Motivating Version Range Adoption in Maven Through Quantified Trust

Gijs Hoedemaker

Supervisor(s): Sebastian Proksch, Cathrine Paulsen

EEMCS, Delft University of Technology, The Netherlands

Name of the student: Gijs Hoedemaker
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Cathrine Paulsen, Georgios Iosifidis

Abstract

As Open-Source projects grow in size and incorporate more and more external dependencies, developers increasingly rely on dependency managers such as Maven to manage version conflicts and automate dependency resolution. However, developers are often unaware of vulnerabilities in their dependencies or do not prioritise security due to the effort required to update dependencies, resulting from a lack of compliance with Semantic Versioning policies. This paper aims to motivate greater adoption of version ranges by empirically analysing a large selection of open-source Maven projects to determine the impact of fixed versions compared to version ranges and identify opportunities for safe updates. It also proposes TeSTer as a conceptual command-line tool that analyse dependencies and provides insight into security practices, release frequency, and `SemVer` compliance to support informed decisions when incorporating dependencies in a project.

1 Introduction

The use of external libraries or packages as building blocks in Java projects, often referred to as *dependencies*, is an integral part of project development. When projects mature and grow larger in size and complexity, it becomes harder and harder to maintain a large selection of dependencies while choosing the correct version for each dependency without running into compatibility issues.

These issues led to the rise of dependency management software, such as NPM for Node.js[1], Cargo for Rust[2], or Maven for Java[3], which allow developers to efficiently declare dependencies with their desired versions, as well as plugins to aid further in managing these dependencies. These dependency managers will automatically download and resolve all selected versions and report any version conflicts or unresolved versions. In the case of Maven specifically, it will select a version that is as close as possible to the declared version. [2]

To facilitate maintainers in deciding which versions to declare, and providing developers with guidelines on how to publish their releases, GitHub co-founder Tom Preston-Werner introduced the Semantic Versioning policy (`semver`) in 2013[4]. `Semver` denotes versions using a Major, Minor, and Patch number. Increments to the major version number indicate the introduction of backward incompatible changes, while updates that change only the minor of patch version numbers are not supposed to introduce any breaking changes.

One of the largest issues that arises from improper dependency management practices is the continued use of vulnerable dependency versions despite the fact that fixes are released promptly after discovery [14]. This issue arises in part because developers are unaware of these vulnerabilities [6][8][12], as well as security not being a high priority or taking too much effort to fix due to backward incompatible changes [4].

An seemingly obvious solution to this problem is the use of *version ranges*. These ranges allow developers to tell the dependency manager to select a version anywhere between two versions, or any version newer than a certain version. This, in combination with Semantic Versioning should compel developers to declare ranges allowing patch updates for dependencies to be automatically adopted. However, Raemakers et al. found that nearly a third of minor releases and a quarter of patch releases contain backward-incompatible changes, leading developers to prefer to use fixed versions over version ranges [9].

Compared to other dependency managers that feature version ranges, the adoption of version ranges in the Maven ecosystem is especially low [5]. Tools have been developed that demonstrate the utility of version ranges in resolving vulnerabilities [14], but the usage of version ranges is still only slowly increasing [1], with 81.5% of projects still including outdated dependencies [8].

The notion of trust has been shown to be an important factor for developers to choose for the adoption of version ranges for specific dependencies [7], but there is limited research into what defines the trustworthiness of a dependency. The "wisdom of the crowds" principle has been shown to provide motivation for developers to employ version ranges for popular dependencies [1].

OpenSSF's Scorecard[5] analyses a project's security hygiene [10][11], but does not reflect a project's maintenance or semantic compatibility between versions. Another factor that was found to be relevant for the stability and security of a dependency is the Mean Time To Update (MTTU) [13]. Together, we hypothesized that these factors could provide a solid foundation for motivating developers to choose whether to trust a dependency.

