern Enumeration and Classification (CAPEC) to help prioritize what to secure.

Tools, lists, and frameworks are not the only aids developers can use, as good coding practices will help as well. Code reviews, testing, and version control will help protect against attacks and can also help with reducing the damage if an attack happens. In addition, there are security practices that are not only relevant for software supply chains. For example, one should not only consider the happy path, and introducing fail-safes should help to hinder cascading failures. Design principles such as encapsulation of less trustworthy services and loose coupling with such services might also help defend the supply chain.

2.4.1 Countermeasures and agile development

The countermeasures need to fit with the software-building methodology used by the company. Agile development is increasing, and currently, 71% of US companies are using agile development methodology [34], making it the most popular software-building methodology in the US. Therefore the countermeasures should be in compliance with the agile working practices and not break the agility aspect. Automatic, quick, and flexible solutions are to be preferred to fit with the changing requirements associated with agile development. Overall, the countermeasures proposed need to be usable and provide enough value for the work the developers put into using them.

2.5 Related work

There have been multiple literature reviews written on various topics relevant to software security and supply chain, but not software supply chain security. Latif et al. studied 41 articles back in 2019 where they focused primarily on network security, information security, web application security, and the internet of things (IoT) [35]. They found out that the highest publishing value came from the Scopus database and that further research into cyber security in supply chain management was needed.

In 2020 Zhao et al. did another literature review where they reviewed 56 papers from 1999 to 2020 [36]. Their focus area was on open-source software and what evaluation criteria should be used to evaluate and compare open-source projects. More specifically they looked at 5 indicators designed for open-source software and performed a correlation analysis on the indicators. The indicators are code, license, popularity, developer, and sponsorship. Security was one of the issues they looked into but they did not exclusively look into this. They found many correlations among the indicators with the most relevant being that project status, age, activity, copyleft, and developer interest will promote its popularity. Also, having a license will reduce the popularity.

A third SLR comparing papers between 1990 and 2017 was performed by Ghadge et al. They wanted to answer the research question "How can organizations manage cyber risk in the supply chain?" and chose to look at five themes: cyber risk types, cyber risk propagation, cyber risk points of penetration, cyber security challenges, and mitigation measures [37]. Their findings point at the human or behavioral elements and these are found to be critical. In addition, their findings show that there is a need for raising risk awareness, standardizing policies, collaborating on strategies, and creating common empirical models to increase cyber-resilience.

Chapter 3

Research Methodology

In this chapter, the research methodology used is described. The guidelines by Kitchenham and Charters on how to do a SLR were used [38]. In the guidelines, they describe three phases that are part of an SLR. These phases are planning, conducting, and reporting the review. In this chapter, the focus is on the planning and conducting phase. The planning phase includes a description of the research questions and review protocol which contains both the search strategy and the inclusion and exclusion criteria. The conducting phase contains study selection steps, data extraction in Excel and MaxQDA, and finally the data synthesizing strategy.

3.1 Research Questions

The reason for doing this literature review is to gain more insight into how the research field and companies deal with software supply chain attacks. We want to find out what issues they face, what measures they are currently taking to prevent these attacks, and if there are differences between open- and closed-source projects. The research questions (RQ) this SLR wants to answer are:

- **RQ1 Attacks**: What types of supply chain attacks are prominent and discussed in academic literature?
- **RQ2 Countermeasures**: What are the current countermeasures used to prevent supply chain attacks?
- **RQ3 Closed- vs. Open-Source**: What are the main differences when trusting closed-source instead of open-source in regard to software supply chain attacks?
- **RQ4 Third party**: What are the criteria used when deciding which third-party solutions are to be trusted in the supply chain?
- **RQ5 Detection Tools**: What are the tools used to detect vulnerabilities related to supply chain attacks?

RQ1 Attacks: What types of supply chain attacks are prominent and discussed in academic literature?

By asking this question the intention is to find out what attacks are most discussed today. This is important both for getting an overview of the different attacks known and to see what the literature finds the most interesting to look more into. Knowing this could give a guideline as to what part of the supply chain is the most vulnerable and potentially where it is suitable to put in defense mechanisms.

RQ2 Countermeasures: What are the current countermeasures used to prevent supply chain attacks?

To be better suited for protecting the software supply chain knowledge about what current countermeasures are being used to stop attacks is valuable. Answering this question could give some insight into which countermeasures could be further improved or which ones should be switched out with new and better approaches. It can potentially also give insight into how many different countermeasures are being used and what parts of the software supply chain are being prioritized.

RQ3 Closed- vs. Open-Source: What are the main differences when trusting closed-source instead of open-source in regard to software supply chain attacks?

The difference between closed- and open-source software is that with open-source software everyone, including attackers, is able to see the source code which describes how the program operates. On the other hand with closed-source solutions, it is not possible for anyone to see the source code, which could potentially make it safer, but at the same time, you will not get help from the public for finding and patching vulnerabilities [39].

By asking this question the objective is to see if there are differences in trust when using source code that is open for everyone compared to code that is closed-source. This question could potentially also give an insight into differences in how the public operates in contrary to private companies that might use a lot of private in-house code.

RQ4 Third party: What are the criteria used when deciding which third-party solutions are to be trusted in the supply chain?

Third-party software solutions are reusable software or components that are available and owned by someone else and most often distributed by a company [40]. The goal with this RQ is to see if there are any common procedures for deciding what third-party components that are safe to use. This could give insight into what steps are currently taken for addressing safe code and what one could do to make sure the data sent and retrieved from third-party services are secure.

RQ5 Detection Tools: What are the tools used to detect vulnerabilities related to supply chain attacks?

When it comes to writing secure software there is always a chance of human errors and mistakes being made. It is therefore important to develop good tools and methods for detecting potential vulnerabilities so that one could fix the issue before it is exploited by an attacker. This is a constant job as most software projects are constantly under development. This RQ focuses on the method used to detect attacks and vulnerabilities in the software supply chain. Knowing which parts of the software supply chain that have detection tools made for it and not give a better starting point for future work on discovering and improving current methods.

3.2 Data Sources and Search Strategy

Paper Databases

When deciding on where to gather the papers from, we wanted to use large libraries that we were familiar with. The decision, therefore, fell on Scopus and Web of Science. Other libraries were considered but after having done some different searches in the two libraries we found out that we got a high enough number of articles from those two to make sure the papers were representative.

As part of the preliminary search, we tried out multiple combinations of strings in both Scopus and Web of Science and compared the number of articles and their relevance. When trying out different search strings we first checked that the number of articles found was high enough to get some valuable data but still low enough that we could cover all of it. We then did some random sampling and read the first sentences of multiple abstracts to see if the searches gave the resulting types of papers we were looking for.

Search Strings

The search strings we ended up using are in Table 3.1. Both Scopus [41] and Web of Science [42] supports a Boolean syntax, which made it possible to express the same search in both libraries even though the syntax is a bit different.

Table 3.1: Search Strings

Source	Search String
Web of Science	ALL = ((Supply AND Chain) AND (Security) AND (Software OR Microservice OR Agile))
Scopus	TITLE-ABS-KEY("supply") AND TITLE-ABS-KEY("chain") AND TITLE-ABS-KEY("security") AND (TITLE-ABS-KEY("software") OR TITLE-ABS-KEY("microservice") OR TITLE-ABS-KEY("agile"))

3.3 Inclusion and Exclusion Criteria

To make the filtering process of the papers more efficient and to make sure the inclusion and exclusion of papers were done as similarly as possible by both researchers we started by defining some inclusion and exclusion criteria. These were made after having done the initial search. Then after reading through 20 abstracts together we added some more criteria. During the remaining filtering, some small changes were made in the criteria when we found new trends either relevant or not relevant that did not break with the criteria for the papers already filtered. We found out that the most common topic of papers we excluded was about physical supply chains using radio frequency identification and not related to software. Table 3.2 lists the final inclusion and exclusion criteria.

Table 3.2: Inclusion and exclusion criteria

	Inclusion criteria
IC1	Papers that describe techniques of preventing software supply chain attacks or vulnerabilities in the software supply chain.
IC2	Papers that mention trust in open-source or closed-source software.
IC3	Papers that mention third-party solutions in the software supply chain context.
IC4	Papers describing companies' approach to software supply chain security.
	Exclusion criteria
EC1	Papers that do not have software as the main focus.
EC2	Papers about blockchain.
EC3	Papers focusing on AI or Machine Learning.
EC4	Papers focusing on hardware.
EC5	Papers focusing on agriculture.

Note that even though we decided to exclude papers discussing blockchain, we decided to include one paper *Blockchain Technology for Secure Supply Chain Management: A Comprehensive Review* (P13). This paper was included since we wanted to explore if blockchain technologies could have any relevance to the software supply chain and since P13 was a comprehensive review, we thought it could be a good summary of use cases of blockchain for the supply chain.

3.4 Quality Criteria

When evaluating the papers it is considered critical to assess their quality [38]. Quality criteria are used in combination with the inclusion and exclusion criteria. There is not one clear definition of "quality" but a common interpretation is that the quality is related to what extent a study minimizes bias and maximizes internal and external validity [38]. In this project, quality criteria were initially not explicitly stated as an early search resulted in few papers. Because of the small number of papers we did not filter on quality criteria, since we did not want to exclude more papers than was strictly necessary. However, when going through the papers we found some criteria such as how well the research was described and if the results seemed plausible based on the research method. These criteria were used in combination with how well the papers answered the RQs to decide and rate the relevance of the papers.

3.5 Study Selection Steps

The filtering of papers from Scopus and Web of Science was done in multiple steps and is illustrated in Figure 3.1. Before we could start the filtering we had to find the relevant papers, and this was done as described in section 3.2. The searches were executed in September 2022 using the search strings from Table 3.1. The search resulted in 609 papers from Scopus and 317 papers from Web of Science. The information from Scopus was exported to RIS format and then into Endnote. The information from Web of Science was exported directly into Endnote. Endnote is a reference management tool [43], and we first used it to identify duplicate references. There were 164 duplicates, which we filtered out, and after the initial filtering, we had 762 papers.

In the second filtering step, we read through the abstract of the 762 papers and filtered based on the inclusion and exclusion criteria. Firstly, we both filtered a set of 20 papers together to see that our interpretation of the inclusion and exclusion aligned. We had a 92% agreement on which articles should be included and excluded, and after a quick discussion, we agreed that the last 8% was small enough to not really affect the end result. Based on the high percentage of agreement, we decided to split the papers into two parts to evaluate them separately. We

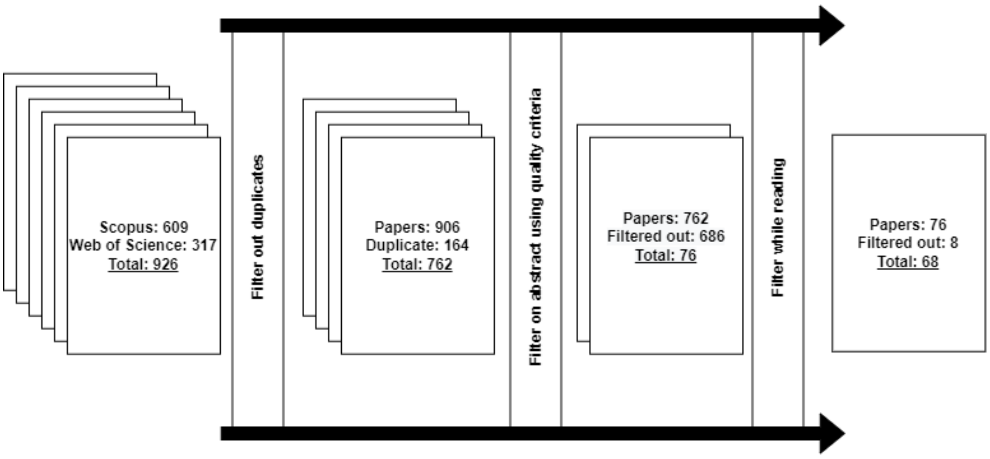


Figure 3.1: The process of filtering papers.

deiced to filter the papers into the three folders "include", "exclude", and "uncertain". After the separate filtering 61 papers were considered included, and 54 we were uncertain about. After examining the 54 papers together we decided to keep 15 of them, which led to 76 relevant papers based on the inclusion and exclusion criteria, and 686 were filtered out.

The last filtering happened naturally while reading through the papers. There were some papers that were not possible to obtain and some were written in another language than English, even though, the abstract and title were in English. The excluded papers from this step were **P05**, **P16**, **P38**, **P42**, **P52**, **P56**, **P61**, and **P75**. These papers were also removed from Table 4.1.

3.6 Data Extraction and Data Synthesis

After collecting the 68 papers we evaluate them based on a data extraction form made in Excel. The data extraction fields are presented in Table 3.3, where we use "#" to note that there could be multiple of this field. Most of the fields were used to extract qualitative data and needed some interpretation from us to extract the important parts. The extraction process was probably affected by researcher reflexivity as our assumptions and beliefs affected what we consider noteworthy.

We used the data we extracted during data extraction to do data synthesis. The information from the data synthesis is included in chapter 4.

3.7 Analysing online resources

As part of the data extraction, we also looked into two discussion forums to see if there were some aspects of securing the supply chain we had missed. We were curious to see if software developers and other computer security practitioners discussed topics not explored in scientific literature.

Table 3.3: Relevant information extracted from the papers.

Type	Data Extraction Fields
Metadata	Status, Extraction date, Title, Author, Year, Source
Publication feature	Type of paper, Goal of the paper, Research questions/hypotheses, Study type
General	Results, Findings, Implications, Threat to validity
Other	Comments
RQ1: Attacks	General vulnerabilities, Attack method #, Procedure #, Risk #, Severity #
RQ2: Countermeasures	Countermeasure #, Name/Description #, Technique #, Tool #, Recommendation #, Feasibility #
RQ3: Closed-source vs. open-source	Discusses closed-source, Discusses open-source, Differences between sources
RQ4: Third-party solutions criteria	Discusses using third-party solutions, Specific mentioning of third-party solutions, Deciding trust criteria
RQ5: Tools to detect vulnerabilities	Tool #, Type of vulnerability #, Which part of the supply chain is affected #, How it works #

We searched on Stack Overflow with the search string *“Supply chain” attacks* and *“Supply chain” security*, which gave 24 and 36 results, respectively. We also searched Security Stack Exchange with the search string *“Supply chain”*. This gave 153 results. These searches were done in September 2022.

These searches did not provide much information of value, since they were very general about the supply chain or how to implement specific tools. The main takeaways were that the French government introduced a “first-level security certification” *La Certification de Sécurité de Premier Niveau* (CSPN), similar to Common Criteria (CC) in 2008, to help establish trust in products [44], and a security principle that one should use a four-eyes principle when including new dependencies [45]. Overall, there were few results, which can be an indication that the problem is not widely known or more likely that there are no good solutions to tackle the issue.

Chapter 4

Results

The results from the SLR are presented in this section. The results are divided into subsections to help answer the RQs presented in section 3.1. Tables and graphs are used to give further insight into the information collected from the 68 papers used in this SLR.

4.1 General results

The papers used as part of this SLR were published between 2002 and 2022. Figure 4.1 shows the graph of the number of papers published per year. The two spikes coincide with the Target breach in 2013 and the SolarWinds Hack in 2020. Overall, there is an upward trend in the number of articles released on this topic, and the decrease in 2022 is likely due to the delay between when papers are written and when they are published.

The 68 papers were also given a classification based on how related they were to software supply chain attacks and if they could be used to answer any of the RQs. The different categories were "Relevant", "Somewhat relevant", "A few interesting points", "Not very relevant", and "Not relevant". The distribution was as follows; "Relevant": 19 papers, "Somewhat relevant": 9 papers, "A few interesting points": 11 papers, "Not very relevant": 20 papers, "Not relevant": 9 papers.

