Aggregating Security Measures from the Dependency Tree

Sarah Elder
seelder@alumni.ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Alex Klevans
asklevan@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Ranindya Paramitha
rparami@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Marcelo d'Amorim
mdamori@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Laurie Williams
lawilli3@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Abstract

Metrics available to practitioners for determining the security of open-source software packages often overlook potential security issues, such as latent vulnerabilities, that may arise in the package's dependencies. However, security risks in the dependencies of a package can still increase the risk to the package itself. *The goal of this work is to aid developers and software architects in making good component choices by analyzing the relevance and utility of security measures of the entire dependency tree associated with a package.* We propose a set of approaches for aggregating security measures from OpenSSF Scorecard that include information from the dependency tree of a package (aggregate scores). We evaluated the aggregate scores in two ways. For one, we performed statistical analysis on packages in the Go ecosystem. We determined the effect size of aggregation and assessed whether aggregation influences the measure's relationship with vulnerability count, a common security risk measure. The other evaluation was a survey to see if practitioners would prefer metrics that are aggregated compared to metrics that focus only on the root package. Our results indicate that for some security risk measures, such as the number of Binary Artifacts in a package, aggregation has a negligible statistical effect. However, for the same Binary Artifacts measure, most survey participants preferred to have aggregate scores. In contrast, most respondents were less concerned about whether maintainers of the package's dependencies performed Code Review, as long as maintainers performed Code Review on the root package.

CCS Concepts

• **Security and privacy → Software and application security**; • **Software and its engineering → Software libraries and repositories**.

Keywords

Aggregation; Security Measures; OpenSSF Scorecard

SCORED '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1915-8/2025/10
https://doi.org/10.1145/3733827.3765528

ACM Reference Format:
Sarah Elder, Alex Klevans, Ranindya Paramitha, Marcelo d'Amorim, and Laurie Williams. 2025. Aggregating Security Measures from the Dependency Tree. In *Proceedings of the 2025 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3733827.3765528

1 Introduction

Factors such as the presence of *vulnerabilities* and whether a package is actively *maintained* can inform practitioners' decisions to include open-source components in their software [1, 2, 7, 21, 24, 26, 27, 39, 41]. One tool designed to help practitioners make these decisions is the Open Source Security Foundation (OpenSSF) Scorecard [29]. OpenSSF Scorecard automatically collects a set of 18 security risk measures covering common attributes highlighted in prior work [24, 39], such as whether the package is *maintained.* However, many of the measures are based on the package itself, with limited information about the package's dependencies. Risks in a package's dependencies, such as the presence of security vulnerabilities, may pose a risk to the package itself [22, 44, 45].

Assessing risk may require practitioners to review the entire dependency tree of a package. Zerouali et al. [44] found that 15.7% of the software packages in the NodeJS Package Manager (npm) are exposed to security vulnerabilities in their direct dependencies, i.e., dependencies the package explicitly referenced. However, they found that 36.5% of npm packages are exposed to vulnerabilities indirectly via transitive dependencies. Transitive, also known as *indirect*, dependencies, are not directly referenced but may be used by a package because they *are* referenced by other dependencies.

The goal of this work is to aid developers and software architects in making good component choices by analyzing the relevance and utility of security measures of the entire dependency tree associated with a package.

First, we performed a set of preliminary interviews to help us develop ways to aggregate security measures taken from all software in a package's dependency tree. Then, the main part of our research is split into two parts. For one, we perform a large-scale data analysis of packages in the Go ecosystem to answer the following research questions:

- RQ1: *Difference*: What is the difference between the root package score and the aggregate score?
- RQ2: *Vulnerabilities*: How does the aggregate score change the relationship between security metrics and vulnerabilities?

We selected Go as the target ecosystem because the packages are easily mapped to source repositories since the package name should indicate the source [6]. Although understudied in the academic literature, Go is used by well-known organizations including Google, Microsoft, Capital One, American Express, Uber, and Netflix [15]. The other part of our study is a survey of practitioners, to address the following research question:

- RQ3: *Practitioners*: What approaches do practitioners recommend for aggregating measures across a component's dependency tree?

We found that some measures, such as the assessment of whether package maintainers were performing *Code Review*, did not seem to benefit from aggregation. Others, such as the security of GitHub *Branch Protection* policies, may benefit from using a score that aggregates information from across the package's dependency tree. The contributions of this work are as follows:

- a comparison of the original security scores for the root package component against aggregate scores (RQ1 and RQ2)
- a survey of practitioners to understand whether they recommend the original root package score or an aggregate score (RQ3)
- a dataset of security vulnerabilities and dependency trees for the Go ecosystem (developed for our evaluation of RQ1 and RQ2)

Our data and other study materials can be accessed online at [3].

2 Measures from OpenSSF Scorecard

OpenSSF Scorecard is a static analysis tool that examines available data, such as source code and GitHub Workflows scripts, to produce a set of security-related measures. Currently, OpenSSF Scorecard is only able to analyze packages whose source repository is hosted on GitHub or GitLab [29]. In this research, we focus on GitHub repositories since only two of the 14 checks examined in this study are fully supported for GitLab repositories[30]. The Scorecard tool can be run by anyone with access to the source code repository of a package. The tool is also run bi-weekly on commonly used packages to populate a Google BigQuery database [29]. OpenSSF Scorecard is one of the technical initiatives promoted by The Open Source Security Foundation (OpenSSF) [28]. Members of OpenSSF include GitHub, IBM, and Google.