This paper intends to fill the research gap introduced in this section, as well as creating a concrete representation of the trustworthiness of dependencies by answering the following research questions:

1. **What is the impact of using fixed versions as opposed to version ranges?**
   Motivation: by investigating a large set of open-source projects for the amount of declared dependency versions which are outdated but can be safely updated without compatibility issues, we can provide a representative picture of the underutilisation of version ranges in Maven projects to inform and motivate developers, as well as provide further motivation for our second research question.

2. **How can developers know if a dependency can be trusted to adhere to semantic versioning?**
   Motivation: other than popularity and community opinion, dependencies have few indicators that suggest whether or not they comply with `semver` policies. By

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

[2]https://doc.rust-lang.org/cargo/

[3]https://maven.apache.org/

[4]https://www.semver.org

[5]https://openssf.org/projects/scorecard/

compiling a trust score metric based on several heuristics of a dependency, developers may be persuaded to use version ranges for dependencies with a high score.

To answer these research questions, we first conduct a large-scale empirical analysis of a large variety of open-source Maven projects to establish the underutilisation of safe, automatic dependency updates through the use of version ranges. Second, we propose a Trust Score Tool, TeSTer, which serves as a conceptual decision-making tool for developers by analysing security hygiene, release frequency, and semantic stability for a clear representation of heuristics. This allows developers to make conscious, risk-aware decisions on whether and how to adopt a dependency without having to rely on automated tooling.

The remainder of the paper is structured as follows. Section 2 explains how our research builds on previous work. Section 3 explains the methodology used to gather the data set used and the strategies for analysing this data and answering the research questions. Section 4 presents the results obtained from this analysis and 5 discusses the findings, highlights any moral and ethical considerations about the research, and reasons about potential future research.. Finally, Section 6 concludes the paper.

2 Related Work

A study by Zhang et al. found that out of 82 million declared dependency relationships, only 1.02% used semantic version ranges [14]. 98.94% of these ranges resolved to patched versions of once-vulnerable dependencies. These numbers show that the version ranges are heavily undervalued. Dietrich et al. support this statement by showing that version ranges are used quite frequently in other dependency managers such as Packagist[6] or NuGet [7], but barely in Maven [5]. This intrigued us to further research how to boost the utilisation of version ranges in Maven specifically.

A tool that provides a large contribution to this goal is Ranger, developed by Zhang et al. [14]. This tool can analyse declared dependencies and rewrite the declared version to a range of versions which are backward-compatible, providing secure version ranges which, by evaluation, remediated 90.32% of vulnerable (transitive) dependencies. However, this research only focused on dependencies known to have vulnerabilities according to the Common Vulnerabilities and Exposures (CVE) advisory[8]. We wished to extend on this research by investigating a broader and more general-purpose dataset to further motivate developers to employ version ranges not only for security reasons, but also for utility and convenience.

Decan et al. investigated the rate of developers using version ranges over time and found a slowly increasing trend of the rate of utilisation of version ranges. They state that developers follow the ”wisdom of the crowds” principle, suggesting that developers are more likely to declare a version range for a dependency if other developers in their community do the same [1]. This research suggests that developers can be motivated to use version ranges if there is an indication of trustworthiness for a dependency, but it does not dive further into which factors aside from the ”wisdom of the crowds” principle could be relevant.

Kula et al. investigated the extent to which developers update their library dependencies, finding that 81.5% of projects still include outdated dependencies [8], as well as the notion of *trust* when using dependencies. They also investigated to what extent developers trusted new releases of dependencies to adhere to `semver` policies. Despite stressing the importance of trust when including dependencies and declaring dependencies, Kula et al. state the need for more research on how to certify the trustworthiness of a library. [7]

The Open-Source Security Foundation (OSSF)[9] provides a Scorecard tool[10] which assesses open-source projects on ”several critical security metrics, such as branch protection, dependency updates, and the use of fuzzing and static analysis tools.” [10] It allows maintainers to improve their security strategies as well as helping developers assess their software supply chain. Sonatype demonstrated the utility of Scorecard by accurately predicting known vulnerabilities in 78% of projects [11]. While the Scorecard score only provides security statistics, we wished to provide a stronger foundation of trust for a dependency by investigating additional statistics such as maintenance policies and semantic compatibility. This choice is backed by Zahan, N and Williams, L, who evaluated Scorecard's scores alongside other metrics such as release frequency, finding both to correspond to fewer vulnerabilities and faster patching. [13]