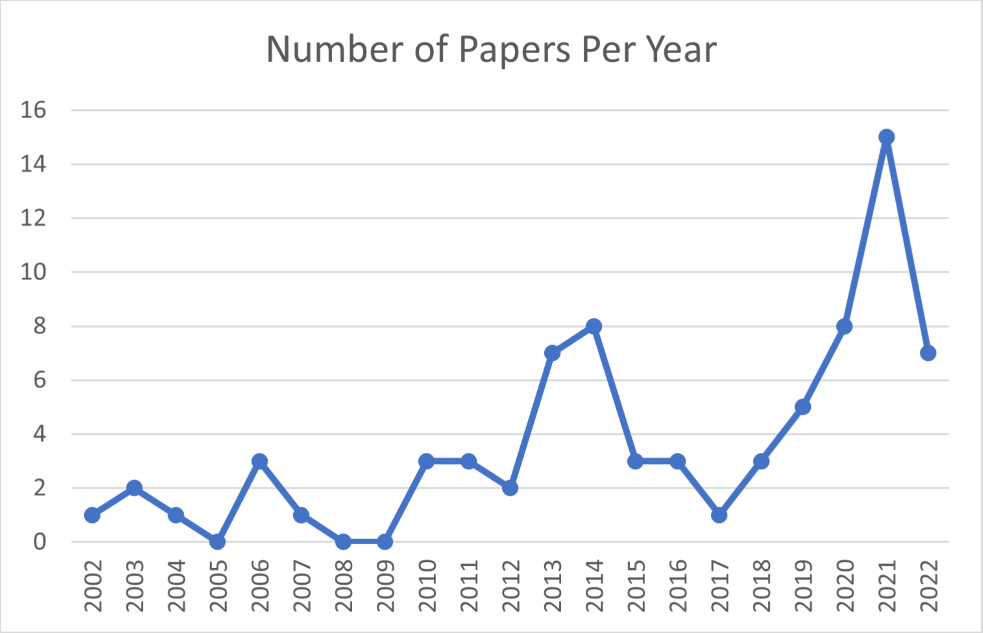


Figure 4.1: The number of papers per year.

A complete list of all the papers with titles, authors, descriptions, and code used to reference them is displayed in Table 4.1, Table 4.2, Table 4.3, Table 4.4, and Table 4.5. The authors with the most articles in this SLR are C. W. Axelrod (P10, P35, P54), R. A. Martin (P09, P73), M. Ohm (P11, P68), and D. L. Vu (P34, P44).

Table 4.1: Overview of the selected papers (1/5)

Code	Title	Authors	Description
P01	Automatic Security Inspection Framework for Trustworthy Supply Chain	Nakano *et al.* [46]	Implemented a framework for supply chain and evaluate the performance.
P02	Information Security in Value Chains: A Governance Perspective	Patnayakuni & Patnayakuni [25]	Presents a framework for information security governance.
P03	A comparative study of vulnerability reporting by software composition analysis tools	Imtiaz *et al.* [47]	Studies the difference in vulnerability reporting by various software composition analysis tools.
P04	A Socio-technical Framework for threat Modeling a Software Supply Chain	Sabbagh & Kowalski [5]	Studies software supply chain security problems from a systemic viewpoint. It addresses three main issues: modeling the target system, identifying threats, and analyzing countermeasures.
P06	Advancing software assurance with public-private collaboration	Mead & Jarzombek [3]	Inform about the DHS SwA Program which seeks to improve people, processes, and technology in the development and acquisition of software.
P07	An Efficient Web Authentication Mechanism Preventing Man-In-The-Middle Attacks in Industry 4.0 Supply Chain	Esfahani *et al.* [13]	They propose a TLS-based authentication mechanism, which is resistant against Man-In-The-Middle Attacks in web applications.
P08	Assessing the Security Risks of Multicloud SaaS Applications: A Real-World Case Study	Akinrolabu *et al.* [14]	Presents the Cloud Supply Chain Cyber Risk Assessment model.
P09	Assurance for CyberPhysical Systems: Addressing Supply Chain Challenges to Trustworthy Software-Enabled Things	R. A. Martin [48]	Addresses issues with insecure software based on CVE, CWE, and CAPEC.
P10	Assuring software and hardware security and integrity throughout the supply chain	C. W. Axelrod [49]	The paper presents a model to ensure supply chain integrity and protect customer assets.
P11	Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks	Ohm *et al.* [11]	Analyses 174 open-source malicious software packages.
P12	Beyond SolarWinds: The systemic risks of critical infrastructures, state of play, and future directions	Raponi *et al.* [10]	Focuses on the SolarWinds attack and how to secure critical infrastructure.
P13	Blockchain Technology for Secure Supply Chain Management: A Comprehensive Review	Agarwal *et al.* [50]	The paper explores options to use blockchain in the supply chain.
P14	Building a Secure Software Supply Chain with GNU Guix	L. Courtès [28]	The paper proposes a model and a tool to authenticate new Git revisions.
P15	CloudChain: A novel distribution model for digital products based on supply chain principles	Vazquez-Martineza *et al.* [51]	Presents a distribution model called CloudChain.
P17	Containing Malicious Package Updates in npm with a Lightweight Permission System	Ferreira *et al.* [12]	A permission system where they encapsulate npm packages.
P18	Critical success factors of web-based supply-chain management systems: An exploratory study	Ngai *et al.* [52]	Reports the results of a survey on the critical success factors of web-based supply chain management systems.
P19	Cyber security in supply chain management: a systematic review	Latif *et al.* [35]	A literature review of 41 articles related to cyber security in supply chain management.

Table 4.2: Overview of the selected papers (2/5)

Code	Title	Authors	Description
P20	Cyber supply chain risk management: Revolutionizing the strategic control of critical IT systems	S. Boyson [6]	Presents a survey of practices aimed at achieving structural integration and risk management across the IT supply chain.
P21	Cyber supply chain security practices DNA - Filling in the puzzle using a diverse set of disciplines	N. Bartol [53]	Describes how the cyber supply chain community has evolved towards creating standards and best practices.
P22	Data security services, solutions and standards for outsourcing	Hamlen & Thuraisingham [54]	Presents security challenges that result from outsourcing data management and software development activities.
P23	Digital Enterprise and Cyber Security Evolution	Raicu & Raicu [55]	Address the evolutionary cyber threats in the last decade using public resources.
P24	Each person and organization in the supply chain path "touches," or has influence on, the security and resilience of software used to control products, systems, and services	R. Stempfley [56]	Presents how people and organizations influence and affect the software supply chain.
P25	Enhancement of confidence in software in the context of international security	Markov & Sheremet [57]	Presents an overview of international aspects of software security enhancement.
P26	Ensuring software assurance process maturity	Wotring & Migues [58]	Describe a software assurance checklist that companies can use to understand their supply chain and its vulnerabilities.
P27	Ensuring your development processes meet today's cyber challenges	Chrissis *et al.* [59]	Provides a brief survey of process capability frameworks, secure lifecycle practices, and implementation approaches.
P28	Estimating the Attack Surface from Residual Vulnerabilities in Open Source Software Supply Chain	Yan *et al.* [60]	This paper discusses the number of papers that contain vulnerabilities and percentages that are patched.
P29	Evaluation indicators for open-source software: a review	Zhao *et al.* [36]	Summarize, collate and identify correlations to guide the evaluation of open-source projects.
P30	How international standard efforts help address challenges in today's global ICT marketplace	Shankles *et al.* [61]	Inform and argue why companies should strengthen their software supply chain.
P31	Identity is the new perimeter in the fight against supply chain attacks	B. Hensley [62]	A case study of the SolarWinds attack and explains what to consider regarding software supply chain attacks.
P32	I'll buy that! Cybersecurity in the internet marketplace	Pfleeger *et al.* [63]	Focus on what determines the level and effort software vendors invest in assuring that their products are free from security vulnerabilities.
P33	Integrating Zero Trust in the cyber supply chain security	Amaral *et al.* [64]	Proposes to integrate a Zero Trust architecture in cyber supply chains.
P34	LASTPYMILE: Identifying the Discrepancy between Sources and Packages	Vu *et al.* [2]	Presents LastPyMile which reduces the number of alerts produced by a malware checking tool to a number that a human can check.
P35	Malware, "weakware," and the security of software supply chains	W. Axelrod [18]	Presents contexts in which software supply chain risks are experienced.
P36	Objectives for managing cyber supply chain risk	M. Windelberg [65]	Recommends a set of objectives for cyber supply chain risk management and examines the connotations of each objective with the intent to improve risk coverage.

Table 4.3: Overview of the selected papers (3/5)

Code	Title	Authors	Description
P37	On SolarWinds Orion Platform Security Breach	Sterle & Bhunia [24]	Analyze the security incident regarding SolarWinds' Orion Platform.
P39	On the Feasibility of Detecting Software Supply Chain Attacks	X. Wang [27]	Presents the technical challenges when defending against software supply chain attacks.
P40	On the security cost of using a free and open source component in a proprietary product	Dashevskyi *et al.* [66]	Presents publicly available factors to identify which has an impact on the security effort of using open-source software.
P41	Open industry standards for mitigating risks to global supply chains	Szakal & Pearsall [67]	The paper defines the global tech supply chain landscape in which global tech vendors are operating.
P43	Poisoning the Software Supply Chain	E. Levy [29]	Investigates how software's distribution, both open-source and proprietary, can invite attacks.
P44	Towards Using Source Code Repositories to Identify Software Supply Chain Attacks	Vu *et al.* [68]	Propose an approach to detect code injected into software packages by comparing their distributed artifacts with the source code repository.
P45	Research on Software Development Process Assurance Models in ICT Supply Chain Risk Management	Xie *et al.* [69]	Propose a software security assurance model for software development.
P46	Risk assessment method for IoT software supply chain vulnerabilities	Zhu *et al.* [7]	Designs and proposes an objective risk assessment method for IoT software chain vulnerabilities, and verifies the feasibility and effectiveness of the method through experiments.
P47	Safety engineering with COTS components	O'Halloran *et al.* [70]	Explore the application of an integrated approach to safety engineering in which assurance drives the engineering process.
P48	Secure Software by Design	S. Rothschild [8]	Presents the requirements to reduce attack and mitigate damage when attacked by malware and reduce the financial loss when a malware attack is successful.
P49	Securing high-tech goods across the supply chain	C. Gottlieb [71]	Discusses how manufacturers can begin to target their investments in supply chain security appropriately by segmenting their products according to two characteristics: complexity and criticality.
P50	Securing Software Ecosystem Architectures Challenges and Opportunities	Scacchi and Alspaugh [72]	Discuss how vulnerabilities in software supply chain processes and ecosystems can be mitigated, and how local enterprise- or platform-specific cybersecurity can be improved.
P51	Security Evaluation Criteria of Open-Source Libraries	Mills & Butakov [73]	Provides the basis for a simple-to-use checklist that can be used to quickly analyze open-source libraries.
P53	Security in the cyber supply chain: A Chinese perspective	Rongping & Yonggang [74]	The paper presents the Chinese practices when it comes to securing the cyber supply chain.

Table 4.4: Overview of the selected papers (4/5)

Code	Title	Authors	Description
P54	Software security assurance SOUP to NUTS	C. W. Axelrod [75]	Use an expanded version of the Cynefin Framework to come up with preferred approaches to categorizing software supply chains.
P55	SolarWinds Software Supply Chain Security: Better Protection with Enforced Policies and Technologies	Yang *et al.* [23]	Investigates what caused the SolarWinds attack and what solutions we might have to prevent similar attacks in the future.
P57	Supply Chain Risk Management - Understanding Vulnerabilities in Code You Buy, Build, or Integrate	P. R. Croll [76]	Describes the scope of the problem regarding software vulnerabilities and the current state of the practice in static code analysis for software assurance.
P58	Supply Chain Trust	Kshetri & Voas [26]	The main goal is to enlighten the public about the security problems with software supply chains.
P59	Supply-Chain Risk Management: Incorporating Security into Software Development	Ellison & Woody [77]	Describes practices that address software defects and mechanisms for introducing these practices into the acquisition life cycle.
P60	Supply-Chain Security for Cyberinfrastructure [Guest editors' introduction]	Forte *et al.* [78]	Discusses the integrity of hardware and software components across their life cycle of design, manufacturing, distribution, integration, and updating.
P62	Talking about the Software Supply Chain	A. Wirth [79]	Discusses the importance of the composition of your products and diligently managing the security of your software supply chain.
P63	The challenge of cyber supply chain security to research and practice - An introduction	Linton *et al.* [19]	Argues that the academic literature, the research, and publications in the area of cyber challenges are rather sparse and the text intendeds to act as a resource and as a call to research.
P64	The Diversification and Enhancement of an IDS Scheme for the Cybersecurity Needs of Modern Supply Chains	Deyannis *et al.* [80]	This work presents the implementation of a low-cost reconfigurable intrusion detection system (IDS) that can be easily integrated into supply chain networks.
P65	Threat analysis in the software development lifecycle	Whitmore *et al.* [81]	This project investigated aspects of security in software development, including practical methods for threat analysis. The project also examined existing methods and tools, assessing their efficacy for software development within an open-source software supply chain.
P66	Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations	W. Enck & Williams [82]	Presents the top five challenges in software supply chain security that they identified through running three summits with a diverse set of organizations.

Table 4.5: Overview of the selected papers (5/5)

Code	Title	Authors	Description
P67	Towards An Analysis of Software Supply Chain Risk Management	Du *et al.* [30]	Presents the risks which software supply chain is facing, some risk management practices, and some concrete measures to improve software supply chain risk management.
P68	Towards detection of software supply chain attacks by forensic artifacts	Ohm *et al.* [83]	Presents a first analysis of observable artifacts of malicious packages and a possible mitigation strategy that might lead to more insight in long term.
P69	Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages	Duan *et al.* [84]	Proposes a comparative framework to qualitatively assess the functional and security features of package managers for interpreted languages.
P70	Trust and tamper-proof software delivery	Naedele & Koch [85]	Examines different trust models in the software supply chain.
P71	Trust Engineering - Rejecting the Tyranny of the Weakest Link	S. D. Alexander [9]	Seeks to expand practitioners conversant with concepts of trust engineering.
P72	Universal and secure object ownership transfer protocol for the Internet of Things	Ray *et al.* [86]	Proposes a novel ownership transfer mechanism that securely transfers an RFID tagged objects in Internet of Things (IoT).
P73	Visibility & Control: Addressing Supply Chain Challenges to Trustworthy Software-Enabled Things	R. A. Martin [4]	Ideas on if Software Bill of Materials (SBOM) can be used in tool-to-tool exchanges.
P74	We cannot blindly reap the benefits of a globalized ICT supply chain!	Davidson & Shankles [87]	Addresses the Supply Chain Risk Management challenges, but not software related, faced by the US.
P76	What are Weak Links in the npm Supply Chain?	Zahan *et al.* [33]	Describes and discusses weak link signals in the npm supply chain.

4.2 RQ1 - Attacks

RQ: *What types of supply chain attacks are prominent and discussed in academic literature?*

After reading through the different papers we found that the most discussed attack was the recent 2020 SolarWinds hack. From Table 4.6 it is also clear that most of the attacks are only mentioned in one of the papers, with P13 and P43 containing the most. In addition, the SolarWinds hack was the only attack that had papers specifically focusing on it (P12, P37, P55).

Table 4.6: Mentions of real world attacks.

Name	Year	Mentioned in papers
SolarWinds (Solarigate, Sunburst, and UNC2452)	2020	P12, P13, P28, P30, P31, P33, P37, P39, P55, P66, P76
Ken Thompson's Trojaned C compiler	1984	P14, P39, P43, P67
Target Breach	2013	P2, P39, P58
Kaseya	2021	P13, P39
Apple Xcode	2021	P13, P69
Equifax	2017	P50, P58
Mimecast	2021	P13
SITA	2021	P13
ClickStudios	2021	P13
BigNox	2021	P13
Verkada	2020	P13
Able	2020	P13
Ledger	2020	P13
Ticketmaster	2018	P58
British Airways	2018	P58
Irssi	2002	P43
Fragroute/Fragrouter/Dsniff	2002	P43
BitchX	2002	P43
OpenSSH	2002	P43
Sendmail	2002	P43
Fragroute	2002	P43
Libpcap/Tcpdump	2002	P43
AIDE	2001	P43
Microsoft distributing Concept	1995	P43

As seen in the Figure 4.2 about attack methods in the papers, "Denial-of-service" and "Insider threat" are mentioned the most. The rest of the attack methods are quite divided and we have both people-focused and software-related attacks. Attack methods are mentioned more than 100 times in the papers.