2.1 The Measures (Checks)

The measures assessed by OpenSSF Scorecard are referred to as 'checks'. Each Scorecard check has a *risk* level, 'low', 'medium', 'high', or 'critical'. We remove low-level risks from our analysis based on input from OpenSSF Scorecard and feedback from practitioners in the Preliminary Interviews, which are discussed in Section 3. The 14 checks with a risk level of 'medium' or higher are shown in Table 1.

The Name and Description of each check are shown in the first two columns of Table 1. The Name of the check indicates the concept that the check attempts to capture. The Description in column 2 of Table 1 is copied from the OpenSSF Scorecard documentation [29].

The third and fourth columns of Table 1 provide information on the *implementation* of each check. Additional details on how each check is computed are available in the Scorecard documentation [31]. In the third column of Table 1, we include the Constituent Elements of the package repository that are being assessed. For example, points for the *Security Policy* check are based on static analysis of the `SECURITY.md` file. In the fourth column of Table 1, we indicate the Standardization (Stand.) approach used for each check to convert the measures of the constituent elements into a score between 1 and 10. As we will discuss in Section 2.2, the standardization approach may influence measure aggregation. The standardization approaches can be grouped into four types:

Deduction (D) Deduction standardization approaches deduct constituent elements, such as the # Vulnerabilities in the *Vulnerabilities* check, from the maximum score (10).

Binary (B) Binary standardization approaches are those that produce a binary output (either 0 or 10).

Points-Based (Pt) In Points-Based standardization, the package is awarded points based on measurements taken of the constituent elements. For example, in the *Security Policy* check, 6 points are awarded for a vulnerability reporting mechanism.

Proportional (Pr) Proportional standardization approaches determine the actual output of the constituent elements, divide it by the expected (minimum) output, and multiply by 10.

If the calculations result in a score greater than 10 or lower than 0, the score is standardized to 10 or 0, respectively. Checks may apply different approaches on a case-by-case basis. For example, the SAST check uses binary standardization to capture the use of the Sonar tool, and proportional standardization to capture other tools.

Invalid Scores. There are instances when the Scorecard tool cannot produce a result, and the tool indicates that a particular check's score is invalid. In these cases, the tool outputs "−1" rather than a score. The reasons a result cannot be produced vary based on the mechanisms used by the check. For example, the metadata used to analyze the Branch Protection check may not be publicly available if a repository's owners set their permissions to block this information. If the repository owners block public access to Branch Protection information and the tool is run by the repository administrator, the Branch Protection check may produce a result. However, in this same example, if the tool is run publicly, the Branch Protection check returns an error and an invalid score.

2.2 Existing Aggregation for *Vulnerabilities*

One check, the *Vulnerabilities* check, is already aggregated across the dependency tree [32]. As shown in Table 1, the score of the *Vulnerabilities* check is standardized by subtracting the number of vulnerabilities from 10 (Deduction). To aggregate the *Vulnerabilities* check across the dependency tree, OpenSSF collects the # Vulnerabilities from the entire dependency tree instead of only from the base component. Hence, this aggregation method depends on the original *Standardization*.

3 Preliminary Interviews

We performed a set of semi-structured interviews to provide sufficient background for our research. At the time of the interviews, we were only aware of 2 papers on OpenSSF scorecard, discussed in the Related Work (Section 9). The lack of literature precluded literature-based initial stages, such as a formal Systematic Literature Review. Given this lack of information, we considered it important to get perspectives from current industry practitioners before making

Table 1: Scorecard Checks

Name	Description *from https://securityscorecards.dev*	Constituent Elements	Stand.
Binary Artifacts	"Is the project free of checked-in binaries?"	Binaries in source code	D
Branch Protection	"Does the project use GitHub Branch Protection?"	GitHub Settings	Pt
Code Review	"Does the project require code review before code is merged?"	GitHub Commit & related metadata	Pr
Dangerous Workflow	"Does the project avoid dangerous coding patterns in GitHub Actions?"	GitHub Actions & Workflows	B
Dependency Update Tool	"Does the project use tools to help update its dependencies?"	GitHub Apps & Workflows	B
Fuzzing	"Does the project use fuzzing tools?"	OSS-Fuzz project list; files & functions associated with fuzzers	B
Maintained	"Is the project maintained?"	Archived Status, Commits & Comments	Pr & B
Packaging	"Does the project build and publish official packages from CI/CD, e.g., GitHub Publishing?"	GitHub Actions & Workflows	B
Pinned Dependencies	"Does the project declare and pin dependencies?"	Dockerfiles, scripts, & GitHub Workflows	Pr
SAST	"Does the project use static analysis tools?"	GitHub Apps and/or GitHub workflows	Pr & B
Security Policy	"Does the project contain a security policy?"	`SECURITY.md` file location and contents	Pt
Signed Releases	"Does the project cryptographically sign releases?"	Filenames of "release assets" on GitHub	Pr
Token Permissions	"Does the project declare GitHub workflow tokens as read-only?"	GitHub workflow yaml files	D
Vulnerabilities	"Does the project have unfixed vulnerabilities?"	Vulnerabilities from OSV Scanner	D

assumptions about the topic. In Section 3.1 we describe the interview process, while in Section 3.2 we describe findings from the interviews that shaped our analysis. Our interview process follows our IRB protocol, as we discuss in Section 11.

3.1 Interview Process

Each interview began with a series of background questions, including "What is your experience in working <in/with> security?" and "How are decisions made about whether to add a particular dependency to a project at your organization?" to help us interpret participant responses. We then proceeded to the technical questions.

In each interview, we use an example package and its dependency tree, such as the one in Figure 1. Where possible, this tree was selected because it was a dependency of software the practitioner was familiar with. All technical questions were asked in the context of this dependency tree, and hypothetical scores were shown for each of the packages of the dependency tree, as can be seen in the example in Figure 1. Figure 1 illustrates *Code Review* scores.