The value of version ranges is clear: automatic adoption of security patches or new functionality minimises developer effort in maintaining their dependencies. However, a lack of adherence to `semver` policies and awareness of vulnerabilities, causes consuming developers to stick to using fixed versions when declaring dependencies. This has several consequences, most notably the persistent use of vulnerable dependencies despite available patches.

While tools like Ranger show that `semver`-compliant version ranges can remediate a majority of vulnerabilities, such tools are limited to known CVEs. Additionally, due to underutilisation of version ranges in Maven compared to other dependency managers, more empirical research into a general-purpose dataset is necessary to provide an understanding of the state of dependency versions in the Maven ecosystem.

Our paper expands on the previously mentioned work by analysing a large variety of projects in the Maven ecosystem to determine the current state of dependency declaration and the role version ranges can play in improving security, utility, and maintenance overhead for developers. Additionally, we provided a concrete basis for defining the notion of trust based on a combination of quantitative metrics which have each been shown individually to contribute to the security and stability of a dependency.

[6]https://packagist.org/

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

[8]https://www.cve.org/

[9]https://openssf.org/

[10]https://openssf.org/projects/scorecard/

3 Methodology

This section explains in detail the methods used for gathering the used dataset and how this dataset was analysed to gather the results used for answering the first research question. Next, it details the methods used for answering the second research question.

3.1 Analysis of declared dependencies (RQ1)

To gather a representative dataset of open-source projects and dependencies, a GitHub repository search tool[11] developed specifically for sampling projects for research purposes [3] was used to make a selection of projects based on the following carefully selected parameters:

- **No toy projects.** By selecting only projects with over 1,000 lines of code and 100 commits, small toy projects were filtered out. Most likely, these projects would not have contributed anything meaningful to the research because smaller projects have very few dependencies and do not employ any dependency management policies.
- **Actively maintained.** In order to select actively maintained projects, one of the filtering requirements was for the most recent activity to have been after January 1, 2025. It is important for the projects to be actively maintained because it suggests that the developers or maintainers still put thought into the management of dependencies for those projects.
- **Mature.** In the earlier stages of a project, dependency management is not a priority, because a skeleton and minimum viable product have to be realised first. Therefore, projects that were less than a year old were filtered out to increase our confidence that the developers had implemented a dependency management strategy.

This search resulted in 6.764 projects, out of which 4.140 projects contained a `pom.xml` file, which is used by Maven. This Project Object Model contains all relevant properties related to dependency management, including plugins, excluded dependencies and defined variables. These variables can be used to define a version which can be used for multiple dependencies or plugins which are part of the same family. For both dependency type and artifact definitions, we had to put extra work into resolving these variables. This was not always trivial, as properties defined in a parent module may be referenced in child modules.

To prepare for the necessary analysis to answer RQ1, each project was analysed and their dependencies were extracted with their defined versions. This resulted in a list of 224k dependency declarations, with around 65k unique dependencies. This large number is partly due to the large amount of references to parent modules, which turned out not to be an issue for analysis.

In order to answer the research question, it was necessary to know which declared versions were outdated, and whether or not they could be updated to a newer version without the introduction of breaking changes. To do this, a Java API comparison tool, `japicmp`[12] was used. This tool compares two versions of a jar archive by comparing the function signatures for all classes, outputting whether a function was added, changed, or removed. Changes to a function signature indicate backward incompatibility, as does the removal of a function.

As this tool only compares signatures, behavioral changes, such as a different variable with the same type being returned, may raise a false positive. In-depth function behaviour checking would require significantly more effort and was out of the scope of this research.

Each declared version was compared step-wise to a newer version, comparing the versions until two were no longer backward-compatible, in order to count how outdated the declared dependency was.