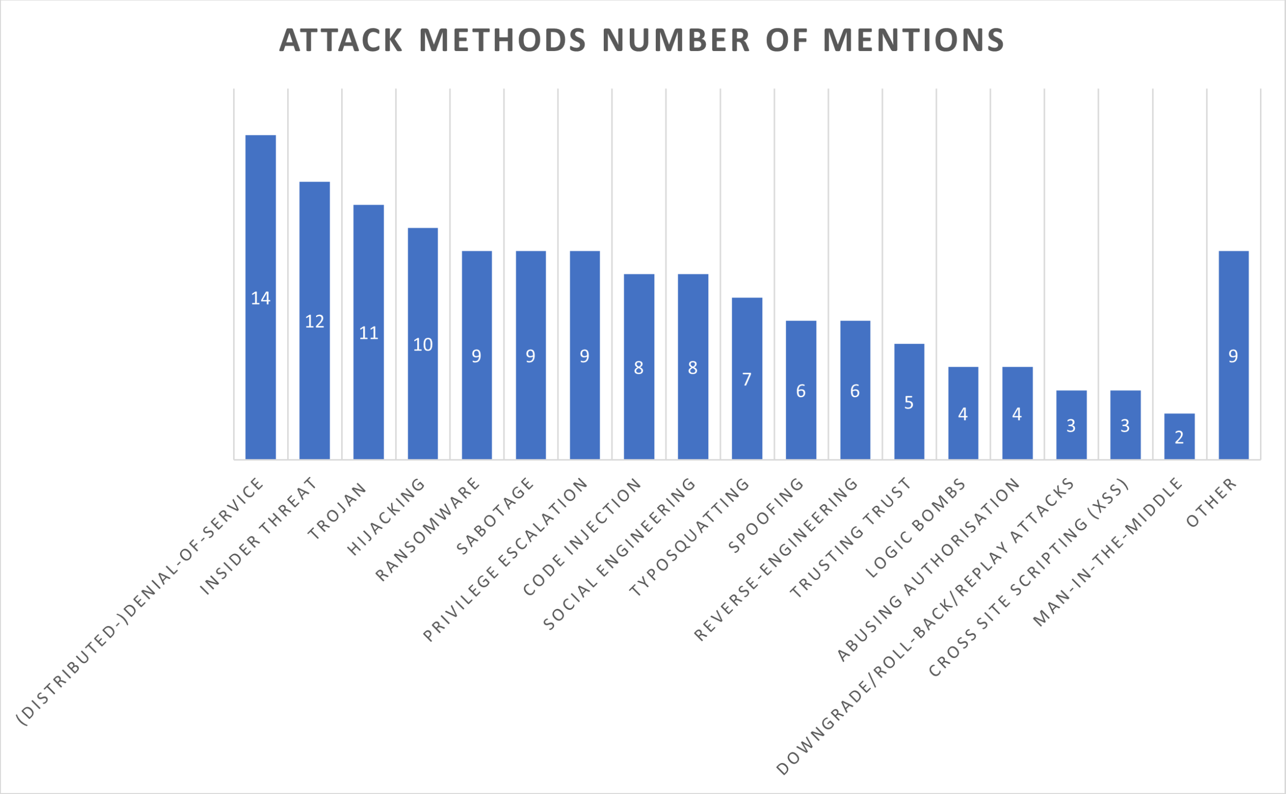


Figure 4.2: Attack methods from the papers.

Table 4.7: Attack methods discussed in papers.

Name	Mentioned in papers
(Distributed-) denial-of-service	P3, P4, P8, P11, P23, P28, P40, P48, P59, P62, P65, P67, P69, P72
Insider threat	P8, P20, P22, P23, P30, P33, P35, P43, P54, P60, P65, P69
Trojan	P11, P12, P28, P29, P37, P39, P55, P57, P60, P66, P74
Hijacking	P11, P34, P39, P44, P50, P62, P67, P68, P69, P76
Ransomware	P11, P12, P28, P31, P33, P39, P62, P64, P69
Sabotage	P7, P8, P23, P35, P49, P54, P62, P69, P74

Continued on next page

Table 4.7 – continued from previous page

Name	Mentioned in papers
Privilege escalation	P8, P12, P17, P27, P37, P55, P57, P59, P69
Code injection	P7, P22, P28, P34, P40, P44, P48, P69
Social engineering	P11, P17, P23, P35, P54, P68, P69, P76
Typosquatting	P11, P17, P34, P44, P66, P68, P69
Spoofing	P4, P48, P59, P65, P67, P70
Reverse-engineering	P4, P37, P45, P48, P57, P60
Trusting trust	P14, P39, P43, P66, P70
Logic bombs	P6, P36, P69, P74
Abusing authorisation	P8, P11, P31, P65
Downgrade/roll-back/replay attacks	P14, P17, P72
Cross Site Scripting (XSS)	P3, P37, P65
Man-in-the-middle	P7, P72
Teleport attack	P14
Mix-and-match attacks	P14
Indefinite freeze attacks	P14
Fast-forward attacks	P14
Cryptojacking	P69
Malvertising	P69
Key loggers	P6
Use after free	P11
Brute force	P70

Table 4.7 displays which papers mentioned the different attacks. It shows that P11 and P69 mentioned the most attack methods. It also displays the attacks that were only mentioned in one paper.

To get some further insight into the RQ we looked at the mentions of trojans, viruses, and worms. In Table 4.8 the trojan Sunburst used in the SolarWinds hack is mentioned the most which makes sense since multiple papers were discussing this attack in detail. We can also see that there are a lot of general mentions of trojans. There are a few mentions of viruses but no virus is mentioned in more than 1 paper, as we can see in Table 4.9. From Table 4.10 one can see that worms were not discussed much in comparison to trojans and viruses.

Table 4.8: Types of trojans discussed in papers.

Type of trojan	Mentioned in papers
Cobalt strike	P55
Sunburst	P12, P33, P37, P39, P55
General mentions	P11, P28, P29, P39, P55, P57, P60, P66, P74

Table 4.9: Types of viruses discussed in papers.

Type of virus	Mentioned in papers
Concept	P43
McAfee	P11
Wannacry	P12
Mirai	P62
Petya	P62
General mentions	P28, P43, P45, P69

Table 4.10: Types of worms discussed in papers.

Type of worm	Mentioned in papers
Stuxnet	P54, P60
SQL Slammer	P35
General mentions	P35

Overall, the papers did not mention many examples of trojans, viruses, and worms. Trojans were mentioned the most and viruses had the most examples. No paper discussed all three kinds of malicious programs.

4.3 RQ2 - Countermeasures

RQ: *What are the current countermeasures used to prevent supply chain attacks?*

To answer the second RQ, we looked at technical countermeasure techniques, relevant frameworks, types of tools, people-focused techniques, and other countermeasure techniques mentioned in the papers. We divided the countermeasure into separate groupings to reflect the different types of countermeasures. Many papers mentioned frameworks that contain many best practices instead of listing the countermeasures themselves.

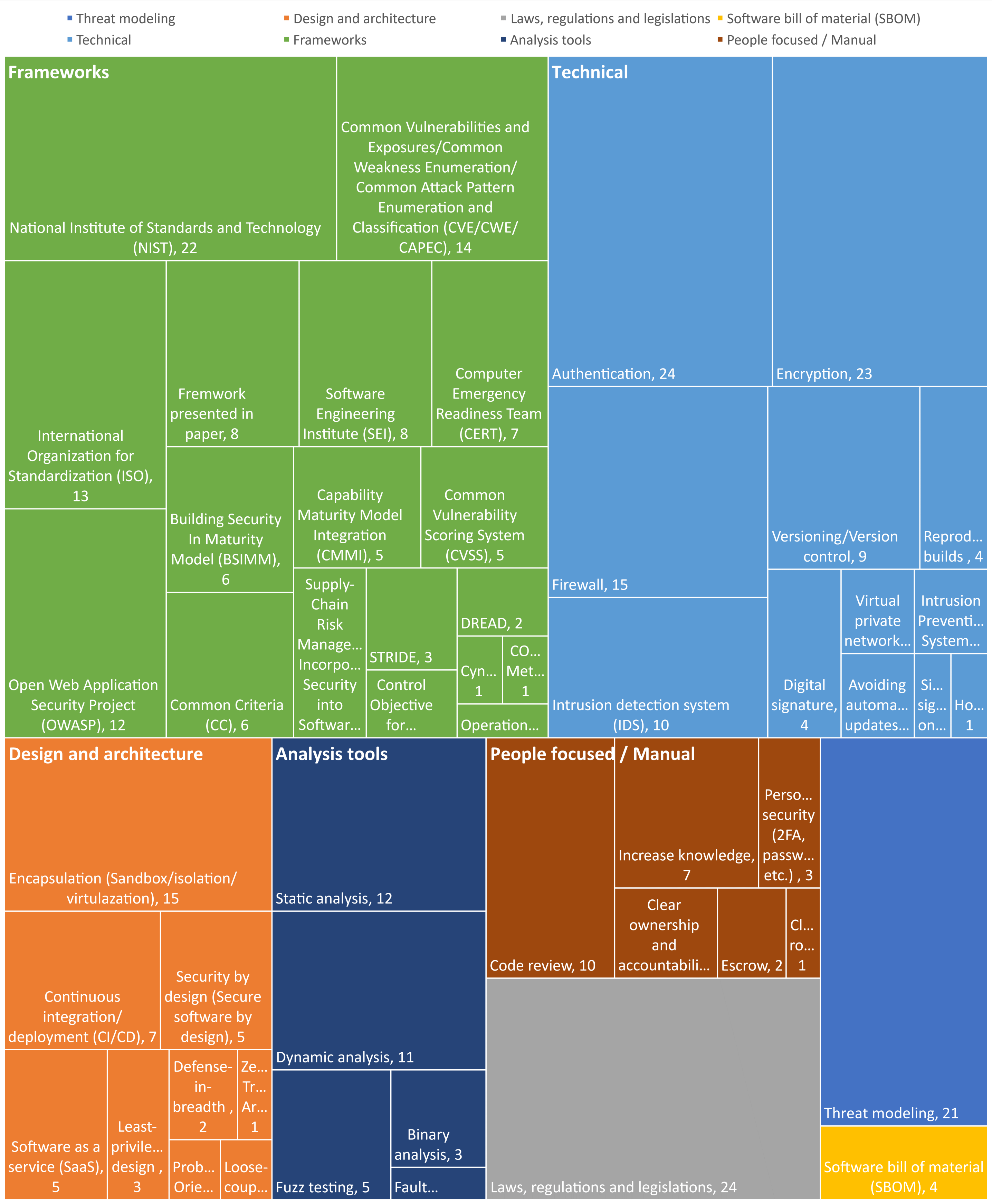


Figure 4.3: The countermeasures from the papers.

Figure 4.3 is a treemap where larger areas are used to visualize a higher number of mentions. The coloring is related to different categories that are displayed as bullet points on top of the graph and in the top left corner of each of the color groups. The number at the bottom of each box represents the number of times the topic related to the box was mentioned in the papers.

From the treemap in Figure 4.3, we see that most papers mention using a framework to support the protection of the software supply chain. Different frameworks were mentioned on 120 occasions in the papers. However, the papers also mention many specific techniques with authentication and encryption being the most common. The different technical countermeasures were mentioned 97 times. Other countermeasures the papers discuss are the importance of good design and architecture, mentioned 40 times, different analysis tools, mentioned 32 times, and people-focused countermeasures, mentioned 26 times. Some papers discuss that there could be laws and regulations to help solve the problem of unsecured software supply chains. This was mentioned 24 times in the papers. Threat modeling and assessment of risk were mentioned in 21 papers. Software bill of materials (SBOM), which we have grouped as a separate countermeasure since it is provided by the software providers, was mentioned 4 times.

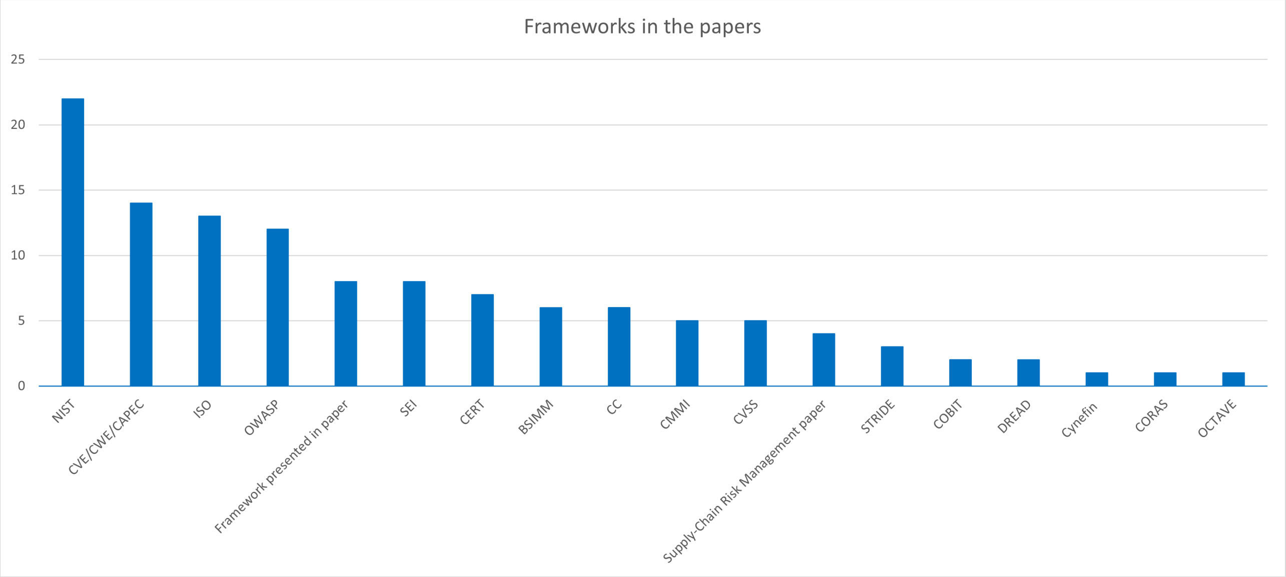


Figure 4.4: The number of papers mentioning different frameworks.

From Figure 4.4 we see the frameworks and organizations mentioned in the papers. NIST, CVE/CWE/CAPEC, ISO, and OWASP seem to be most discussed but there is a great variety of frameworks in the papers. NIST, ISO, and OWASP are not in themselves a single framework, but organizations that provide multiple frameworks, lists, and standards. They are however united in Figure 4.4 to indicate what the trusted authorities are when it comes to knowledge about securing the software supply chain.

Table 4.11: Frameworks in the papers.

Name	Abbreviation	Mentioned in papers
National Institute of Standards and Technology	NIST	P1, P3, P4, P6, P8, P10, P20, P21, P23, P24, P27, P33, P34, P36, P40, P45, P48, P51, P54, P57, P59, P74
Common Vulnerabilities and Exposures/ Common Weakness Enumeration/ Common Attack Pattern Enumeration and Classification	CVE/ CWE/ CAPEC	P1, P3, P11, P25, P28, P29, P33, P40, P54, P57, P59, P65, P67, P69
International Organization for Standardization	ISO	P2, P4, P6, P8, P12, P21, P25, P27, P30, P45, P57, P70, P74
Open Web Application Security Project	OWASP	P3, P6, P8, P11, P20, P26, P27, P35, P46, P48, P65, P66
Framework presented in paper	-	P4, P20, P33, P46, P48, P51, P59, P70
Software Engineering Institute	SEI	P2, P6, P27, P35, P36, P59, P65, P67
Computer Emergency Readiness Team	CERT	P6, P26, P27, P30, P36, P45, P65
Building Security In Maturity Model	BSIMM	P6, P26, P27, P59, P65, P66
Common Criteria	CC	P10, P21, P32, P57, P70, P74
Capability Maturity Model Integration	CMMI	P6, P10, P26, P27, P45
Common Vulnerability Scoring System	CVSS	P1, P3, P11, P29, P46
Supply-Chain Risk Management paper	-	P6, P29, P59, P67
Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege	STRIDE	P48, P59, P65
Control Objective for Information and Related Technology	COBIT	P2, P25

Continued on next page

Table 4.11 – continued from previous page

Name	Abbreviation	Mentioned in papers
Damage, Reproducibility, Exploitability, Affected users, Discoverability	DREAD	P46, P48
CORAS Method	-	P65
Operationally Critical Threat, Asset, and Vulnerability Evaluation	OCTAVE	P65
Cynefin	-	P54

In Table 4.11 we see which papers have mentioned the different frameworks and organizations. There are multiple papers mentioning more than one framework. We can also see from "Framework presented in paper" that there were 8 papers presenting and discussing their own framework: P4, P20, P33, P46, P48, P51, P59, and P70. The Supply-Chain Risk Management paper, P59, was the only of the 8 papers presenting their own frameworks that were also referenced in other papers (P6, P29, P67).