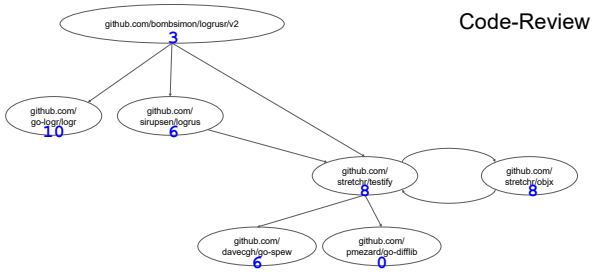

Figure 1: Example Dependency Tree from Interviews with Original Scores

We would go through as many of the checks as we could within the one-hour timeframe, starting with the *Vulnerabilities* check. For each check, we asked the following questions:

(1) Are some of the scores more concerning than others? If so, why? What factors (e.g., size of the dependency, depth of the dependency) would be needed to make this determination?
(2) If you are determining whether to include <package name>, how should information from this entire dependency tree be aggregated? Should the scores be averaged? Should the minimum score be taken? Should the maximum score be taken? Should a different approach be used?

We start with the *Vulnerabilities* check because vulnerabilities are commonly associated with security risk [22, 45]. Additionally, the *Vulnerabilities* check is already aggregated, allowing us to use this check to illustrate our goal. The aggregation also meant that the second question for the *Vulnerabilities* check was replaced with:

(2) How do you feel about this aggregation approach? If you are determining whether to include <package name>, should a different approach be used (e.g., should the total vulnerabilities be subtracted from 10)?

Participants: Overall, we interviewed seven participants, selected based on convenience. All of the participants had professional, industry experience in software development or security; and six participants had experience with *open source* software development. Of these participants, three had experience in Go and were therefore able to participate in our later survey, as we discuss in Section 6.2.3.

3.2 Findings

The following findings were each discussed in at least three of the interviews. The first four findings influenced our aggregation approaches, which we will discuss in Section 4. The last two findings influenced our overall methodology, as we will discuss in Section 6.

(1) **The aggregate score should not be higher than root**

(2) For most checks, it would be **good to know the "Worst Case"** in the dependency tree (i.e., minimum score)
(3) Practitioners wanted to know **if there are *any* Binary-Artifacts** in the package's dependency tree
(4) **Code Review scores of packages at lower levels of the dependency tree are less important** than scores closer to the root.
(5) Practitioners were **more concerned about lower scores**, and not concerned about higher scores
(6) Practitioners considered it **difficult to judge how the aggregate value would relate to risk for low-risk checks**

4 Aggregation Methods

Based on the existing aggregation approach for the *Vulnerabilities* check, our preliminary interviews, and the implementation of each check, we developed aggregation methods for 13 checks that are currently unaggregated. The aggregation approaches are summarized below.

Binary Artifacts Subtracting 1 from the score for each Binary Artifact in the entire dependency tree.
Branch Protection Taking the minimum value (i.e. "worst case") for the entire dependency tree.
Code Review Taking the minimum value of the root node and its direct dependencies
Dangerous Workflow Subtracting the number of dependencies with Dangerous Workflows from the root score.
Dependency Update Tool Taking the minimum value for the entire dependency tree.
Fuzzing Subtracting the number of dependencies that do *not* use fuzzing from the root score.
Maintained Taking the minimum value for the entire dependency tree
Packaging Taking the minimum value for the entire dependency tree
Pinned Dependencies Taking the average score for the entire dependency tree or the original score, whichever is lower
SAST Taking the minimum value of the root node and its direct dependencies
Security Policy Taking the minimum value for the entire dependency tree.
Signed Releases Taking the minimum value for the entire dependency tree.
Token Permissions Taking the minimum value for the entire dependency tree.

If any calculation resulted in a score greater than 10 or less than 0, the score was standardized to 10 or 0, respectively. Additionally, if a dependency of one of the packages had an invalid score, we removed the invalid score and calculated the aggregate value based on the root score and the remaining dependencies with valid scores.

For example, consider the Code Review aggregation described above, which is the minimum of the root node and its direct dependencies. If we look at the example dependency tree in Figure 1, we see that the root package and its direct dependencies have the Code Review scores [3, 10, 6, 8, 8]. The minimum of these values is 3. If, in this same example, the `github.com/go-logr/logr` package had an invalid score, we would then drop the package from our calculation and calculate the minimum of [3, 6, 8, 8]. In this example, the result of aggregation would not change since the minimum of the set is still 3. However, as we acknowledge in Section 10, there may be cases where our decision to simply drop the invalid scores would potentially alter the result.

5 Data Collection Overview

The two primary components of our dataset are the Go packages, which we describe in Section 5.1, and the Scorecard data, which we discuss in Section 5.2. We also collected vulnerabilities that have been found in Go packages, which we describe in Section 5.3.

A key distinction in this dataset is between dependents and dependencies. A package's *dependent* is software that depends on the package. A package's *dependency* is the software the package depends on.

5.1 Go Packages

To extract a list of Go packages and their dependency trees, we leveraged the dataset provided by Open Source Insights [18], which was previously used by Hu et al. [20]. The dependency data in Open Source Insights uses best practices for dependency tree resolution, such as focusing on production dependencies [34]. Table 2 shows an overview of the package selection process.