[11]https://seart-ghs.si.usi.ch/

[12]https://siom79.github.io/japicmp/

3.2 Trustworthiness of dependencies (RQ2)

For RQ2, in order to provide concrete, quantitative metrics to determine whether a dependency could be trusted, a *trust score* was compiled based on several parameters:

- **OpenSSF Scorecard Score.**[13] This is an assessment of a library based on a number of heuristics to indicate the security of a project. Some important factors include code reviews, the existence of a security policy, and the use of dependency update tools. By itself, this is a very strong metric to indicate whether a dependency can be trusted. Sonatype used Scorecard to successfully predict vulnerabilities in 78% of projects. [11]. However, since it only analyses the security hygiene for a dependency, additional heuristics were necessary to provide a full-fledged trust score.
- **Release frequency** A high frequency of releases indicates active maintenance, suggesting that potential vulnerabilities get fixed promptly after discovery [13].
- **Automated analysis of update compatibility.** This is an important factor in motivating developers to use version ranges, as low semantic compatibility introduces a large maintenance overhead. Providing insight of the amount of backward-compatible minor and patch updates can provide a lot of information on the trustworthiness of a dependency.

Using these heuristics, we developed a command-line Trust Score Tool, TeSTer, which allows developers to analyse a dependency to determine its trustworthiness and make conscious, risk-aware decisions on whether to use the dependency and whether to use an open version range when declaring the dependency.

OpenSSF's Scorecard tool analyses a dependency's repository to compile a scorecard based on a good amount of heuristics. Unfortunately, this tool cannot analyse projects that are not hosted on GitHub or GitLab, barring assessment of some major dependencies only hosted on the Maven repository. Our command-line tool simply pulls OpenSSF's docker image and runs the analysis on the desired dependency. Without additional arguments, only the aggregated Scorecard score is displayed, but users can add the argument `--full_scorecard` to display the full listing of scores.

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

To calculate the release frequency, a list of all versions and their publication dates was polled from the maven search API and sorted by release date to determine the Mean Time To Update (MTTU) in days. The reason for sorting by release date is that sometimes patch updates for older but still frequently used versions may get released, causing a version to be semantically older but newer in timestamp. A lower MTTU indicates an active maintenance policy, ensuring vulnerabilities get patched sooner [13].

Finally, to calculate the compatibility score, the same list of versions was used and compared using the API comparison tool mentioned in section 3.1. In order to avoid comparisons between updates that changed the major version number, all versions were divided into *major groups*, then compared using the tool, keeping track of how many updates were backward-compatible. Subsequently, the results were combined and a final compatibility score was calculated, along with scores for how many minor or patch versions were backward-compatible.

Finally, a combined score was calculated according to a weighted linear combination of scores, according to the following formulas:

$$release\_norm = 1 - \frac{days}{365} \tag{1}$$

$$\begin{aligned} trustscore = & W_1 * scorecard \\ & W_2 * release\_norm \\ & W_3 * compatibility \end{aligned} \tag{2}$$

For equation 2, weights were defined according to impact of the stat and fine-tuning based on popular dependencies, resulting in the weights shown in table 1. The reason for the priority on semantic compatibility comes from the main focus of our research: a higher compatibility score leads to more motivation to use open version ranges.

Metric	Weight
Scorecard score	0.35
MTTU	0.2
Patch compatibility	0.3
Minor compatibilty	0.25

Table 1: Weights of TeSTer metrics for the combined Trust Score

In the absence of a GitHub or GitLab repository link, the Scorecard score cannot be generated and is therefore not taken into account for the final score, so the weights for the other metrics are scaled accordingly, keeping the same ratio.

4 Results

This section highlights the results of analysis performed for the first research question, and proposes a new command-line tool, TeSTer, to answer the second research question.

4.1 Impact of fixed versions (RQ1)

Of the 224k dependency declarations, around 65k had a newer available version. As shown in table 3, 34k dependencies had at least one newer backward compatible version available. Out of these 34k dependencies, 14k of these had a newer minor version available and 20k had a newer patch version available. This shows that nearly 30% of declared dependencies are outdated, emphasizing the need for automated update mechanisms such as version ranges.