Table 4.12: Design and architecture techniques

Name	Mentioned in papers
Encapsulation (Sandbox/isolation/ virtualization)	P8, P11, P14, P17, P34, P36, P43, P48, P50, P59, P66, P68, P69, P70, P71
Continuous integration/deployment	P11, P28, P34, P66, P68, P69, P76
Security by design	P2, P32, P48, P65, P71
Software as a service	P8, P12, P22, P39, P45
Least-privilege design	P17, P31, P45
Defense-in-breadth	P20, P21
Problem Oriented Engineering	P47
Zero-Trust Architecture	P33
Loose-coupling	P22

In Table 4.12 the countermeasures related to software or system design and architecture are listed. Keeping less secure components encapsulated to hinder their errors to propagate is the most referenced design and architecture technique. In many ways, it is similar to least-privilege design, zero-trust architecture, and loose-coupling as they all are related to separating components and not giving more access than necessary.

Table 4.13: Technical countermeasures.

Name	Mentioned in papers
Authentication	P4, P7, P8, P11, P13, P14, P15, P17, P22, P23, P31, P33, P39, P43, P51, P55, P59, P60, P63, P65, P69, P70, P72, P76
Encryption	P2, P4, P6, P7, P8, P11, P13, P15, P20, P21, P22, P31, P33, P36, P39, P45, P48, P50, P55, P64, P65, P71, P72
Firewall	P2, P8, P10, P13, P19, P21, P23, P39, P45, P55, P59, P62, P64, P66, P69
Intrusion Detection System	P7, P12, P19, P23, P37, P39, P48, P59, P64, P69
Versioning/Version control	P11, P14, P17, P20, P29, P34, P45, P68, P76
Reproducible builds	P14, P34, P44, P66
Digital signature	P4, P11, P13, P70
Virtual Private Network	P4, P37
Intrusion Prevention System	P55, P64
Avoiding automatic updates	P17, P66
Honeypot	P48
Single sign-on	P31

Table 4.13 contains the technical countermeasures that can be used to strengthen the software supply chain. Many of these recommendations are general security recommendations that one should apply not only when considering the software supply chain, such as authentication, encryption, and firewall. The more software supply chain-specific countermeasures are Intrusion Detection System (IDS), Intrusion Prevention System (IPS), versioning, and reproducible builds.

Table 4.14: Analysis tools.

Name	Mentioned in papers
Static analysis	P3, P9, P11, P12, P17, P29, P34, P40, P45, P55, P57, P69
Dynamic analysis	P3, P11, P12, P17, P29, P34, P44, P45, P68, P69
Fuzz testing	P9, P45, P59, P65, P69
Binary analysis	P45, P57, P67
Fault injection	P45

Table 4.14 shows the tools used for analyzing code presented in the papers. Static analysis is used for evaluating code without running it while dynamic analysis is when the code is analyzed while running. Fault injection measures the coverage and robustness of tests by inputting errors. Fuzz testing tests the stability of the software product by inserting random data. Binary analysis is an analysis used to evaluate the binary code and can be done as static and dynamic analysis.

Table 4.15: People focused / Manual techniques.

Name	Mentioned in papers
Code review	P9, P11, P32, P34, P35, P51, P54, P57, P69, P70
Increase knowledge	P6, P9, P11, P21, P32, P48, P65
Personal security (2FA, passwords)	P11, P55, P69
Clear ownership and accountability	P2, P24, P72
Software escrow	P4, P70
Clean room	P25

In Table 4.15 we see what techniques can be used by individual programmers or developing teams to help secure their software. Techniques like code reviews and methods for increasing knowledge are often implemented by managers while using two-factor authentication and secure passwords often rely on individual programmers. A software escrow is a three-party agreement between a software developer, an end user, and a source code escrow company. The agreement is to provide insurance to the end user so that if the software provider is unwilling or unable to support the software, the code can still be released to them because of the software escrow company. The clean room technique was presented in P25, where the authors argued that the customer's security team should be allowed into a "clean room" to manually inspect third-party closed-source code.

Table 4.16: Other countermeasure techniques.

Name	Mentioned in papers
Laws, regulations and legislations	P1, P4, P8, P10, P12, P13, P20, P25, P27, P29, P30, P36, P37, P41 P48, P53, P54, P55, P58, P62, P66, P67, P69, P71
Threat modeling	P2, P3, P4, P8, P14, P20, P32, P33, P46, P48, P51, P54, P57, P58, P59, P61, P62, P65, P67, P69, P76
Software bill of material	P33, P62, P66, P73

Table 4.16 contains the countermeasures that did not fit well into any of the other categories. 24 papers discussed some forms of laws, regulations, or legislation. 21 papers mentioned some form of threat modeling or risk assessment. 4 papers considered SBOM a potential countermeasures. SBOMs are different from many of the other countermeasures as they rely on the third-party software provider to do the work and supply and keep the SBOM updated.

To summarise we see that frameworks are the most mentioned countermeasure category and there are many different frameworks discussed. In addition, the papers refer to many different countermeasure techniques that can be utilized by both individual programmers and teams.

4.4 RQ3 - Closed- vs. Open-Source

RQ: *What are the main differences when trusting closed-source instead of open-source in regard to software supply chain attacks?*

To answer this RQ we looked at all the papers that directly or indirectly discuss either open- or closed-source software. We have also looked at the specific papers that discuss the differences between the two types.

Figure 4.5 show the number of times trust in open-source, trust in closed-source, and the differences in trusting them are mentioned. What we see is that open-source is clearly the most discussed, but the total number of times it is mentioned is quite low considering the number of papers analyzed. The papers included in "Differences" are also included in both open- and closed-source so there are 2 papers that only discuss closed-source and 9 papers that only discuss closed-source.

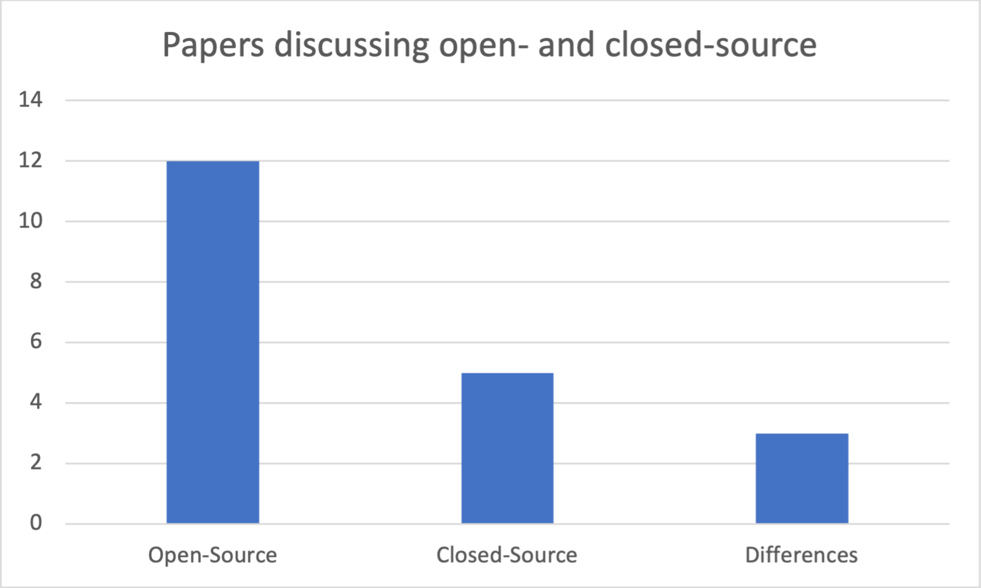


Figure 4.5: The number of papers mentioning open-source, closed-source, and the differences between them.

Table 4.17: Mentions of open- and closed-source software.

In paper	Argument
P25	Concludes that it is only possible to have confidence in the software if access to the source code is provided. If the source code is not provided it is not possible to have confidence in the software, even if thorough testing is done.
P28	Open source software projects often have multiple contributors which can lead to a higher possibility of resolving vulnerabilities involved in the program. However, more contributors could also increase the risk of injecting malicious code into the projects because of unintentional or bad-intentional operations. They also mention that version inheritance is a significant cause of supply chain attacks for open-source software.
P35	With open-source software you can use security features like code reviews and static testing. However, open-source software can still have as many and as severe security issues as Commercial off-the-shelf/Government off-the-shelf (COTS/GOTS) software.

Continued on next page

Table 4.17 – continued from previous page

In paper	Argument
P40	First they argue that free and open source software (FOSS) should security-wise be treated as one's own code. Later they argue that there is no point in discussing if FOSS is more or less secure because first, there may be just no alternative to use FOSS components in a software supply chain because FOSS components are the de-facto standard (e. g., Hadoop for big data). Second, FOSS may offer functionalities that are very expensive to re-implement and, thus, using FOSS is the most economical choice.
P43	Many open-source projects do not provide the information required to verify software's integrity. Projects should provide cryptographic hashes of their software packages and make sure it is not stored in the same distribution channel as the software. When storing hashes in the same place as the software it will be easy to replace the hashes for attackers. Another point mentioned is that open-source software can be more difficult to secure than closed-source because it lives in a more complex environment.
P51	Gives a checklist to evaluate when looking at open-source: 1) Age of the Project 2) Code Releases 3) Project Freshness 4) Contributor Strength 5) Documentation 6) Popularity 7) Open Issues 8) Repository State 9) Automated Code Analysis
P57	Product managers should request COTS vendors provide Assurance Cases for their COTS products detailing both the vendor's secure coding practices and the results of internal static code analysis or third-party assessment (e.g. CC certification). In cases where such information is unavailable, and there is still a desire to use the COTS component, the product managers should consider analyzing the executable using binary code analysis.

Table 4.17 shows short summaries of how the different papers mention open- and closed-source. These include the different aspects of what information open-

source projects should provide to make it possible to evaluate. One of the more interesting findings in the table is P51's checklist of specific evaluation criteria for assessing if third-party software is secure.

4.5 RQ4 - Third party

RQ: *What are the criteria used when deciding which third-party solutions are to be trusted in the supply chain?*

When considering RQ4 and the decisions currently being used to evaluate third-party solutions we decided to look at the specific trust criteria used. We also looked at the mostly discussed package managers that are used to distribute third-party software.

Table 4.18: The trust criteria found in the papers

No.	Trust criteria	P25	P28	P40	P51	P66	P70	P73	P76
1	Make sure the supplier will be liable for any damages caused by its artifacts.						•		
2	Trust components that have been in widespread use for a considerable time and no harmful effects have been observed so far.	•		•	•		•		
3	If the project has a software certification from a certified provider it will strengthen the trust.	•					•		
4	Make sure there is an active maintainer domain.			•	•	•			•
5	Be aware of installation scripts.								•
6	Make sure that the repository is not deprecated and that it does not depend on unmaintained packages.				•	•			•
7	Make sure that there is an adequate number of maintainers and/or contributors.		•		•				•
8	If there are few maintainers make sure that they are not overloaded.								•

Continued on next page

Table 4.18 – continued from previous page

No.	Trust criteria	P25	P28	P40	P51	P66	P70	P73	P76
9	Make sure the team is using a programming language that they are familiar with.			•					
10	How is the maintenance lifecycle of the component, is it being kept up to date?			•	•			•	
11	Make sure there is a trusted author and/or supplier.							•	
12	Is there a hash or signature that can be used to make sure that the software has not been tampered with?							•	
13	Having good documentation. It does not have to be written pages but can also be code examples.				•				
14	Popularity is not something that should be used by itself, but the more popular a library is, the less likely it is to have glaring errors.				•			•	
15	The number of open issues registered on the project can be an indicator of the state of the project. A project can have open issues but the age of the open issues and the responsiveness of the developer on those issues should be evaluated.				•				
16	See if the developers have used automated code analysis to look over the code. This will help with finding known code issues.				•	•			

In Table 4.18 we see that the two most frequently mentioned trust criteria are to "trust components that have been in widespread use over time" and to "make sure there is an active maintainer domain". Other important trust criteria mentioned by multiple papers are to avoid deprecated repositories, make sure there are an adequate number of maintainers, and make sure the component is properly maintained. Out of the papers, we see that P51 mentions the most trust criteria but P40, P73, and P76 also mention quite a few.

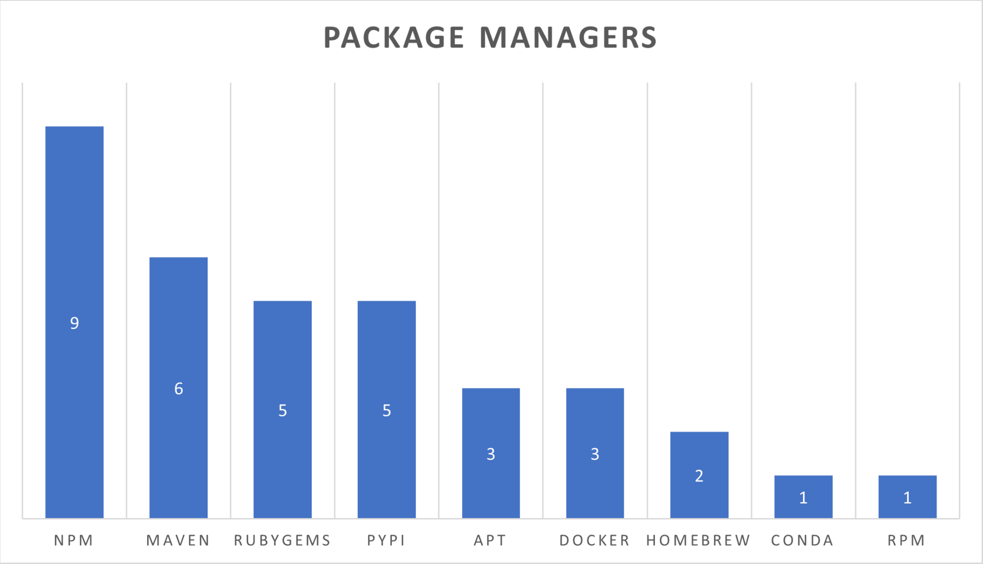


Figure 4.6: The number of papers mentioning different package managers.

When evaluating trust criteria we wanted to explore which package managers were discussed in the papers. This information could be used to gain some insight into what can be evaluated as trust criteria since different package managers display different metadata to the public. In Figure 4.6 we see the package managers most frequently discussed in the papers like npm, Maven, RubyGems, and PyPI.

Table 4.19: Package managers

Name	Mentioned in papers
npm	P3, P11, P17, P28, P34, P44, P68, P69, P76
Maven	P3, P11, P28, P34, P40, P73
RubyGems	P11, P34, P44, P68, P69
PyPI	P11, P34, P44, P68, P69
apt	P11, P14, P28
Docker	P14, P17, P69
Homebrew	P11, P14
CONDA	P14
RPM	P43

In Table 4.19 we see that P11 mentions all the package managers except three. We can also observe that there is an exact match of the papers mentioning RubyGems and PyPI. We also see a tendency where papers either mention npm, RubyGems, and PyPI or the papers mention Maven. The package managers for specific programming languages are mentioned more than package managers for operating systems such as apt and Homebrew.