Table 2: Go Dataset Selection Criteria

Total Number of Packages	768,115
Number of Packages where we can find a URL to a GitHub repo	742,027
Number of packages with at least one dependent from a *different* organization	57,613
Packages which have dependency tree data	44,237
Packages which have a GitHub repo for the entire dependency tree	34,617
Package Count after removing Redundant Packages	30,730
Packages with Valid Repository for Entire Dependency Tree	30,717
Packages for which there were no missing or invalid scores for individual checks	27,951

At the time of our data collection in June-July 2024, the Open Source Insights dataset listed 768,115 Go packages. Since we map each package to a Scorecard score via GitHub source code repositories, we then identify the GitHub repositories of the packages using metadata in the Open Source Insights dataset, the package names[1], and the Go package index. We exclude all packages for which we could not locate a GitHub repository. Additionally, as discussed in Section 2, we focus on GitHub repositories since only two checks, Binary Artifacts and Branch Protection, are fully supported when using OpenSSF Scorecard with Gitlab[30]. This resulted in the removal of 26,088 packages, with 742,027 packages remaining. Since we are interested in packages that practitioners may decide to use, we narrowed our dataset to packages that had dependents that were *not* from the same organization. This further reduced the number of packages examined to 57,613.

We then removed 22,996 packages that did not have sufficient dependency data. The majority (12,315), are leaf nodes that do not have dependencies. An additional 1,070 packages did not have

[1]Go package names are based on the source code URL, such as 'github.com/luzifer/rconfig'

dependency tree information. For 9,611 packages, we were unable to map a GitHub repository for all dependencies. Consequently, we identified 34,617 packages for which we could identify a GitHub respository for the entire dependency tree as seen in Table 2

Next, we identified packages where the information between the packages was likely to be redundant, using the following criteria:

- For some go packages, major versions are labeled with a new package name, for example, 'github.com/luzifer/rconfig/v1' and 'github.com/luzifer/rconfig/v2'. These packages were consolidated to the latest version.
- If the package name of one package contained the other, such as 'github.com/olive-io/bpmn' and 'github.com/olive-io/bpmn/schema', we selected the parent package.
- If a GitHub repository was associated in the metadata with two names, one under a GitHub URL and the other a different URL, (for example, 'github.com/gigawattio/testlib' and 'gigawatt.io/testlib'), we selected the non-GitHub URL as the primary package name.

After consolidating redundant packages, 30,730 packages remained as shown in Table 2.

There were 5 packages for which we were unable to get any OpenSSF Scorecard information since the original GitHub repository was unavailable. Removing the unavailable packages and the packages that depended on them reduced the dataset to 30,717 packages. As we will discuss in Section 5.2.2, we used the 30,717 packages to identify which checks we had sufficient data for.

After our analysis of Scorecard data, which we will discuss in detail in Section 5.2, an additional 2,766 packages were removed due to having invalid values for at least one of the six key checks. The final dataset contained 27,951 package, as shown in Table 2.

5.2 OpenSSF Scorecard Data

We collected OpenSSF Scorecard Scores based on the repository associated with each package. There were cases where multiple packages were associated with the same repository. Consequently, we analyzed Scorecard data for 30,545 repositories associated with the initial 30,717 packages.

5.2.1 Source. To extract Scorecard scores, we first rely on the Google BigQuery dataset provided by OpenSSF Scorecard [32]. However, OpenSSF Scorecard only runs the checks on popular packages, and packages that have been specifically requested to be included. Of the 30,717 root-level packages that met our non-Scorecard criteria, only 12,468 had Scorecard data in the original Biquery dataset. We ran Scorecard locally for any packages not included in the BigQuery dataset, using the same configuration as is used to populate the BigQuery dataset.

5.2.2 Checks Removed from the Analysis. Since the OpenSSF Scorecard's *Vulnerability* check is already aggregated, we only used it in our initial interviews and did not include it in any further analysis. Additional checks were removed due to how the checks are implemented, and the resulting lack of statistical data. Excluding checks in these cases does not mean the of the concept the check is trying to capture is invalid. The first of these checks is the *Dependency Update Tool* metric. The high computational resources required by the Dependency Update Tool check necessitated its removal from the bi-weekly updates to the BigQuery dataset used in this study[2]. The Dependency Update Tool check was therefore not included in our analysis.

Six additional checks were excluded due to invalid or missing data. As discussed in Section 2.1, the Scorecard tool marks the output for one or more checks as invalid, without providing a score. These scores we consider *invalid*. Additionally, in the Google BigQuery dataset, some packages did not have scores for all of the checks. In this case, we consider the score *missing*. Table 3 shows the number and percentage of repositories that had missing or invalid results out of the 30,545 repositories associated with our initial packages. Following the methodology of Zahan et al. [43], we focus our data analysis on checks for which less than 10% of the data is either missing or invalid. This removed the *Dangerous Workflow, Packaging, Pinned Dependencies, SAST, Signed Releases,* and *Token Permissions* checks. These checks were excluded based on the need to perform statistical analysis and the limitations of the OpenSSF Scorecard tool. The excluded check with the lowest percentage of missing or invalid scores, the SAST check (13%), had additionally been highlighted by our interviewees as potentially problematic since the current OpenSSF Scorecard rules are only able to identify usage for a small subset of available SAST tools. Six (6) checks remained for analysis: *Binary Artifacts, Branch Protection, Code Review, Fuzzing, Maintained,* and *Security Policy.*

Table 3: OpenSSF Checks Data Summary

Red text indicates why a check was excluded

Name	Missing or Invalid
Binary Artifacts	0 (0%)
Branch Protection	2,656 (8.7%)
Code Review	71 (< 0.1%)
Dangerous Workflow	22,257 (72.8%)
Fuzzing	9 (< 0.1%)
Maintained	1 (< 0.1%)
Packaging	29,881 (97.8%)
Pinned Dependencies	16,598 (54.3%)
SAST	3,963 (13.0%)
Security Policy	0 (0%)
Signed Releases	27,317 (89.4%)
Token Permissions	18,408 (60.3%)