A substantial portion of these outdated versions could be safely updated to a newer version without compatibility issues: over 60% for minor updates and almost 50% for patch updates. This strongly supports our hypothesis that version ranges could reduce maintenance overhead and improve security for developers that trust the dependencies they use.

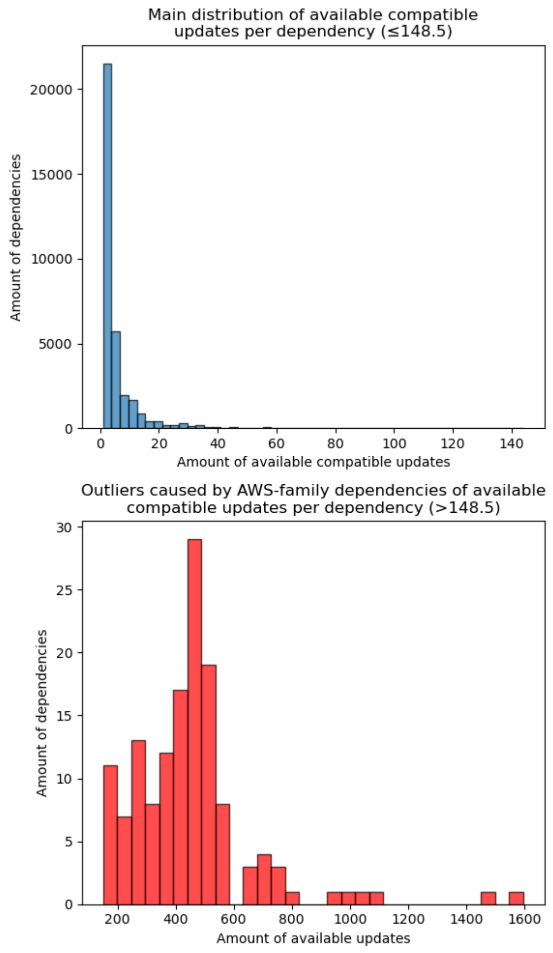

Figure 1: Distributions of amount of available updates per dependency, separated by main distribution and outliers caused by dependencies in the Amazon Web Services family.

Figure 1 shows the distribution of the amount of backward-compatible updates that are available for the analysed dependencies. The large amount of outliers can be traced back to dependencies under the Amazon Web Services (AWS) family, which upon further inspection turned out to have a very large amount of versions, explaining the amount of available updates.

Figure 2 show the 99th percentile of the amount of available compatible updates, ruling out the large amount of out-

GroupId	ArtifactId	Scorecard Score	Minor Score	Patch Score	Release Frequency	Trust Score
org.junit.jupiter	junit-jupiter-api	8.0	44.8%	76.8%	37 days	74.1
org.jetbrains.kotlin	kotlin-stdlib	3.5	12.5%	77.0%	18 dayas	53.0
org.slf4j	slf4j-api	4.7	41.67%	85.23%	64 days	62.6
com.google.guava	guava	8.8	0.00%	17.07%	36 days	53.1
org.mockito	mockito-core	7.5	53.7%	77.7%	16 days	75.6

Table 2: TeSTer analysis results of the most popular dependencies in the Maven repository

	Compatible	Incompatible	Total
Newer minor available	13,825 (62,2%)	8,409 (37,8%)	22,234
Newer patch available	20,481 (48,0%)	22,926 (52,0%)	43,407
Total	34,306 (52,2%)	31,335 (47,8%)	65,641

Table 3: The amount of dependencies which can or can not be updated to a newer minor or major version.