4.6 RQ5 - Detection Tools

RQ: *What are the tools used to detect vulnerabilities related to supply chain attacks?*

For the last RQ we looked at all the different tools that are mentioned and pointed out their focus areas.

Table 4.20: Specific tools mentioned in papers.

Name	Description	Focus on	Deprecated	In papers
Snyk	Security tool for vulnerabilities in open-source, code, containers, Terraform, and Kubernetes code.	Analyses open-source dependencies, code, containers, Terraform and Kubernetes code.	No	P3, P11, P17, P69, P73, P76
GitHub dependabot	Handles the logic for updating dependencies on GitHub, GitLab, and Azure DevOps.	Third-party dependencies.	No	P3, P11, P28, P55, P66, P73
SonaType	Open-source security monitoring across the software supply chain.	Checks open-source dependencies.	No	P3, P34, P66, P67, P76

Continued on next page

Table 4.20 – continued from previous page

Name	Description	Focus on	Deprecated	In papers
Jenkins/ JenkinsX	A platform for creating Continuous Integration/ Continuous Deployment (CI/CD) environments. JenkinsX is the opinionated version and uses best practices for cloud-focused development.	CI/CD pipelines.	No	P11, P15, P34, P66, P68
In-toto	A framework to protect the integrity of the software supply chain.	Artifacts created by a step cannot be altered before the next step.	No	P14, P55, P66, P69, P73
WhiteSource	Scans open-source repositories, and cross-references this data with open-source components in your build.	Checks open-source dependencies.	Changed name to Mend.	P3, P17, P73
Travis CI	CI platform used to build and test software projects hosted on GitHub and Bitbucket.	CI	No	P34, P66, P68
GitHub Actions	A CI/CD platform to automate build, test, and deployment pipelines.	CI/CD pipelines.	No	P34, P66
Tekton	Cloud-native solution for building CI/CD systems.	CI/CD pipelines.	No	P66, P73
Sigstore	Tool for securely signing software artifacts such as release files, container images, binaries, SBOMs.	Verification for software providers.	No	P14, P66
OpenSSF	Different tools for open-source projects	Open-source projects.	No	P66, P76

Continued on next page

Table 4.20 – continued from previous page

Name	Description	Focus on	Deprecated	In papers
Grafeas	A knowledge-base of the critical metadata organizations need to enforce policies across software development teams and pipelines.	Contains metadata for CI/CD pipelines and containers.	No	P73
SourceClear (SRC:CLR)	Identifies vulnerabilities that are not in the National Vulnerability Database by scanning all open-source repositories' code, metadata, commit logs, bug fixes, patch notes, and other developer comments.	Checks open-source dependencies.	Changed name to Veracode.	P73
CAST Software	To main types: CAST Highlight which provides end-to-end transaction flows, data-call graphs, and architecture blueprints, and CAST Imaging which is for application portfolio analysis.	Gives an overview of the architecture of the application.	No	P73
JFrog	DevOps platform to control and secure the software supply chain.	Provides end-to-end automation and management of binaries and artifacts through the application delivery process.	No	P73
Black Duck	Software composition analysis for open-source and third-party code in applications and containers.	Third-party dependencies in code and containers.	No	P73
Spinnaker	Open-source CD platform for releasing software changes.	CD	No	P73

Continued on next page

Table 4.20 – continued from previous page

Name	Description	Focus on	Deprecated	In papers
Eclipse steady	Analyses Java and Python applications for open-source dependencies with known vulnerabilities, collects evidence regarding the execution of vulnerable code in a given application context (through the combination of static and dynamic analysis techniques), and supports the mitigation of such dependencies.	Third-party Java dependencies in code.	No	P3
npm audit	This is a native tool of npm package manager for scanning npm projects.	Third-party npm dependencies in code.	No	P3
OWASP Dependency check	Software Composition Analysis tool that attempts to detect publicly disclosed vulnerabilities contained within a project's dependencies.	Third-party dependencies.	No	P3
SootDiff	Compare the Java Byte-Code create by different Java compilers.	Third-party Java dependencies.	Yes	P55
CHIRP	Indicator of Compromise tool after the SolarWinds Software Supply Chain attack for the Windows operating system.	Look for signs of compromise after the SolarWinds attack such as the presence of TEARDROP and RAINDROP viruses.	Yes	P55
AppVeyor	Cloud service to automatically build and test your project in a Windows environment in the cloud GitHub, GitLab, and Bitbucket.	CI	No	P34

Continued on next page

Table 4.20 – continued from previous page

Name	Description	Focus on	Deprecated	In papers
greenkeeper	Real-time monitoring for npm dependencies.	Third-party npm dependencies in code.	Yes (part of Snyk)	P17

Table 4.14 lists 24 different tools used to secure the software supply chain. The papers mentioning most tools and their matching count are P73, 10; P66, 9; P3, 7; and P34, 5. We see that the popular tools Snyk and Github Dependabot are the most discussed. Of the 24 tools, 17 of them are mentioned in just one or two papers. Most of the tools focus on open-source software or Continuous Integration/Continuous Deployment (CI/CD). Some of the tools are only working on specific coding collaboration and code-sharing platforms like GitHub, GitLab, and Azure DevOps. There are also two deprecated tools and one tool that has been merged into Snyk.

We have presented the results from the 68 papers used in this SLR. The data was displayed in the different tables and figures with the goal of answering the RQs. We found both expected and unexpected results that will be further discussed and analyzed in chapter 5.

Chapter 5

Discussion

This chapter will be used to discuss the results presented in chapter 4. The discussion is intended to provide insight and our interpretation of the data presented in chapter 4 to answer the RQs found in section 3.1.

From the data gathering and filtration from section 3.2 and section 3.5 we ended up with 68 papers, where 29 of these were considered to be either "Not very relevant" or "Not relevant". We found fewer relevant papers than we expected to find before starting on the SLR. The low number of relevant papers can be an indication that this research field needs further research. We could also have been more strict with quality criteria and the exclusion criteria, however, we wanted to do a "catch-all" approach to make sure we did not miss any relevant categories. From Figure 4.1 the trend is that the number of papers is increasing so it seems like software supply chain security is getting more attention.

The increasing number of published papers observed in Figure 4.1 coincides with reports about an increasing number of software supply chain attacks in recent years [88–90]. We can also see a lot of recent attacks mentioned in the papers in Table 4.6. The recent attack mentioned in section 2.3 is an example showing that this is still happening and it is another indicator that software supply chain attacks are a problem that needs more attention.

Large technology companies such as Google and Microsoft have put efforts into this problem and Google has launched GUAC open-source project to help secure the software supply chain [91, 92]. The increased focus on securing the software supply chain from academia and large technology companies is what we need going forward.

5.1 RQ1 - Attacks

RQ: *What types of supply chain attacks are prominent and discussed in academic literature?*

From the scientific literature, it was hard to grasp the actual number of software supply chain attacks. Many papers mentioned some attacks they thought were relevant, but only briefly. The only event that was discussed in detail was the 2020 SolarWinds Hack, which was presented in P12, P37, and P55. It makes sense that this attack was the one mentioned the most since it was such a significant attack and it could be argued that it rekindled the interest in software supply chain security. The SolarWinds hack is also interesting due to the potential of new attacks that might happen because of this attack since it is likely that hackers have gotten away with critical security information from multiple technology firms.

Many of the attacks in Table 4.6 are old due to one old paper listing many attacks (P43). There are no attacks mentioned between 2002 and 2013, and only one between 2002 and 2017. We found it strange that there were so few mentions of real-world attacks between 2002 and 2017 since 31 of the 68 papers were published between 2017 and 2002. We think it is essential to showcase real-world attacks in scientific papers to emphasize the relevance of the research and the fact that we find such a high number of mentions of real-world attacks in the newer papers shows that it is an ongoing problem.

In Table 4.7 we would have imagined that code injection, typosquatting, and downgrade/roll-back/replay attacks would be more mentioned and higher on the list. These attacks are what we consider more software supply chain specific. There were also no mentions of dependency confusion, which we thought should have been mentioned as this is an attack highly relevant for the software supply chain as it is an attack that tricks a package manager to use a public malicious package instead of a private secure package [93].

Another observation we found to be strange was that the "use after free" attack was only mentioned in one paper (P11). We thought it would be mentioned more since the attack is directly linked to an exploit relevant to the software supply chain. The low number of "use after free" mentions, might be due to a name ambiguity. P11 defines "use after free" as when "An attacker uses the opportunity to reuse the identifier of an existing project, package, or user account withdrawn by its original and legitimate maintainer" [11]. However, CWE defines "use after free" as "Referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code." [94]. There seems that two different phenomena have gotten the same name, and based on online results for "use after free" it would be less confusing if the definition in P11 had a different name, like "abandoned name exploit". We also would have thought cryptojacking to be more mentioned since it is an attack that has been increasing in the last years [95].

Cross-Site Scripting (XSS) was ranked 17^{th} out of the 27 attack methods in Table 4.7 which was unexpected since it is the second highest security threat from CWE's list of Top 25 Most Dangerous Software Weaknesses [96]. Code injection was ranked 8^{th} in Table 4.7, but it is ranked 25^{th} in CWE's list [96].

Another interesting observation from Table 4.7 is that Insider threat, Sabotage, and Social engineering are ranked high, which says something about the importance of the human aspect in software supply chain security. It is often said that humans are the weakest link in cyber security [97], and the high number of mentioned threats can indicate that humans also play an important role in attacks that target the software supply chain.

We also observed that ransomware was discussed in 9 of the papers (P11, P12, P28, P31, P33, P39, P62, P64, P69). However, as we focused on how systems are infiltrated and malware spread instead of focusing on how they damage the system, we did not include these findings in the result. There was also one paper that mentioned spyware (P57), but we did not look more into it as there was only one paper focusing on it.

We think it was unexpected how seldom trojans, viruses, and worms were mentioned in the papers, as we can see in Table 4.8, Table 4.9 and Table 4.10. Researching what kind of malware is used in software supply chain attacks could give more information into how software supply chain attacks have been conducted, and help strengthen the software supply chain. However, the current information does not provide any significant results, since the malware types were mentioned so few times.

Most of the supply chain attacks prominent and discussed in the academic literature are directly related to the events in the 2020 SolarWinds attack, or they are similar to general security attacks. We would have expected to see more focus on what we consider "software supply chain"-specific attacks such as code injection, typosquatting, downgrade/roll-back/replay, use after free/abandoned name exploit, and dependency confusion in the papers.

5.2 RQ2 - Countermeasures

RQ: *What are the current countermeasures used to prevent supply chain attacks?*

In Figure 4.3 we see that there is most focus on frameworks, which is expected since they provide more general information than listing particular countermeasures. Looking at Figure 4.4, it was surprising that there were so many different frameworks mentioned without one or two that really stood out, as we initially thought there would be one industry standard. There are some frameworks that were more referenced such as NIST, CVE, CWE, CAPEC, ISO, and OWASP, but no defining framework that the majority of the papers referenced.

One thing that was expected when looking at Figure 4.3 is that there are multiple countermeasures and techniques to secure the software supply chain. It is a complex problem to solve and many different aspects targeting different parts of the supply chain are needed, as the software supply chain is large and complex.

We can also see that all the countermeasures mentioned in Figure 4.3 cover all the bullet points in the list *Build Trust in Your Software Supply Chain* made by Synopsys, which also has made the Building Security In Maturity Model (BSIMM) [98]. The fact that the countermeasures found in the papers match frameworks strengthens the trustworthiness of the countermeasures.

Frameworks

There were a lot of frameworks mentioned in the papers that can help developers strengthen the software supply chain. This variety in frameworks shows that creating one framework that addresses all the challenges facing supply chain security is a complex and demanding problem. There are multiple sources of recommendations, and developers currently have to examine various frameworks to get the best practices. This result was expected as numerous security organizations are concerned with improving the software supply chain.

P65 had an important statement concerning frameworks: "Frameworks should be adaptable to waterfall development and agile development. It should not be "attacker"-centric, but "actor"-centric as this lets developers think about non-malicious abuses that lead to security weaknesses." [81]. This is similar to what P35 writes about not only being concerned about malware but also weakware. The security risks from unintentional bugs might be as critical as malicious actions from bad actors. The observation that frameworks need to be adaptable to waterfall development and agile development is also notable as companies are working with different development strategies, and frameworks should not hinder the development process. The number of different frameworks could be a consequence of earlier frameworks being made for waterfall development that did not fit the agile workflow.

Looking more into detail on the frameworks in Figure 4.4, we see that NIST is the most popular organization that publishes frameworks. The frameworks that were mentioned in the papers are presented below with the year for the newest version:

- NIST SP 500-287
    - The Second Static Analysis Tool Exposition (2009)
- NIST IR 7622
    - Notional Supply Chain Risk Management Practices for Federal Information Systems (2012)
- NIST 800-30v1
    - Guide for Conducting Risk Assessments (2012)
- NIST SP 800-53
    - Security and Privacy Controls for Information Systems and Organizations (2020)
- NIST SP 800-55
    - Performance Measurement Guide for Information Security (2008)
- NIST SP 800-161
    - Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations (2022)

- NIST SP 800-171
    - Protecting Controlled Unclassified Information in Nonfederal Systems and Organizations (2020)
- NIST SP 800-218
    - Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities (2022)

Based on an initial assessment of the NIST sources, NIST SP 800-161 might be very relevant for people working on software supply chain security. It is, however, over 300 pages long, and the information provided in the framework in NIST SP 800-218 could be as valuable for developers since it is much more concise. Overall, the NIST resources combined provide a lot of information and developers can find different frameworks to help their needs, but all this information might make it hard to navigate and lead one to suffer from information overload.

CVE/CWE/CAPEC were also frequently cited. CVE is used to "Identify, define, and catalog publicly disclosed cybersecurity vulnerabilities." and can be used by developers to look at newly found vulnerabilities and search for specific vulnerabilities among its list of over 180000 vulnerabilities [99]. CWE "is a community-developed list of software and hardware weakness types." [100]. They provide a top 25 list each year of the most dangerous weaknesses and let developers search among their list of over 900 weaknesses. CAPEC provides "a comprehensive dictionary of known patterns of attack employed by adversaries to exploit known weaknesses in cyber-enabled capabilities." and lets developers search for attack patterns in their list of 550 attack patterns [101]. CAPEC also allows developers to view attack patterns by supply chain risks, which helps developers to consider attacks that are specific to Design, Development and Production, Distribution, Acquisition and Deployment, Sustainment, and Disposal. CVE/CWE/CAPEC provides much information when searching in them, but it might be hard for developers to know what to search for and what to prioritize.

The ISO standards were referenced often in the papers. Some of the standards that were mentioned in specific were:

- ISO/IEC/IEEE 12207
    - Systems and software engineering — Software life cycle processes (2017)
- ISO/IEC/IEEE 15288
    - Systems and software engineering — System life cycle processes (2015)
- ISO/IEC 15408
    - Information security, cybersecurity and privacy protection — Evaluation criteria for IT security (2022)
- ISO/IEC/IEEE 15939
    - Systems and software engineering — Measurement process (2017)

- ISO/IEC/IEEE 24765
  - Systems and software engineering — Vocabulary (2017)
- ISO/IEC 27001
  - Information security, cybersecurity and privacy protection — Information security management systems — Requirements (2022)
- ISO/IEC 27002
  - Information security, cybersecurity and privacy protection — Information security controls (2022)
- ISO/IEC 27004
  - Information technology — Security techniques — Information security management — Monitoring, measurement, analysis and evaluation (2016)
- ISO/IEC 27005
  - Information security, cybersecurity and privacy protection — Guidance on managing information security risks (2022)
- ISO/IEC 27034
  - Information technology — Security techniques — Application security (2011)
- ISO/IEC 27036
  - Cybersecurity — Supplier relationships (2021)
- ISO 31000
  - Risk management — Guidelines (2018)

The ISO/IEC 27000 series were referenced the most with ISO/IEC 27001 being the most referenced out of them. One interesting observation was that the ISO/IEC 27003 standard, about Guidance for Information security management systems was not referenced in any of the papers. The ISO standards are a credible source of good security practices and being ISO 27001 certified is a requirement from some contractors. The standards are, however, behind a paywall which might make them less used and relied upon by smaller companies.