5.3 Vulnerability Data

In RQ2, we examine whether the aggregation approaches alter the relationship between the scores and vulnerability count. Vulnerability count is a common metric for security [22, 45], and is a consideration for practitioners when determining which packages to use [35]. We collect vulnerability data from the Open Source Vulnerabilities database (OSV) [33]. OSV is readily available and covers a greater number of vulnerabilities than GitHub Security Advisories. We consider each unique ID in the OSV data to be a unique vulnerability. To determine the number of vulnerabilities in a package, we count each vulnerability in the current version of the package itself and each vulnerability in its dependencies. For dependencies, we map vulnerabilities based on the version from

[2]The configuration used to create the dataset was confirmed by the OpenSSF team.

the dependency trees extracted as described in Section 5.1. Additionally, to help improve our statistical analysis, we standardized the number of vulnerabilities and dealt with outliers using the same approach as the OpenSSF Scorecard *Vulnerabilities* metric [32], capping the number of vulnerabilities at 10. Capping the number of vulnerabilities only affected vulnerability counts for 3 packages.

6 Methodology

Our methodology is broken into two parts. In Section 6.1, we discuss the large-scale data analysis, covering RQ1 and RQ2. In Section 6.2, we discuss the methodology of our survey for RQ3. Our specific research questions are as follows.

- RQ1: *Difference*: What is the difference between the root package score and the aggregate score?
- RQ2: *Vulnerabilities*: How does the aggregate score change the relationship between security metrics and vulnerabilities?
- RQ3: *Practitioners*: What approaches do practitioners recommend for aggregating measures across a component's dependency tree?

6.1 Data Analysis

In this section, we describe the statistics used to answer RQ1 *Difference*, and RQ2 *Vulnerabilities*. The statistics are applied to the dataset described in Section 5.

6.1.1 RQ1: Difference. To assess the result of our aggregation approaches, we assess whether the difference in the original and aggregated scores is statistically significant by applying the Mann-Whitney U test for each check [8, 25]. We use the Bonferroni correction to mitigate the multiple comparisons problem [10, 16]. However, changing scores should, by definition, change the scores. More important is the question of whether those changes substantially affect the overall population. Consequently, we calculated Cohen's d (effect size) to understand the difference in the means between the original scores and the aggregate scores relative to the scores' standard deviation [4].

6.1.2 RQ2: Vulnerabilities. For RQ2, we looked at the relationship between the Scorecard scores, our Aggregated Scorecard Scores, and the number of vulnerabilities in a package. We include any vulnerability in the current version of the package itself or in any of its dependencies, as described in Section 5.3. We use the number of vulnerabilities as the dependent variable in linear regression to determine the relationship between vulnerabilities, the checks, and the aggregation methods. In this analysis, we standardized the number of vulnerabilities and dealt with outliers using the same approach as the OpenSSF Scorecard *Vulnerabilities* metric [32], capping the number of vulnerabilities at 10. We modeled the independent variables as individual checks, using interaction [11] to analyze the difference between the original values and the aggregate scores. Due to the size of our dataset, we selected $p < 0.01$ as our threshold for statistical significance.

6.2 Survey

In this section, we describe the methodology for our survey to address RQ3 Practitioners*: What approaches do practitioners recommend for aggregating measures across a component's dependency tree?*.

6.2.1 Survey Design. We constructed the survey such that for each metric, survey participants were asked whether they would prefer (a) the original score; (b) the aggregate score; or (c) another approach. If participants selected the third option, they were asked to fill in a free-response description of what approach they preferred. For each check, we also asked participants why they preferred the approach they selected as a free-response (open) question. At the end of the survey, we asked participants a series of background questions to understand their experience with open source software, software development, and security. An example survey can be found in our dataset [3].

We based our survey on five example packages. Our selection of example packages was based on (1) the package's popularity (i.e. the number of dependents); (2) whether the package had a dependency tree to analyze; (3) the package was not associated with a well-known organization like Microsoft or RedHat to avoid introducing potential bias due familiarity and potential trust of the organization. We also ensured that at least one of the examples was a dependency of the projects developed by our interview participants, to facilitate their potential participation in the survey.

6.2.2 Deployment. The survey was built and deployed using the Qualtrics tool [36], leveraging the built-in survey logic to present different dependencies to different respondents. The survey was developed in two iterations. An initial pilot survey was distributed within the research team. We then refined and clarified questions before deploying the survey. In the final deployment of the survey, the survey was open for 12 days from July 9-21, 2024.

6.2.3 Recruiting. To recruit participants, we identified dependents of the example packages. Additionally, the website for one of the packages included a list of other open-source projects that used the package. We excluded dependent packages that had one or more of the following concerns:

- the dependent package had been archived
- the dependent package was a fork of another package, since they may not have made the decision to include the example
- the dependent package name included the dependency to be reviewed, since there may have been no alternative

Using Google search engine, we searched for the organization or owner of the dependent's repository and the name of the dependent package to try to find contact information for maintainers of the repository. We used the contact information to try to recruit survey participants. We also contacted interview participants who were familiar with Go to request their participation in the survey. To further incentivize participation and thank participants, participants were entered into a drawing for a $20 Amazon gift card.

7 Results

In Section 7.1, we discuss the results of our data analysis for RQ1 *Difference* and RQ2 *Vulnerabilities*. In Section 7.2, we discuss the survey results for RQ3 *Practitioners*.