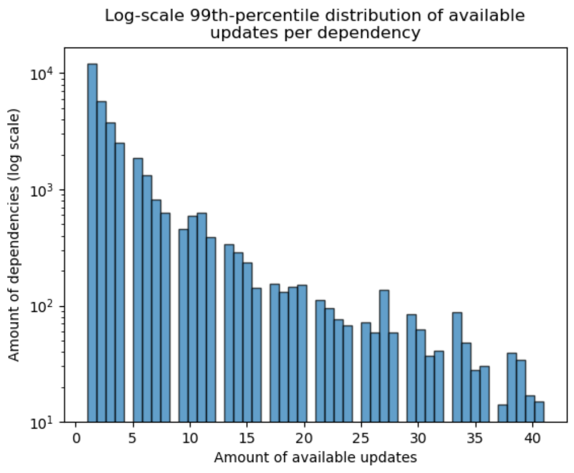

Figure 2: Logarithmic distribution of the 99th percentile of available updates per dependency.

liers caused by AWS dependencies to provide a clearer image of the results. This distribution suggests that while most dependencies can only be updated once without compatibility issues, a considerable amount of dependencies is multiple versions behind, causing developers to miss out on multiple bug-fix or security patches, or new functionality added by minor updates.

Further analysis revealed that not all dependencies could be updated to their newest version. As shown in table 4, almost half of the dependencies which had newer compatible versions available were actually able to be updated to their most recent version with the same major version number without conflicts. The ”unknown” category refers to dependencies for which the declared version was, for unknown reasons, not present in the `metadata.xml` file and thus could not be used for analysis. Manual inspection showed that, for each of these cases, the declared version was significantly outdated and can be safely assumed to not be compatible with the latest version. The fact that almost half of the outdated dependencies can be safely updated to the latest version implies that even conservative use of ranges, such as ranges including only patch versions, could yield significant benefit.

	Amount
Latest release backward compatible	28,930 (44,8%)
Latest release not backward compatible	36,152 (55,9%)
Unknown	855 (1,32%)
Total	65,641

Table 4: Amount of dependencies which are able to be updated to their latest version without breaking changes.

Taken together, these results indicate that 52.2% of declared dependencies are outdated and would have safely been resolved to a newer version, had a range been used. It is important to note that this does not mean that it is safe to use an infinite version range, since more than half of these dependencies could not be updated to the latest version without breaking changes being introduced. Instead, a more conservative range including only a few versions can be used.

Many dependencies even have multiple newer versions available, suggesting that they are potentially missing out on multiple bug-fix or security patches, or additional useful functionality added by minor updates. The large amount of available compatible updates for dependencies with an active release schedule such as AWS-family dependencies further serve to stress the importance of release frequency as a metric for defining the trustworthiness of a dependency.

4.2 Trust Score (RQ2)

To demonstrate the utility of our command-line tool, TeSTer, we ran it on some of the most popular and representative dependencies listed in the Maven repository[14]. The results are listed in table 2. One clear indication of TeSTer’s utility is the analysis of the `com.google.guava:guava` dependency. Despite the high Scorecard score, the low semantic compatibility score is a very important factor in deciding whether to trust the dependency. TeSTer reflects this by giving it a low combined score of 53.1, expressing a low trustworthiness. Conversely, the `org.jetbrains.kotlin:kotlin-stdlib` dependency has a decent patch score and release frequency, but low Scorecard and minor scores, which also results in a low Trust Score.

[14]https://mvnrepository.com/popular

`software.amazon.awssdk:s3`	
Scorecard Score	6.6
Minor compatibility score	75.0%
Patch compatibility score	99.49%
Release frequency	1 day
Trust Score	8.29/10.0

Table 5: Results from TeSTer's analysis of the most popular aws dependency.

Analysis of AWS dependencies, which were shown to be significant outliers in terms of available compatible updates in section 4.1, shows that the dependencies follow a daily release schedule. As shown in table 5, nearly all patch updates are backward compatible, while minor updates are less, but still fairly reliable.

The creation of TeSTer provides developer with a simple way to analyse a desired dependency in order to make conscious, risk-aware decisions on whether to include and use version ranges for a dependency. This way, developers do not have to rely on automated tools to make decisions for them, but rather use the tool to aid them with quantitative statistics while allowing the developer to maintain full control over their projects.