OWASP was the 4^{th} most mentioned framework in the papers. There were some parts of OWASP that were mentioned in specific such as the Software Assurance Maturity Model and their top 10 list of web application security risks. It was not mentioned in any of the papers, but OWASP also provides a Security Knowledge Framework, Web Security Testing Guide, and Juice Shop which is a detailed demo of an insecure web application. OWASP also provides a tool for dependence tracking which was mentioned in Table 4.20.

Other relevant frameworks not mentioned as often in the papers were the BSIMM and CC. BSIMM publishes a framework that focuses on the domains: Governance, Intelligence, Software Security Development Lifecycle Touchpoints, and Deployment, and they publish a yearly report with trends and insights informing about the current security trends [102].

CC is used as an international agreement on how information technology can be evaluated on security by licensed laboratories. They publish a set of documents to support the evaluation process and to help define the requirements. It is the same as the ISO/IEC 15408 standard. CC has a list of certified products that can be used as a resource when developers look for trustworthy information technology.

The CSPN framework introduced by the French government was not mentioned in any of the papers but was mentioned when we searched online resources as discussed in section 3.7. The fact that we only looked at papers written in English might have led to missing some interesting frameworks from non-English speaking countries.

The frameworks are one of the tools that can be used by developers to gain knowledge on how to prevent supply chain attacks. There are multiple frameworks with positive and negative attributes, however, all of them help developers, designers, product owners, etc. to narrow down what to focus on when securing the software supply chain.

Design and architecture

There were multiple architectural and design techniques proposed in the papers. From Table 4.12 we see that encapsulation is mentioned the most. This was as expected since encapsulation would help contain potential errors or security risks. Least-privilege design is also similar to encapsulation since it is about restricting unnecessary access.

It was also expected that CI/CD and Software as a Service was frequently mentioned since these techniques are linked to the cloud. There were multiple forms of service mentioned such as software as a service, database as a service, and security as a service. While companies shift towards using the cloud, this shift will increase the number of attack vectors and increase the complexity of securing the software supply chain. This is because during this phase old on-premise threats and new cloud threats will both be relevant [62].

The design and architecture techniques give good pointers to developers on what to consider while creating applications. However, they might not be concrete enough during the development process to provide enough guidance.

Technical

The technical countermeasures were the second most discussed in the papers. As expected, authentication, encryption, and firewalls were the most discussed which makes sense as they are very general countermeasures. However, they are perhaps not that supply-chain-specific. Intrusion detection and prevention, versioning, and reproducible builds are more supply-chain-specific as they are linked to dealing with intrusion and practices for the build process.

One countermeasure that is heavily debated is how to deal with updates. P17 and P66 recommended avoiding automatic updates. This recommendation would help prevent getting compromised if a third-party package gets infected, as it is usually the newest versions that get targeted, but the newest software does have the latest security patches. It is hard to give a best practices recommendation when it comes to automatic updates, but being aware that it has a positive and negative impact is probably what is the most important to consider.

The fact that authentication is the most discussed technical countermeasure in the papers matches CWE's list of most critical software weaknesses as Improper Authentication and Missing Authorization are the 14th and 16th most critical software weaknesses of 2022 in their top 25 list [96].

Overall, the technical countermeasures were as expected in that there were many specific methods mentioned but they were not described in too much detail. Except for avoiding automatic updates, the technical countermeasures were not very controversial.

Analysis tools

The types of analysis tools recommended were static analysis, dynamic analysis, fuzz testing, binary analysis, and fault injection. Static analysis was the most recommended and it can give warnings about common errors, however, many errors might occur when the code is running. Dynamic analysis solves this and was also frequently mentioned in the papers. The dynamic analysis takes some more effort to set up as it needs some running environment to test in.

Binary analysis is important when no access to the source code is available. Binary analysis is harder to perform as the 1's and 0's of binary code give less information than the source code, but it could find some known vulnerable patterns. When a software supplier has closed source code, binary analysis will be the analysis tool that can be used by customers to provide some confidence in the code. The binary analysis will also help if the program gets tampered with during distribution.

Fuzz testing will help reveal unexpected bugs as it is more of a brute-force way of testing with random input to see if the testing can trigger unexpected errors. In 2006 Microsoft reported that between 20 to 25 percent of their security bugs were found via fuzz testing [77]. This has been used for some time and has proven to find errors that developers do not necessarily consider.

Fault injection was the least mentioned and it helps detect errors that occur when a fault happens. It helps detect what happens when the "Happy path" is not taken and gives insight into the robustness of the software.

Tools are important as it is difficult for developers to do a manual assessment of large code bases. Analysis tools help find known patterns. It was expected that analysis tools would be mentioned as a countermeasure as frequently as they were since they provide an extra layer of defense that is relatively easy to implement and use.

People focused / manual techniques

It could be argued that people are the hardest factor to control since threats can come intentionally or unintentionally from insiders and outsiders. Large companies will often change employees regularly which demands huge resources in security education. Therefore, it was expected that there would be recommendations for people-focused countermeasures and that increasing knowledge would be mentioned as one of them.

Code reviews, increasing knowledge, and clear ownership and accountability are all people-focused techniques that will help in the development process. The four-eyes principle, where more than one person should evaluate new dependencies when including them was not mentioned in any of the papers but in the online forums discussed in section 3.7. We would have expected it to have been mentioned in at least one of the papers, however, code reviews are very similar to this principle.

Personal security is relevant to the software supply chain and is linked to the supply chain only being as secure as its weakest link. It is speculated that the SolarWinds hack was made possible by a weak password [23]. The focus on personal security has also led to npm requiring high-impact npm package maintainers to use two-factor authentication (2FA) [103].

Code review, increase knowledge about security, and clear ownership are all relevant to a secure development process. Personal security such as 2FA and passwords are relevant for all employees.

Other countermeasure techniques

The other countermeasures that were different were using SBOM, enforcing laws and regulations, and conducting threat modeling.

SBOM is a list of dependencies a software provider can release to improve transparency in elongated supply chains. It takes some effort to create and maintain, but it would give customers more information if they are affected by a large security vulnerability when it is detected, such as Log4Shell. In BSIMM's yearly report about the current security trends, they recommended that all software firms should "Create a comprehensive SBOM as soon as possible" [102]. The issue with SBOMs is that the providers have to release and update it to increase the customers' security. This could perhaps be recommended by laws or regulations.

Laws and regulations were mentioned in many of the papers. Having strict laws could help shift the focus toward securing the software supply chain. One step in this direction is the Securing Open Source Software Act of 2022 which proposes to create a framework for guiding the US federal government in its use of open-source software. As the number of attacks and focus on supply chain security increases, we will likely see more laws and regulations targeting software supply chain security.

Threat modeling and risk assessments are mentioned in many of the papers and in the frameworks. It was expected to be mentioned as a countermeasure. It could be argued that risk assessment is a crucial requirement for implementing security as one needs to evaluate what the security risks are before one can secure them. It can, however, be hard for developers to assess what the most relevant security risks are since there is a vast amount of different risks and vulnerabilities. This challenge of assessing the most important risks can be aided by frameworks built for risk assessment.

From the papers, we found many countermeasures linked to frameworks, design, architecture, technical, people-focused, manual techniques, and some more special ones. From our evaluation, many of the countermeasures mentioned in the papers are also mentioned in the frameworks. This similarity in recommendations helps to strengthen the credibility of the papers.

5.3 RQ3 - Closed- vs. Open-Source

RQ: *What are the main differences when trusting closed-source instead of open-source in regard to software supply chain attacks?*

There were only 15 papers discussing either open- or closed-source and we initially thought that this would be more discussed as we think these are important factors to consider when evaluating software supply chain. Even though the number of papers was quite small, there were still many interesting perspectives that were discussed.

We see from Figure 4.5 that open-source is mentioned twice as often as closed-source. The reason for this is probably that open-source provides a lot more data, just by being open-source. This data makes it possible to better analyze and discuss the code and gives scientists more information to use when writing papers. It could also be argued that open-source is less secure than closed-source because by having the code open for everyone it is easier for bad actors to find potential weaknesses. This potential danger of open-source code might be why researchers think it is more vulnerable than closed-source code and why they decide to focus on it. This is, however, not necessarily true as "security through obscurity", which is to rely on secrecy to minimize the chance that weaknesses may be detected and targeted, is not recommended for information technology according to NIST. NIST recommends that "System security should not depend on the secrecy of the implementation or its components" [104].

If we look closer at the different arguments from Table 4.17 we see that they cover many different aspects and that some are related:

P25 & P40 Both these papers discuss the trust one can have to open-source software. P25 argues that open-source software is the only one possible to trust and that trusting closed-source is not possible. P40 on the other hand says that there is no point in discussing the trust of open source as we are so dependent on open-source software and that there are no decent alternatives to using open-source code.

P28 This paper points out that there are both positives and negatives related to having multiple contributions. Both arguments are quite strong, and it might indicate that it is important to differentiate the security measures taken by a team based on the number of people contributing.

P35 Here they describe that open-source makes it possible to use analysis tools on the code, which makes it possible to test third-party code before using it in projects. However, one should be careful with the level of trust given to analysis tools as they cannot find all issues.

P43 The argument from P43 is that open-source is hard to distribute in a safe way. They suggest storing a hash of the distributed software at another distribution channel than where the software is stored. This is a classic example of using multiple-layer security.

P51 This paper provides the most evaluating criteria. One interesting observation is that it does not mention anything about the distribution aspect discussed in P43. However, it also mentions contributor strength as P28 does, but it only argues that multiple contributors are a good thing since it would reduce the chance of the project getting abandoned.

P57 The main argument from P57 is that COTS vendors should provide assurance cases for their products. The argument about assurance cases is about transparency and is in some sense similar to SBOMs that is discussed in section 5.2. They also mention, in contrast to what is stated in P25, that since it is still possible to execute binary analysis on closed-source code it is possible to build some trust into closed-source third-party software. We agree with this paper and think the reasoning from P25 is a bit too pessimistic.

Overall we see that most arguments provided in the literature are specific to open-source. This makes sense since it is easier to evaluate as there is more information to use. The only arguments specifically mentioned for closed-source are providing assurance cases and that it is still possible to do some analysis by just having access to the binary code. However, we were surprised that the use of software vendor evaluation was not discussed in more detail as vendor assessment would be an important factor when assessing closed-source software.

5.4 RQ4 - Third party

RQ: *What are the criteria used when deciding which third-party solutions are to be trusted in the supply chain?*

Based on the trust criteria in Table 4.18 the two most frequently mentioned criteria in the papers are to "trust components that have been in widespread use over time" and to "make sure there is an active maintainer domain". Both of these criteria make sense to evaluate as a widely used component is more likely to have uncovered security issues. It is probably also more companies and developers that are willing to help if there are issues with a component that is in widespread use. Having an active maintainer domain is also important since more maintainers can help solve more issues and it is less likely that the software will be abandoned and deprecated.

Further analyzing the trust criteria in Table 4.18 we can see that criteria 1 and 13 are somewhat linked to metadata and documentation as liability should be clarified in a license and licensing can be seen as part of the documentation.

Both criteria 2 and 14 are linked to the positive aspects of popularity. The fact that more users can uncover security issues and other glaring errors is a positive consequence of popularity. What was not discussed in the papers is that popular software usually is more valuable to attack as there are more potential victims.

Criteria 3, 11, and 12 are linked to trusted authors and suppliers. Both criteria 3 and 12 are linked to criteria 11 which recommends looking for trusted authors and suppliers. Trust criteria 3 advocates using certification to strengthen the trust in the software. Criteria 12 promotes using hash and signatures to show if the software has been tampered with after or during distribution. Both criteria 3 and 12 are measures to strengthen criteria 11.

Maintainer evaluation is also relevant to many of the trust criteria in Table 4.18. Criteria 4, 7, 8, and 9 are linked to there being enough maintainers that are active, not overloaded, and familiar with the programming language they use. These are all important to consider, however, we believe that it is hard for users to evaluate the quality of the maintainers. The only exception might be if the project is public on a code-sharing platform such as GitHub where the maintainers and their code commits are displayed.

Trust criteria 6, 10, 15, and 16 are all linked to the health of the repository. Trust criteria 6 advocates avoiding deprecated repositories, 10 cares about the maintenance lifecycle of the component, and 15 urges users to evaluate the number of open issues and for how long they have stayed open. Using components that are being kept up to date and not deprecated should be an indicator that the repository is able to fix potential new security issues. Trust criteria 16 is a bit different than the others since it evaluates if automated code analysis is being used and is more concerned with the quality of the code repository.

Many of the criteria are hard to evaluate if there is no access to the source code or the development process. The fact that most of the criteria from the papers are linked to open-source makes sense as this is what most of the papers focused on.

We also looked at the package managers mentioned in the papers as they might give some insight into what kinds of metadata are provided and possible to use as evaluation criteria. Different package managers provide different metadata that can be used as trust criteria. We think the differences between package managers should be further researched to see what kind of data they should provide and what can be used in the evaluation of the software. We believe that it is important to look at the different package managers as it is through them that much code gets distributed. By making the package managers safe, more software will be secure. It is also important to remember that the information provided by the package managers is the only information many developers use to asses if they trust a software package.

In Table 4.19 we see that there is more focus on programming language package managers, which was expected since the risk from their packages is more tangible since they are related to code. The top four mentioned package managers in the papers are used for JavaScript, Python, Ruby, and Java projects. They were most likely discussed in the papers due to their popularity.

There were many trust criteria suggested in the papers. The focus is primarily on open-source. We believe all of the criteria in Table 4.18 can help assess third-party software. However, we do think that there are many more criteria, not mentioned in the papers, that could be interesting to evaluate.

Many of the criteria we found were linked to open-source software and we agree with P51 that there should be more criteria to assess commercial software source code. We also believe that researching package managers could be an untapped source of knowledge when it comes to trust criteria. They provide different metadata and we believe package managers to be the only place developers look to assess third-party packages.

5.5 RQ5 - Detection Tools

RQ: *What are the tools used to detect vulnerabilities related to supply chain attacks?*

There are a lot of different tools mentioned in the papers and only three of them are deprecated. Having all these resources available for developers can give a lot of valuable defense mechanisms and insight but it also makes it hard for developers to decide on which tools to use. The number of detection tools is also an indication of the complexity of protecting against software attacks and how hard the problem is to solve. It is also a good thing that there are many tools made since it shows that a lot of effort is being put into solving the problem.

12 of the 24 frameworks discussed are focusing on third-party dependencies and security tools for checking software written by others. Out of these tools, many are package manager specific or written for a certain programming language. In addition, many of them are built specifically for open-source projects. The fact that many of the tools are specifically made for certain programming languages could indicate that it is hard to make a general tool for all programming languages. This might be due to the differences in metadata provided by the package managers as discussed in section 5.4.

The second big grouping of tools are the ones focusing on CI/CD. 4 of the tools focus on both CI and CD, while 2 only focus on CI and 1 only focus on CD. Many of these tools are made for the DevOps process, and if they are configured with an added focus on security, they can contribute to a DevSecOps process and a strengthened software supply chain.

The last 5 tools mentioned, that are not related directly to third-party dependencies or CI/CD are In-toto, Sigstore, CAST software, JFrog, and CHIRP:

In-toto is a tool for securing the integrity of the software supply chain, by making it transparent for the user what steps were performed, by whom, and in what order.

Sigstore is a tool that helps with signing and verifying your software, making sure your software does not get tampered with.

CAST software has multiple products and one of them, CAST Imaging, can be used to provide blueprints of the software's inner workings, and automatically detect structural flaws.

JFrog provides multiple platforms that can be used to secure DevOps platforms.

CHIRP is a Python-based tool released by the American Cybersecurity and Infrastructure Security Agency. It can detect post-compromise threat activity after the SolarWinds hack. It was the only tool that can find irregularities after an attack has been executed.

None of the detection tools mentioned focus on closed-source software. It would be useful to have an automatic tool to help with the security evaluation of closed-source projects. This is, however, very challenging, if not impossible to create a perfect tool for this, as the lack of source code and related information will make the assessment no more than a best-effort attempt. Not having access to source code will cause such a tool to only be based on available metadata about the project which does not necessarily reflect the actual code and its quality.