7.1 RQ1 & RQ2 - Data Analysis

In this section, we describe the results of our data analysis. In Section 7.1.1 we look at the difference our aggregation approaches make, addressing RQ1 *Difference*. Section 7.1.2 examines aggregation and vulnerability count in RQ2 *Vulnerabilities*.

7.1.1 Difference between Original and Aggregate Scores. One of the questions that we statistically evaluated is how our aggregation approaches make a difference in the scores. Specifically, *RQ1:* Difference*: What is the difference between the root package score and the aggregate score?* The results of this question are shown in Table 4.

First, we look at the number of packages where the score differs, which is the second column of Table 4. Since the score can never be higher than the root score, any differences in score between the original and aggregate scores will be decreases in score. Next, we look at whether the difference between the original and aggregate scores is statistically significant at $p < 0.01$ using the Mann-Whitney U-test with Bonferroni correction; the results are shown in column 3 of Table 4. We then leverage Cohen's d, a statistic commonly used for effect size, to understand the difference in average scores between the original and aggregate scores. We include the Cohen's d value in the fourth column of Table 4 alongside the interpretation of the Cohen's d statistic [4] in the fifth column.

Table 4: Effect of Adjusted Scores

	Pkg. w/ Score Decr.	p-value	Cohen's d (95% confidence interval)	Interpretation
Binary Artifacts	999	$< 2.2e^{-16}$*	-0.14 ± 0.02	Negligible
Branch Protection	2885	$< 2.2e^{-16}$*	-0.41 ± 0.02	Small
Code Review	4775	$< 2.2e^{-16}$*	-0.32 ± 0.02	Small
Fuzzing	201	0.06	-0.04 ± 0.02	Negligible
Maintained	2688	$< 2.2e^{-16}$*	-0.36 ± 0.02	Small
Security Policy	1106	$< 2.2e^{-16}$*	-0.27 ± 0.02	Small

**Statistically significant at $p < 0.01$ with Bonferroni correction*

The difference between the original and aggregate values was statistically significant for all checks except the *Fuzzing* check. This may be due to the relative scarcity of packages using Fuzzing tools that are picked up by this check. Only 306 of the Go repositories used the fuzzing tools identified by OpenSSF Scorecard. The effect size of the difference between original and aggregate values was negligible for the *Binary Artifacts* and *Security Policy* checks but higher for *Branch Protection, Code Review,* and *Maintained*.

7.1.2 Relationship with Vulnerabilities. Table 5 shows the logistic regression analysis for RQ2 Vulnerabilities*: How does the aggregate score change the relationship between security metrics and vulnerabilities?*. The first six rows show the relationship between each check and vulnerability count. Row 7 shows the cumulative effect of aggregating the scores, addressing the question "Does aggregation alter the relationship between vulnerabilities and checks?" without discriminating between the different checks. Rows 8 through 13 show the extent to which aggregating the score alters each check's relationship with vulnerability count.

Table 5: Regression of Checks and Adjusted Values against Active Vulnerability Count

	Coefficient	Std. Error	p-value
Binary Artifacts	< 0.001	0.001	0.661
Branch Protection	-0.003	0.044	< 0.001*
Code Review	0.003	< 0.001	< 0.001*
Fuzzing	0.013	0.001	< 0.001*
Maintained	0.004	< 0.001	< 0.001*
Security Policy	0.003	< 0.001	< 0.001*
Aggregate checks	0.007	0.016	0.665
Binary Artifacts: aggregate	−0.004	0.001	0.805
Branch Protection: aggregate	−0.001	0.003	0.877
Code Review: aggregate	0.003	0.001	< 0.001*
Fuzzing: aggregate	−0.001	0.002	0.735
Maintained: aggregate	0.003	0.002	0.122
Security Policy: aggregate	−0.006	0.003	0.026

**Statistically significant at $p < 0.01$*

As can be seen in Table 5, the *Binary Artifacts* check does not have a statistically significant relationship with vulnerability count, and aggregating the check does not alter the relationship in a statistically significant way. The *Fuzzing, Maintained, Security Policy*, and *Code Review* checks all have statistically significant, *positive* relationships with vulnerability count (higher check scores are related to higher vulnerability counts) as seen by the coefficient and p-values for each. The positive relationship, which has been observed in other work [42, 43], may seem counterintuitive if we consider vulnerability count as an indicator of risk. However, this positive relationship may be explained by the nature of the checks themselves. If someone is using a fuzzer, has a security policy for how to report vulnerabilities, uses code review, and generally keeps their package maintained and up-to-date, we would expect them to find vulnerabilities. If a maintainer is not following these practices, they are less likely to find vulnerabilities.

Only *Branch Protection* has a statistically significant, *negative* relationship with vulnerability count (i.e., lower *Branch Protection* is related to more vulnerabilities). If our previous assumptions are correct, this difference may be partly explained by the fact that *Branch Protection* is less directly linked to finding vulnerabilities.

Our aggregation methods only resulted in a statistically significant change to the relationship with vulnerability count for the *Code Review* check. In this case, our aggregation method is related to a further increase in the relationship between the *Code Review* score and vulnerability count.

7.2 RQ3 - Practitioner Survey

Section 7.2.1 provides details of the recruiting process and demographics of the participants. The primary results of our survey are described in Section 7.2.2. In Section 7.2.3 we discuss the reasons practitioners gave for *why* they selected a particular choice. An overview of the results of the survey is shown in Table 6

7.2.1 Survey Deployment and Recruiting. Following the recruiting methodology outlined in Section 6.2.3, we reviewed over 900 packages and contacted one or more maintainers at 22 organizations. Eight (8) individuals filled out the survey. Due to an error in the survey logic, one participant could not fill out the demographic information and did not reply to our follow-up request. For the other

7 participants, the average experience was *22.6 years*. The average security expertise on a scale from 0 to 4 was *3*. All participants had experience using and maintaining open-source software.