5 Discussion

This section discusses the results found in section 4, relates them back to previous work mentioned in section 2, and proposes avenues for further research. Next, it discusses threats to validity of the research and discusses how the research done is responsible, ethical, and reproducible.

5.1 Impact of fixed versions

The results found by section 4.1 indicate that, out of the outdated dependencies with a newer minor version available, 62% can be safely updated without compatibility issues. For patch versions, this percentage is surprisingly lower, with 48% of newer patch releases being backward compatible. Both percentages are lower than expected, considering the research done by Raemakers et al. which showed that only a third of minor updates and a quarter of patch updates introduced breaking changes [9].

Analysis of how many newer versions are available for dependencies shows that a considerable amount of dependencies are multiple versions behind. The results show that half of the declared dependencies are outdated, of which again around half can be updated to the latest version with the same major version number. This means that only a quarter of all currently declared dependencies can be safely defined with a version range instead of a fixed version, which is less than we expected. Still, it should provide some motivation for developers to put more thought into their dependency declaration strategies.

It is hard to quantify exactly to what extent the results motivate developers to adopt version ranges in their projects. Further qualitative research based on surveys and case studies needs to be performed to properly establish the impact of the results.

5.2 Trust Score

TeSTer analysis of popular dependencies, as shown in table 2, produces a variety of Trust Scores. However, it is quite notable that there are no extremely low or extremely high scores. This was the case for nearly all dependencies we analysed manually, posing the question of whether a trilemma exists in dependency development, where a dependency cannot have a high security hygiene, low update frequency, and high semantic compatibility at the same time. Further research into these factors is required to pinpoint the exact cause of this relationship. Another reason for the low variety in Trust Scores may be the limited amount of factors, indicating the need for more research on other factors that may influence the trustworthiness of dependencies.

As is the case for the first research question, the impact of TeSTer's analysis on developer motivation is yet to be measured and further qualitative research is required to precisely determine the utility of the Trust Score.

Currently, TeSTer's weights were manually selected and adjusted based on manual analysis of a small set of popular dependencies. These weights can be further and more accurately specified using an LLM, resulting in a more representative trust score.

TeSTer is currently available as a command-line tool, hosted publicly on our GitHub repository. It could be made even more accessible by repurposing it as a Maven plugin, integrating it into GitHub CI pipelines, or developing an IDE plugin which provides real-time suggestions to convert fixed versions into ranges. The latter is similar to Ranger's functionality[14], but provides developers a choice based on multiple quantitative metrics instead of automatically making changes.

5.3 Threats to validity

This section discusses internal and external threats to the validity of the research, and how these were circumvented.

An internal threat to validity is the limited selection of Trust Score metrics due to the small scope and timeframe of our research. Aside from the Scorecard score, release frequency, and semantic compatibility, many other factors could contribute to the trustworthiness of a dependency. This may result in an overly simplified Trust Score, or causes developers to miss out on key signals that could affect their decision-making. Therefore, we present TeSTer as a *proof of concept*, recommending further research on the inclusion of other factors to further solidify the Trust Score metric.

An external threat to validity is the fact that only dependencies hosted on GitHub or GitLab can be analysed by scorecard, causing the absence of an important metric when analysing a dependency that is not hosted on these platforms. To partially circumvent this, the other two factors are scaled accordingly, but a trust score based on only two metrics is not very representative, further indicating the need for more research into additional factors to compose the Trust Score rating.

Another external threat to validity is the assumption of developer rationality when adopting dependencies. We assume developers to easily change their development behaviour when presented with rational incentives, but in reality, other factors like personal habits and company policies may also influence developers. This could result in a lower than expected adoption of version ranges, emphasizing the need for more qualitative, survey-based research to validate the motivation impact of our research.

5.4 Responsible Research

This section highlights factors contributing to the integrity, reproducibility, and ethical justification of our research.