As we can see there are a lot of detection tools available. Many of these tools run automatically which makes them less time-consuming to implement and use. We believe using tools is an important part of securing the software supply chain as it works as an extra layer of security without creating more to do for the developers. The main downside is that the number of tools makes it hard to know which tools to use, but the comparative study of software composition analysis tools in P3 is a contribution to help find the best tool.

5.6 Implications for Research

This section will summarise the discoveries from this SLR that can be implications for research in general and potential future research.

In General

- There is no specific definition of what should be considered part of software supply chain security as the software supply chain itself is not explicitly defined.
- There is a lack of science on closed-source compared to the focus on open-source.

- There are some disagreements about what is best of having few or many maintainers in regard to security.
- There are some disagreements about what is the best practices concerning automatic updates.
- Many of the articles concerning package managers are focusing on only npm, PyPI, Maven, and RubyGems. There are many more programming language package managers that should be evaluated.

Possible Directions For Future Research

- A list of evaluation criteria to assess closed-source.
- A tool to help prevent typosquatting could provide some value for developers.
- A tool that gives a warning if the author of the third-party dependency changes to prevent the "Use after free" or "abandoned name exploit" could contribute to securing the supply chain.
- More research into how package managers' metadata can be used for evaluation criteria.
- Dependency Confusion attacks were not mentioned in any of the papers. Dependency confusion should be researched more.
- Create a tool that will provide a security score to help developers assess if a package is trustworthy before downloading it.

5.7 Implications for Practice

This section will encapsulate the discoveries that have implications for the practices concerning protecting the software supply chain.

- The evaluation criteria from P51 seems promising for evaluating open-source projects.
- There are many competing tools and frameworks that make it hard for developers to evaluate which are best suited for their project, but the work in P3 can give some guidance.
- Software providers should always supply a SBOM with their software as a standard to increase transparency and make it easier to assess dependencies for the consumer.

5.8 Threats to Validity

There are some threats to the validity of the research. We only used two databases of articles: Web of Science and Scopus. They do, however, provide many articles from other databases. We did a manual extraction of much of the information which has some risk of missing some data during the extraction process. The whole process and the extraction process in particular have also been affected by research reflexivity, as our thoughts, knowledge, and behavior will affect the result somewhat. This is probably more likely to occur when the papers assessed did not provide straightforward research questions and results.

A general threat to security research is that it is hard to evaluate what security measures are relevant and actually work. Security recommendations are often believed to work unless they fail which is not always easy to evaluate. We have focused on including recommendations provided by the papers in this SLR rather than assessing all their positive and negative aspects.

We did include some grey literature that was extracted from Web of Science and Scopus, and some other sources that were used to substantiate arguments in chapter 5. Only relying on pair-reviewed published articles would be challenging due to the small number of papers in this research field. We did also look into some online resources as discussed in section 3.7 to see if the community was concerned with the same as researchers, but this did not provide much additional information.

We only did assess English sources. Most of the research used came from western countries and it is not unlikely that other cultures have other thoughts about trust which might be relevant for software supply chain security, or have different frameworks such as CSPN.

Chapter 6

Conclusion

Securing the software supply chain is a complex topic and there are a lot of things to consider for suppliers, customers, and researchers. The goal of this SLR was to summarize information published in the selected literature to answer the research questions. We found that the 2020 SolarWinds hack has affected much of the recent research on software supply chain security. Many papers referenced frameworks instead of specific countermeasures when they discussed countermeasures against software supply chain attacks. We experience that there is more focus on open-source than closed-source in scientific literature. If this is because open-source has more information to assess and write about, or if it is because open-source is more important to consider when securing the software supply chain was not clear from the research, but we expect the first to be the main reason. There were multiple evaluation criteria that can be used to assess third-party software. They were, however, also mostly focused on open-source. We have provided a list of the tools that were mentioned and recommended in the papers to help secure the software supply chain. Overall, we agree with many of the papers that the topic is complex and that there is no silver bullet to stop software supply chain attacks. Dealing with software supply chain security is a problem that needs to be handled by incorporating many different countermeasures. These countermeasures will constantly change, and some of the current measures discussed in the literature are presented in this SLR.

6.0.1 Future Work

After this SLR we have decided to conduct further research into how developers can assess third-party dependencies when they include them in projects. The first step of this process will be to find good trust criteria that can be used to evaluate the security of different third-party dependencies. Here we can use the trust criteria found in P51, the criteria listed in Table 4.18, and we need to look further into what is available on sites such as GitHub, GitLab, and package-specific websites. An example of one such trust criterion that was not explicitly mentioned in the papers is trusting what is already trusted by other developers internally in the company. It is similar to criteria No.2 and No. 14 in Table 4.18 as they are linked to trusting what others have trusted. The trusting internal developer trust criterion is one example of a trust criterion that will be assessed in our future work, but we will look further into more trust criteria.

After identifying the trust criteria we will evaluate and compare existing security evaluation tools. This involves looking at their focus areas, find out what they are lacking, and how they are different. These tools can be found by searching online and from Table 4.20.

The next potential step can be to create a tool that can be used for evaluating different third-party software. This tool will provide a security score that is calculated by using relevant trust criteria and available metadata from different code hosting sites. In addition, we want to query databases like the OSV database and Black Duck to find currently known vulnerabilities in the third-party software.

We have already discussed with a Norwegian software company, Visma, to create the tool in collaboration with them. This can also be combined with making an analyzing tool specific to Visma, that will analyze internal Visma projects and the third-party software already in use to find internal data that can be used as trust criteria such as the trusting internal developer trust criterion. Since Visma is using JavaScript and C# in many of their projects we will look into npm and NuGet. Focusing on npm is similar to what many papers have done as seen in Table 4.19, but focusing on NuGet will be different than the papers in this SLR. This process will then be documented so that it can be reproduced by other companies.

A final step would then be to conduct experiments to assess the tool. Firstly we would do some assessment based on known secure and insecure packages ourselves to see if the tools work as expected. Secondly, we would have some teams test out this tool and do an analysis to assess if the tool is convenient and valuable for developers.

Bibliography

[1] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland and D. Thomas, *Manifesto for agile software development*, 2001. [Online]. Available: `http://www.agilemanifesto.org/`.

[2] D.-L. Vu, F. Massacci, I. Pashchenko, H. Plate and A. 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*, ser. ESEC/FSE 2021, Athens, Greece: Association for Computing Machinery, 2021, pp. 780–792, ISBN: 9781450385626. DOI: `10.1145/3468264.3468592`. [Online]. Available: `https://doi.org/10.1145/3468264.3468592`.

[3] N. Imtiaz, S. Thorn and L. Williams, 'A comparative study of vulnerability reporting by software composition analysis tools,' in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM '21, Bari, Italy: Association for Computing Machinery, 2021, ISBN: 9781450386654. DOI: `10.1145/3475716.3475769`. [Online]. Available: `https://doi.org/10.1145/3475716.3475769`.

[4] R. A. Martin, 'Visibility & control: Addressing supply chain challenges to trustworthy software-enabled things,' in *2020 IEEE Systems Security Symposium (SSS)*, 2020, pp. 1–4. DOI: `10.1109/SSS47320.2020.9174365`.

[5] B. A. Sabbagh and S. Kowalski, 'A socio-technical framework for threat modeling a software supply chain,' *IEEE Security & Privacy*, vol. 13, no. 4, pp. 30–39, 2015. DOI: `10.1109/MSP.2015.72`.

[6] S. Boyson, 'Cyber supply chain risk management: Revolutionizing the strategic control of critical it systems,' *Technovation*, vol. 34, no. 7, pp. 342–353, 2014, Special Issue on Security in the Cyber Supply Chain, ISSN: 0166-4972. DOI: `https://doi.org/10.1016/j.technovation.2014.02.001`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0166497214000194`.

[7] Z. Zhu, K. Lan, Z. Rao and Y. Zhang, 'Risk assessment method for iot software supply chain vulnerabilities,' *Journal of Physics: Conference Series*, vol. 1732, no. 1, p. 012 051, Jan. 2021. DOI: `10.1088/1742-6596/1732/1/012051`. [Online]. Available: `https://dx.doi.org/10.1088/1742-6596/1732/1/012051`.

[8] S. Rothschild, 'Secure software by design,' in *2018 IEEE International Symposium on Technologies for Homeland Security (HST)*, 2018, pp. 1–6. DOI: `10.1109/THS.2018.8574188`.

[9] S. D. Alexander, 'Trust engineering: Rejecting the tyranny of the weakest link,' in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 145–148.

[10] S. Raponi, M. Caprolu and R. Di Pietro, 'Beyond solarwinds: The systemic risks of critical infrastructures, state of play, and future directions,' in *ITASEC '21: Italian Conference on Cyber Security*, 2021. [Online]. Available: `http://star.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-2940/paper33.pdf`.

[11] M. Ohm, H. Plate, A. Sykosch and M. Meier, 'Backstabber's knife collection: A review of open source software supply chain attacks,' *CoRR*, vol. abs/2005.09535, 2020. arXiv: `2005.09535`. [Online]. Available: `https://arxiv.org/abs/2005.09535`.

[12] G. Ferreira, L. Jia, J. Sunshine and C. Kästner, 'Containing malicious package updates in npm with a lightweight permission system,' in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1334–1346. DOI: `10.1109/ICSE43902.2021.00121`.

[13] A. Esfahani, G. Mantas, J. Ribeiro, J. Bastos, S. Mumtaz, M. A. Violas, A. M. De Oliveira Duarte and J. Rodriguez, 'An efficient web authentication mechanism preventing man-in-the-middle attacks in industry 4.0 supply chain,' *IEEE Access*, vol. 7, pp. 58 981–58 989, 2019. DOI: `10.1109/ACCESS.2019.2914454`.

[14] O. Akinrolabu, S. New and A. Martin, 'Assessing the security risks of multicloud saas applications: A real-world case study,' in *2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/ 2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, 2019, pp. 81–88. DOI: `10.1109/CSCloud/EdgeCom.2019.00-14`.

[15] Snyk, *Software security in 2021 | Definition, Issues & Types*, Sep. 2022. [Online]. Available: `https://snyk.io/learn/software-security/`.

[16] 'Cost of a Data Breach Report 2022.' (2022), [Online]. Available: `https://www.ibm.com/reports/data-breach` (visited on 15/11/2022).

[17] European Commission, *Art. 82 GDPR – Right to compensation and liability*, Mar. 2018. [Online]. Available: `https://gdpr-info.eu/art-82-gdpr/`.

[18] W. Axelrod, ‘Malware, "weakware," and the security of software supply chains,’ *CrossTalk, The Journal of Defense Software Engineering*, vol. 27, pp. 20–24, Mar. 2014.

[19] J. D. Linton, S. Boyson and J. Aje, ‘The challenge of cyber supply chain security to research and practice – an introduction,’ *Technovation*, vol. 34, no. 7, pp. 339–341, 2014, Special Issue on Security in the Cyber Supply Chain, ISSN: 0166-4972. DOI: `https://doi.org/10.1016/j.technovation.2014.05.001`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0166497214000522`.

[20] Berger, *What is Log4Shell? The Log4j vulnerability explained (and what to do about it)*, Dec. 2017. [Online]. Available: `https://www.dynatrace.com/news/blog/what-is-log4shell/`.

[21] The Apache Software Foundation, *Log4j – Apache Log4j Security Vulnerabilities*, Sep. 2022. [Online]. Available: `https://logging.apache.org/log4j/2.x/security.html`.

[22] ‘Open source vs closed source software.’ (2022), [Online]. Available: `https://www.ibm.com/topics/open-source` (visited on 16/11/2022).

[23] J. Yang, Y. Lee and A. McDonald, ‘Solarwinds software supply chain security: Better protection with enforced policies and technologies,’ in Jan. 2022, pp. 43–58, ISBN: 978-3-030-92316-7. DOI: `10.1007/978-3-030-92317-4_4`.

[24] L. Sterle and S. Bhunia, ‘On solarwinds orion platform security breach,’ in *2021 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/IOP/SCI)*, 2021, pp. 636–641. DOI: `10.1109/SWC50871.2021.00094`.

[25] R. Patnayakuni and N. Patnayakuni, ‘Information security in value chains: A governance perspective,’ in *AMCIS*, 2014.

[26] N. Kshetri and J. Voas, ‘Supply chain trust,’ *IT Professional*, vol. 21, no. 2, pp. 6–10, 2019. DOI: `10.1109/MITP.2019.2895423`.

[27] X. Wang, ‘On the feasibility of detecting software supply chain attacks,’ in *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*, 2021, pp. 458–463. DOI: `10.1109/MILCOM52596.2021.9652901`.

[28] L. Courtès, ‘Building a secure software supply chain with GNU guix,’ *The Art, Science, and Engineering of Programming*, vol. 7, no. 1, Jun. 2022. DOI: `10.22152/programming-journal.org/2023/7/1`. [Online]. Available: `https://doi.org/10.22152%5C%2Fprogramming-journal.org%5C%2F2023%5C%2F7%5C%2F1`.

[29] E. Levy, ‘Poisoning the software supply chain,’ *IEEE Security & Privacy*, vol. 1, no. 3, pp. 70–73, 2003. DOI: `10.1109/MSECP.2003.1203227`.

[30] S. Du, T. Lu, L. Zhao, B. Xu, X. Guo and H. Yang, 'Towards an analysis of software supply chain risk management,' in *Proceedings of the World Congress on Engineering and Computer Science*, vol. 1, 2013.

[31] Montalbano, *Supply Chain Attack Pushes Out Malware to More than 250 Media Websites*, Nov. 2022. [Online]. Available: `https://www.darkreading.com/application-security/supply-chain-attack-pushes-out-malware-to-more-than-250-media-websites`.

[32] Paganini, *250+ U.S. news sites spotted spreading FakeUpdates malware in a supply-chain attack*, Nov. 2022. [Online]. Available: `https://securityaffairs.co/wordpress/138052/cyber-crime/supply-chain-attack-fakeupdates.html`.

[33] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila and L. Williams, 'What are weak links in the npm supply chain?' In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022, pp. 331–340. DOI: `10.1145/3510457.3513044`.

[34] Flynn, *16 Agile Statistics [2022]: What You Need To Know About Agile Project Management – Zippia*, May 2022. [Online]. Available: `https://www.zippia.com/advice/agile-statistics/`.

[35] M. N. Abd Latif, N. A. Abd Aziz, N. S. Nik Hussin and Z. Abdul Aziz, 'Cyber security in supply chain management: A systematic review,' *Logforum*, vol. 17, pp. 49–57, Mar. 2021. DOI: `10.17270/J.LOG.2021555`.

[36] Y. Zhao, R. Liang, X. Chen and J. Zou, 'Evaluation indicators for open-source software: A review,' *Cybersecurity*, vol. 4, no. 1, p. 20, Jun. 2021, ISSN: 2523-3246. DOI: `10.1186/s42400-021-00084-8`. [Online]. Available: `https://doi.org/10.1186/s42400-021-00084-8`.

[37] A. Ghadge, M. Weiss, N. D. Caldwell and R. Wilding, 'Managing cyber risk in supply chains: A review and research agenda,' *Supply Chain Management: An International Journal*, 2019.

[38] B. Kitchenham and S. Charters, 'Guidelines for performing systematic literature reviews in software engineering,' vol. 2, Jan. 2007.

[39] Satyabrata Jena. 'Difference between Open Source Software and Closed Source Software.' (2022), [Online]. Available: `https://www.geeksforgeeks.org/difference-between-open-source-software-and-closed-source-software/` (visited on 31/10/2022).

[40] 'Third Party Software Components definition.' (2022), [Online]. Available: `https://www.lawinsider.com/dictionary/third-party-software-components` (visited on 31/10/2022).

[41] 'Scopus Search Guide.' (2022), [Online]. Available: `http://schema.elsevier.com/dtds/document/bkapi/search/SCOPUSSearchTips.htm` (visited on 31/10/2022).

[42] 'Web of Science Core Collection Help.' (2020), [Online]. Available: https://images.webofknowledge.com/images/help/WOS/hs_search_operators.html (visited on 31/10/2022).