7.2.2 Survey Results. Table 6 shows the survey results. The first column of Table 6 lists the name of the metric. The second column shows the total number of respondents who were presented with the check. To respect participants' time, each respondent was presented with a randomly selected subset of the checks. Consequently, the total respondents varies between checks. The third column shows the number of respondents who indicated they preferred the original score. The fourth column shows how many respondents prefer our aggregate score. The final column shows how many participants indicated they would prefer some "other" score. Where practitioners preferred some "other" score, we summarize the scoring approach they indicated they *would* prefer.

As can be seen in Table 6, more practitioners preferred the original score for the *Code Review*, *Maintained*, and *Security Policy* checks. More practitioners preferred an aggregate approach, although not necessarily ours, for the *Binary Artifacts*, *Branch Protection*, and *Fuzzing* checks. In many cases, the number of respondents who preferred the original and aggregate approach was very close, and there was never unanimous agreement.

7.2.3 Respondent Rationale. For each check, we asked practitioners *why* they had made the previous selection. The reasons why practitioners preferred aggregate scores for the *Binary Artifacts*, *Branch Protection*, and *Fuzzing* checks were mixed, with no clear themes echoed by multiple participants. Participants had more consistent responses for checks where the majority of participants preferred the original score. For both *Code Review* and *Security Policy*, respondents who indicated they preferred the original score used statements such as "I usually just want to know the score of the top-level package", suggesting that **no** aggregation approach may be appropriate for these checks. However, for the *Security Policy* and *Maintained* checks, the measure implementation may have affected responses. For *Security Policy*, two participants indicated that the low adoption rate of SECURITY.md files and security policies influenced their response. For the *Maintained* check, two individuals who did not select the Aggregate score raised concerns that smaller packages may not have recent activity that would be detectable with a static analysis tool such as Scorecard.

One common theme across checks and preferences was the need to emphasize the root score over the dependencies. At least three participants indicated that the aggregate values did not place sufficient emphasis on the score for the root node. Additionally, one respondent indicated that they appreciated how the aggregation approach for Fuzzing provided "info about the dependencies' behaviors" while "focusing on the root". Similarly, at least three individuals indicated it would be nice to have both the original and aggregate scores for one or more of the *Binary Artifacts, Code Review, Fuzzing*, and *Security Policy* checks.

8 Discussion

In our statistical analysis, similar to Zahan et al [43], we found that checks reflecting practices associated with vulnerability detection, such as *Fuzzing*, had a positive relationship with the number of vulnerabilities in software. If we assume that higher values mean lower risk and that vulnerability count is a measure of risk, then these findings are counterintuitive. However, ***organizations with robust practices around detecting and reporting vulnerabilities are more likely to find vulnerabilities in their system.***

Although our survey results were mixed, for all checks at least one respondent preferred the aggregated metrics, and ***in three cases (Binary Artifacts, Branch Protection, and Fuzzing), more practitioners preferred an aggregated score*** over the original, root package score.

While some checks should be aggregated, ***aggregating the Code Review check may not be worth any additional computation*** that aggregation could incur. The aggregate value only increases the relationship with vulnerability count, which does not necessarily add value. Even the more limited aggregation approach we selected, which ignored indirect dependencies, was rejected by six of the seven survey respondents who were presented with this check.

9 Related Work

Third-Party Component Selection. Academic research [1, 2, 7, 21, 24, 26, 27, 39, 41] has used surveys and interviews with practitioners to identify the attributes and measures used to select open-source, third-party components. Their findings include metrics which are similar to the industry-developed measures of OpenSSF Scorecard, such as concerns around the *active maintenance* of a library identified by Vargas et al [21], which is reflected in the *Maintained* metric of OpenSSF Scorecard. We expand on prior work by further examining how to aggregate measures from packages in the dependency tree of a component.

OpenSSF Scorecard Scores. The academic research on OpenSSF Scorecard to date includes studies by Zahan et al. [42, 43]. In one work [43], they examine whether OpenSSF Scorecard checks can predict vulnerabilities in the PyPi (Python) and npm (JavaScript) ecosystems. They found that the *Branch Protection, Code Review, Maintained, Pinned Dependencies,* and *Security Policy* checks were good predictors. In another work, Zahan et al. [42] examine the distribution of Scorecard scores, the efficacy of the Scorecard tool, and the adoption of the practices captured by the Scorecard metrics in the npm and PyPi ecosystems. Similar to our work in the Go ecosystem, they found that the *Packaging* and *Signed Releases* checks had a high number of invalid scores. Our work focuses on a different ecosystem and examines how scores can be aggregated.

The Relationship Between Package Depth and Security Prior research examining security risk in a package's dependency tree primarily focuses on *vulnerabilities*. For example, Latendresse et al. [22] analyze dependency trees in the npm ecosystem, examining attributes such as how many dependencies are *production* dependencies[3], and which dependencies (production or non-production, direct or transitive) have more "npm-audit" security alerts, i.e. vulnerabilities. The authors found that most dependencies (52,403 out of 53,405) were only for non-production builds; and that most (308 out of 313) of the high- and critical-severity security alerts occurred

[3]For example, dependencies that are only used for testing as part of development and not included in a production build would NOT be considered *production* dependencies.