Data privacy and integrity. All data used for the research is publically available and usable by anyone who wishes to replicate the experiments. No personal data of any kind which may link the research to individuals or companies was used, ensuring full privacy. To ensure data variety, we selected publicly available data filtering only on parameters ensuring the data to be representative for our research, not on ethnic, personal, or societal factors.

Reproducibility and replicability. All steps of the methodology were clearly mentioned and explained. Researchers wishing to reproduce the results can sample a near-identical set of GitHub projects and replicate the experiments to get the same results using our code, which is publicly available on our GitHub repository. TeSTer performs deterministic calculations so it will always produce the same results if ran twice on the same dependency, but is also reactionary to changes in a dependency.

Usage of Large Language Models (LLMS). All text in this research paper is written by humans, with minimal input of LLMS to evaluate sections on writing style. No text was copied from LLM outputs, and the entire paper was subjected to peer feedback before submission. LLMs such as GitHub Copilot and ChatGPT were used to aid in writing code for analysing the data, but all code was critically reviewed before use.

6 Conclusions

With the maintenance of a large set of dependencies becoming harder as the size of projects increases, Java developers look to Maven to aid them in declaring dependencies, avoiding version conflicts, and automatically resolving compatible versions of desired dependencies. However, despite the ability to declare version ranges in Maven, the lack of adherence to semantic versioning policies or knowledge as to which dependencies can be trusted to do so leads to this powerful feature being under-utilised. To motivate developers to consider making the transition to using version ranges, we performed quantitative empirical analysis of a large set of open-source projects, showing that more than half of the declared dependencies can be safely updated to a newer compatible version, and a quarter can be safely updated to the latest minor or patch release.

To further aid developers in deciding whether to use a version range for a dependency, we proposed TeSTer, a command-line Trust Score tool which analyses a dependency for key security and compatibility heuristics and outputs a score indicating whether the dependency can be trusted to promptly release security patches which adhere to the semantic versioning policy. TeSTer can be used by developers as a conceptual tool to support informed decisions in their development process while maintaining full control over their projects, promoting the use of version ranges in open-source Maven projects by offering quantitative insights into security hygiene, release frequency, and adherence to semantic versioning.

References

[1] T. Mens A. Decan. What do package dependencies tell us about semantic versioning? In *IEEE Transactions on Software Engineering*, volume 47.6, pages 1226–1240, 2019.

[2] Maven documentation. https://maven.apache.org/guides/index.html.

[3] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 560–564. IEEE, 2021.

[4] E Dell, S Bugiel, S Fahl, Y Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *CCS*, pages 2187–2200, 2017.

[5] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. Dependency versioning in the wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 349–359, 2019.

[6] F. Massacci I. Pashchenko, D. Vu. A qualitative study of dependency management and its security implications. In *2020 ACM SIGSAC conference on computer and communications security*, pages 1513–1531, 2020.

[7] Raula Gaikovina Kula, Daniel M. German, Takashi Ishio, and Katsuro Inoue. Trusting a library: A study of the latency to adopt the latest maven release. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 520–524, 2015.

[8] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Emperical Software Engineering*, 23:384–417, 2017.

[9] J. Visser S. Raemakers, A. van Deursen. Semantic versioning and impact of breaking changes in the maven repository. In *Journal of Systems and Software*, volume 129, pages 140–158, 2017.

[10] Accelerating openssf adoption: Unlocking scorecard insights with a centralized dashboard. https://openssf.org/blog/2025/01/22/accelerating-openssf-adoption-unlocking-scorecard-insights-with-a-centralized-dashboard/, 2025.

[11] Sonatype. 8th state of the software supply chain. Technical report, Sonatype, 2022.

[12] Y Wang, SC Cheung, H Yu, and Z Zhu. Boosting the propagation of vulnerability fixes in the npm ecosystem. *Springer*, 2025.

[13] Nusrat Zahan and Laurie Williams. Prioritizing security practice adoption: Empirical insights on software security outcomes in the npm ecosystem, 2025.

[14] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. Mitigating persistence of open-source vulnerabilities in maven ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 191–203, 2023.