[43] EndNote, *EndNote | The best reference management tool*, Aug. 2022. [Online]. Available: https://endnote.com/.

[44] Gilles 'SO- stop being evil', *Security evaluation: what is there apart from common criteria?* Accessed on 2022-11-14., Dec. 2011. [Online]. Available: https://security.stackexchange.com/questions/9512/security-evaluation-what-is-there-apart-from-common-criteria/9542.

[45] user3240316, *Basically only 2 ways of npm supply chain attacks?* Accessed on 2022-11-14., Sep. 2022. [Online]. Available: https://security.stackexchange.com/questions/264722/basically-only-2-ways-of-npm-supply-chain-attacks.

[46] Y. Nakano, T. Nakamura, Y. Kobayashi, T. Ozu, M. Ishizaka, M. Hashimoto, H. Yokoyama, Y. Miyake and S. Kiyomoto, 'Automatic security inspection framework for trustworthy supply chain,' in *2021 IEEE/ACIS 19th International Conference on Software Engineering Research, Management and Applications (SERA)*, 2021, pp. 45–50. DOI: 10.1109/SERA51205.2021.9509040.

[47] N. Imtiaz, S. Thorn and L. Williams, 'A comparative study of vulnerability reporting by software composition analysis tools,' in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ser. ESEM '21, Bari, Italy: Association for Computing Machinery, 2021, ISBN: 9781450386654. DOI: 10.1145/3475716.3475769. [Online]. Available: https://doi.org/10.1145/3475716.3475769.

[48] R. A. Martin, 'Assurance for cyberphysical systems: Addressing supply chain challenges to trustworthy software-enabled things,' in *2020 IEEE Systems Security Symposium (SSS)*, 2020, pp. 1–5. DOI: 10.1109/SSS47320.2020.9174201.

[49] C. W. Axelrod, 'Assuring software and hardware security and integrity throughout the supply chain,' in *2011 IEEE International Conference on Technologies for Homeland Security (HST)*, 2011, pp. 62–68. DOI: 10.1109/THS.2011.6107848.

[50] U. Agarwal, V. Rishiwal, S. Tanwar, R. Chaudhary, G. Sharma, P. N. Bokoro and R. Sharma, 'Blockchain technology for secure supply chain management: A comprehensive review,' *IEEE Access*, vol. 10, pp. 85 493–85 517, 2022. DOI: 10.1109/ACCESS.2022.3194319.

[51] G. A. Vazquez-Martinez, J. Gonzalez-Compean, V. J. Sosa-Sosa, M. Morales-Sandoval and J. C. Perez, 'Cloudchain: A novel distribution model for digital products based on supply chain principles,' *International Journal of Information Management*, vol. 39, pp. 90–103, 2018, ISSN: 0268-4012. DOI: https://doi.org/10.1016/j.ijinfomgt.2017.12.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0268401217306357.

[52] E.W.T.Ngai, T.C.E.Cheng and S.S.M.Ho, 'Critical success factors of web-based supply-chain management systems: An exploratory study,' *Production Planning & Control*, vol. 15, no. 6, pp. 622–630, 2004. DOI: 10.1080/09537280412331283928. eprint: https://doi.org/10.1080/09537280412331283928. [Online]. Available: https://doi.org/10.1080/09537280412331283928.

[53] N. Bartol, 'Cyber supply chain security practices dna – filling in the puzzle using a diverse set of disciplines,' *Technovation*, vol. 34, no. 7, pp. 354–361, 2014, Special Issue on Security in the Cyber Supply Chain, ISSN: 0166-4972. DOI: https://doi.org/10.1016/j.technovation.2014.01.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166497214000066.

[54] K. W. Hamlen and B. Thuraisingham, 'Data security services, solutions and standards for outsourcing,' *Computer Standards & Interfaces*, vol. 35, no. 1, pp. 1–5, 2013, ISSN: 0920-5489. DOI: https://doi.org/10.1016/j.csi.2012.02.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920548912000414.

[55] A. Raicu and G. Raicu, 'Digital enterprise and cyber security evolution,' *Macromolecular Symposia*, vol. 396, no. 1, p. 2 000 326, 2021. DOI: https://doi.org/10.1002/masy.202000326. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/masy.202000326. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/masy.202000326.

[56] R. Stempfley, 'Each person and organization in the supply chain path "touches," or has influence on, the security and resilience of software used to control products, systems, and services,' *CrossTalk, The Journal of Defense Software Engineering*, vol. 26, no. 2, 2013. eprint: https://apps.dtic.mil/sti/pdfs/ADA576064.pdf. [Online]. Available: https://apps.dtic.mil/sti/pdfs/ADA576064.pdf.

[57] A. S. Markov and I. A. Sheremet, 'Enhancement of confidence in software in the context of international security,' in *CEUR Workshop Proceedings*, vol. 2603, 2019, pp. 88–92.

[58] E. Wotring and S. Migues, 'Ensuring software assurance process maturity,' *CrossTalk, The Journal of Defense Software Engineering*, vol. 24, no. 2, 2011. eprint: https://apps.dtic.mil/sti/pdfs/ADA538093.pdf. [Online]. Available: https://apps.dtic.mil/sti/pdfs/ADA538093.pdf.

[59] M. B. Chrissis, M. Konrad and M. Moss, 'Ensuring your development processes meet today's cyber challenges,' *CrossTalk, The Journal of Defense Software Engineering*, vol. 29, 2013.

[60] D. Yan, Y. Niu, K. Liu, Z. Liu, Z. Liu and T. F. Bissyandé, 'Estimating the attack surface from residual vulnerabilities in open source software supply chain,' in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021, pp. 493–502. DOI: 10.1109/QRS54544.2021.00060.

[61] S. Shankles, M. Moss, J. Pickel and N. Bartol, 'How international standard efforts help address challenges in today's global ict marketplace,' vol. 26, pp. 10–15, Mar. 2013.

[62] B. Hensley, 'Identity is the new perimeter in the fight against supply chain attacks,' *Network Security*, vol. 2021, no. 7, pp. 7–9, 2021, ISSN: 1353-4858. DOI: https://doi.org/10.1016/S1353-4858(21)00074-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S135348582100074X.

[63] S. Lawrence Pfleeger, M. Libicki and M. Webber, 'I'll buy that! cybersecurity in the internet marketplace,' *IEEE Security & Privacy*, vol. 5, no. 3, pp. 25–31, 2007. DOI: 10.1109/MSP.2007.64.

[64] T. M. S. do Amaral and J. J. C. Gondim, 'Integrating zero trust in the cyber supply chain security,' in *2021 Workshop on Communication Networks and Power Systems (WCNPS)*, 2021, pp. 1–6. DOI: 10.1109/WCNPS53648.2021.9626299.

[65] M. Windelberg, 'Objectives for managing cyber supply chain risk,' *International Journal of Critical Infrastructure Protection*, vol. 12, pp. 4–11, 2016, ISSN: 1874-5482. DOI: https://doi.org/10.1016/j.ijcip.2015.11.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1874548215000785.

[66] S. Dashevskyi, A. D. Brucker and F. Massacci, 'On the security cost of using a free and open source component in a proprietary product,' in *Engineering Secure Software and Systems*, J. Caballero, E. Bodden and E. Athanasopoulos, Eds., Cham: Springer International Publishing, 2016, pp. 190–206, ISBN: 978-3-319-30806-7.

[67] A. Szakal and K. Pearsall, 'Open industry standards for mitigating risks to global supply chains,' *IBM Journal of Research and Development*, vol. 58, pp. 11–113, Jan. 2014. DOI: 10.1147/JRD.2013.2285605.

[68] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate and A. Sabetta, 'Poster: Towards using source code repositories to identify software supply chain attacks,' Nov. 2020. DOI: 10.1145/3372297.3420015.

[69] F. Xie, T. Lu, B. Xu, D. Chen and Y. Peng, 'Research on software development process assurance models in ict supply chain risk management,' in *2012 IEEE Asia-Pacific Services Computing Conference*, 2012, pp. 43–49. DOI: 10.1109/APSCC.2012.41.

[70] M. O'Halloran, J. G. Hall and L. Rapanotti, 'Safety engineering with cots components,' *Reliability Engineering & System Safety*, vol. 160, pp. 54–66, 2017, ISSN: 0951-8320. DOI: https://doi.org/10.1016/j.ress.2016.11.016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0951832016308298.

[71] C. Gottlieb, 'Securing high-tech goods across the supply chain,' *EDN*, Sep. 2010.

[72] W. Scacchi and T. A. Alspaugh, 'Securing software ecosystem architectures: Challenges and opportunities,' *IEEE Software*, vol. 36, no. 3, pp. 33–38, 2019. DOI: 10.1109/MS.2018.2874574.

[73] V. Mills and S. Butakov, 'Security evaluation criteria of open-source libraries,' in *Computational Science and Its Applications – ICCSA 2022 Workshops: Malaga, Spain, July 4–7, 2022, Proceedings, Part V*, Malaga, Spain: Springer-Verlag, 2022, pp. 422–435, ISBN: 978-3-031-10547-0. DOI: 10.1007/978-3-031-10548-7_31. [Online]. Available: https://doi.org/10.1007/978-3-031-10548-7_31.

[74] R. Mu and F. Yonggang, 'Security in the cyber supply chain: A chinese perspective,' *Technovation*, vol. 34, Jul. 2014. DOI: 10.1016/j.technovation.2014.02.004.

[75] W. Axelrod, 'Software security assurance soup to nuts,' *CrossTalk, The Journal of Defense Software Engineering*, vol. 28, pp. 37–43, Jan. 2015.

[76] P. R. Croll, 'Supply chain risk management - understanding vulnerabilities in code you buy, build, or integrate,' in *2011 IEEE International Systems Conference*, 2011, pp. 194–200. DOI: 10.1109/SYSCON.2011.5929123.

[77] R. J. Ellison and C. Woody, 'Supply-chain risk management: Incorporating security into software development,' in *2010 43rd Hawaii International Conference on System Sciences*, 2010, pp. 1–10. DOI: 10.1109/HICSS.2010.355.

[78] D. Forte, R. Perez, Y. Kim and S. Bhunia, 'Supply-chain security for cyberinfrastructure [guest editors' introduction],' *Computer*, vol. 49, pp. 12–16, Aug. 2016. DOI: 10.1109/MC.2016.260.

[79] A. Wirth, 'Cyberinsights: Talking about the Software Supply Chain,' *Biomedical Instrumentation & Technology*, vol. 54, no. 5, pp. 364–367, Oct. 2020, ISSN: 0899-8205. DOI: 10.2345/0899-8205-54.5.364. eprint: https://meridian.allenpress.com/bit/article-pdf/54/5/364/2622296/i0899-8205-54-5-364.pdf. [Online]. Available: https://doi.org/10.2345/0899-8205-54.5.364.

[80] D. Deyannis, E. Papadogiannaki, G. Chrysos, K. Georgopoulos and S. Ioannidis, 'The diversification and enhancement of an ids scheme for the cybersecurity needs of modern supply chains,' *Electronics*, vol. 11, no. 13, 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11131944. [Online]. Available: https://www.mdpi.com/2079-9292/11/13/1944.

[81] J. Whitmore, S. Türpe, S. Triller, A. Poller and C. Carlson, 'Threat analysis in the software development lifecycle,' *IBM Journal of Research and Development*, vol. 58, no. 1, 6:1–6:13, 2014. DOI: 10.1147/JRD.2013.2288060.

[82] W. Enck and L. Williams, 'Top five challenges in software supply chain security: Observations from 30 industry and government organizations,' *IEEE Security & Privacy*, vol. 20, no. 2, pp. 96–100, 2022. DOI: 10.1109/MSEC.2022.3142338.

[83] M. Ohm, A. Sykosch and M. Meier, 'Towards detection of software supply chain attacks by forensic artifacts,' in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ser. ARES '20, Virtual Event, Ireland: Association for Computing Machinery, 2020, ISBN: 9781450388337. DOI: 10.1145/3407023.3409183. [Online]. Available: https://doi.org/10.1145/3407023.3409183.

[84] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio and W. Lee, 'Towards measuring supply chain attacks on package managers for interpreted languages,' *arXiv preprint arXiv:2002.01139*, 2020.

[85] M. Naedele and T. E. Koch, 'Trust and tamper-proof software delivery,' ser. SESS '06, Shanghai, China: Association for Computing Machinery, 2006, pp. 51–58, ISBN: 1595934111. DOI: 10.1145/1137627.1137636. [Online]. Available: https://doi.org/10.1145/1137627.1137636.

[86] B. R. Ray, J. Abawajy, M. Chowdhury and A. Alelaiwi, 'Universal and secure object ownership transfer protocol for the internet of things,' *Future Generation Computer Systems*, vol. 78, pp. 838–849, 2018, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2017.02.020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17302182.

[87] S. Shankles, 'We cannot blindly reap the benefits of a globalized ict supply chain!' *CrossTalk*, p. 5, 2013.

[88] National Institute of Standards and Technology, *Software Supply Chain Attacks*, 2018. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Supply-Chain-Risk-Management/documents/ssca/2017-winter/NCSC_Placemat.pdf.

[89] O. E., *Software Supply Chain Attacks: 2021 in Review*, Jun. 2022. [Online]. Available: https://blog.aquasec.com/software-supply-chain-attacks-2021.

[90] D. V., *Software supply chain attacks tripled in 2021 says Argon*, Jan. 2022. [Online]. Available: `https://cybermagazine.com/cyber-security/software-supply-chain-attacks-tripled-2021-says-argon`.

[91] L. R., *Google Launches GUAC Open Source Project to Secure Software Supply Chain*, Oct. 2022. [Online]. Available: `https://thehackernews.com/2022/10/google-launches-guac-open-source.html`.

[92] Microsoft, *Supply chain attacks*, Nov. 2022. [Online]. Available: `https://learn.microsoft.com/en-us/microsoft-365/security/intelligence/supply-chain-malware?view=o365-worldwide`.

[93] B. A., *Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies*, Dec. 2021. [Online]. Available: `https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610`.

[94] Common Weakness Enumeration, *CWE - CWE-416: Use After Free (4.9)*, Oct. 2022. [Online]. Available: `https://cwe.mitre.org/data/definitions/416.html`.

[95] Kondratyev and Ivanov, *The state of cryptojacking in the first three quarters of 2022*, Nov. 2022. [Online]. Available: `https://securelist.com/cryptojacking-report-2022/107898/`.

[96] Common Weakness Enumeration, *CWE - 2022 CWE Top 25 Most Dangerous Software Weaknesses*, 2022. [Online]. Available: `https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html`.

[97] J. Jalkanen, 'Is human the weakest link in information security?: Systematic literature review,' 2019. [Online]. Available: `https://jyx.jyu.fi/handle/123456789/64186`.

[98] Synopsys, *What Is Software Supply Chain Security and How Does It Work? | Synopsys*, 2022. [Online]. Available: `https://www.synopsys.com/glossary/what-is-software-supply-chain-security.html`.

[99] Common Vulnerabilities and Exposures, *CVE - CVE*, Jul. 2022. [Online]. Available: `https://cve.mitre.org/`.

[100] Common Weakness Enumeration, *CWE - CWE*, Oct. 2022. [Online]. Available: `https://cwe.mitre.org/data/index.html`.

[101] Common Attack Pattern Enumeration and Classification, *CAPEC - Common Attack Pattern Enumeration and Classification (CAPEC™)*, Sep. 2022. [Online]. Available: `https://capec.mitre.org/index.html`.

[102] BSIMM, *BSIMM 13 TRENDS & INSIGHTS REPORT 2022*, Sep. 2022. [Online]. Available: `https://www.bsimm.com/content/dam/bsimm/reports/bsimm13.pdf`.

[103] GitHub, *High-impact package maintainers now require 2FA | GitHub Changelog*, Nov. 2022. [Online]. Available: `https://github.blog/changelog/2022-11-01-high-impact-package-maintainers-now-require-2fa/`.

[104] Scarfone, Jansen and Tracy, *Guide to General Server Security*, Jul. 2008. [Online]. Available: `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-123.pdf`.

NTNU

Norwegian University of
Science and Technology