Table 6: Survey Results

Dark Grey text indicates an aggregate score was more preferred. *Red* text indicates the original score was more preferred

		Preference			
Metric	Total	Original	Aggregate	Other	Explanation(s) for Other
Binary Artifacts	7	2	4	1	Neither (Check Irrelevant)
Branch Protection	5	1	3	1	Check Irrelevant
Code Review	7	4	1	2	1) Neither (Check Irrelevant); 2) Weighted Average, root contributes 50% of total
Fuzzing	5	2	2	1	Min. of Entire Tree
Maintained	7	3	2	2	1) Neither (Check Irrelevant); 2) Both Preferred
Security Policy	6	3	1	2	1) Neither (Check Irrelevant); 2) Strong Preference for a Security Policy

in non-production dependencies. Our research expands on existing work by examining other measures of risk, beyond vulnerabilities.

Security Risks in the Go Ecosystem. Prior research has analyzed security-related defects in Go. Dunlap et al. [9] examined ways to use large-language models (LLMs) to identify fixes associated with vulnerabilities, through analysis of source code. Wang et al. [38] developed a static analysis tool for finding vulnerabilities in Go code. The authors summarize how CWEs may apply in Go, but consider the CWE list "not fully applicable to Golang" and develop their own system. Compared with Dunlap et al. [9], Wang et al. [38], and Hu et al [20] our work looks into additional security metrics, beyond vulnerabilities, that can be used to understand the strengths and weaknesses of the software development process.

10 Threats to Validity

We discuss the Threats to Validity of our approach in this Section. We group these threats based on Conclusion Validity, External Validity, Internal Validity, and Construct Validity [5, 13, 40].

Conclusion Validity. Conclusion Validity is about whether conclusions are based on statistical evidence [5, 40]. For our data analysis, we have sufficient information to draw statistically significant conclusions. However, for our survey, we did not have sufficient participation to perform statistical analyses. Statistically, this threat cannot be mitigated. Our conclusions are therefore tentative. However, for both the interviews and survey, we reached out to practitioners with experience in security and software development. As part of the survey recruitment process, we examined over 900 packages searching for participants from different projects. Expanding the survey to additional ecosystems in future work may help with broadening participation.

Our survey included an open, free-response question for the reasons why participants selected the original, aggregate, or other score. Our analysis of these results is informal coding by a single author, which limits our ability to draw strong conclusions from the results. However, the answers provided by survey participants provide us a starting point for considering future work to understand *why* our survey produced the results described in Section 7.2.2.

Construct Validity. Construct Validity concerns the extent to which the treatments and outcome measures used in the study reflect the higher-level constructs we wish to examine [5, 37, 40]. The OpenSSF Scorecard checks may not fully capture the concepts that they claim to represent. However, the tools are supported by several companies and industry organizations, and we are not aware of a more widely accepted alternative.

Additionally, our analysis depends on the accuracy of our dependency trees. Resolving a package's dependency tree is a non-trivial problem. We leverage the Open Source Insights dataset[18] dependency tree data, since the dependency trees in the Open Source Insights database only include production dependencies, as recommended by Paschenko et al. [34]. The maintainers of the Open Source Insights database claim to have tested their algorithms against "native" dependency tree graph resolution implementation for each language, and "given identical inputs, the results agree closely: 99% or higher, often much higher" [19].

Our current study focuses on vulnerability count as a security dependent measure. Other measures, such as exploitability [12] or time-to-fix [23], may provide additional information key to understanding the relationship between the OpenSSF Scorecard checks and software security. However, the examined dataset did not include the exploitability of each vulnerability in the context of each system or provide fix information. This limits our approach. Extraction and inclusion of these measures would be a valuable area of future work.

Internal Validity. Internal Validity concerns whether the observed outcomes are due to the treatment applied, or other factors [5, 13]. Our initial participants in the interviews were based on convenience sampling, which is known to produce bias [17]. However, these interviews were only used to provide initial insights, and were not part of the main methodology. As seen in our methodology and results for RQ3 (Sections 6.2 and 7.2) as well as many other academic studies, identifying industry practitioners willing to participate in research is time-consuming. To efficiently identify participants through other means would have required us to lower our expectations for the experience of interview participants. To help mitigate this threat, for the survey, we follow a more systematic approach, identify developers from relevant packages, and contact participants based on their public, online profiles.

Our decision to exclude low-risk checks based on preliminary feedback may have also biased our results. To help minimize the potential threat to validity, we analyzed the data for the low-risk checks and determined that all of the checks labeled 'low-risk' in the OpenSSF Scorecard had insufficient data and would therefore have been removed from our analysis for other reasons. Additionally, as noted in Section 3, the interviewees expressed concern over comparing low-risk checks against other security measures.

External Validity. External Validity concerns the generalizability of our results [5, 13, 40]. As discussed in Section 5, we excluded *packages* from the analysis where the main package, i.e., the root of the dependency tree, was missing one or more of the Scorecard

scores we were analyzing. Since we do not know, precisely, why each score is missing, we do not know if these packages could introduce potential bias or other factors into our work. However, excluding the packages increases the risk that our results are not generally applicable. For aggregation purposes, if a package had all available scores, but its dependencies do not, we exclude the dependencies from our aggregation as discussed in Section 4. Analysis of why the checks failed and altering our approach based on the analysis is an area of future work that could further improve our mitigation of potential bias and improve our confidence in the generalizability of our findings.

More broadly, our results may not generalize to other ecosystems. However, the Go language is used in important applications for companies such as American Express and Capital One [14]. Go has many characteristics similar to those of more popular languages, including type-safety and memory-safety.

Notably, participants in our preliminary interviews indicated that the Signed Releases and Packaging checks were less applicable for Go since packages are distributed via source code. As we see in Section 5.2.2, these two checks were excluded from later analysis due to a lack of data. These two aggregation approaches are particularly likely to require additional future work on other areas.

Finally, as discussed in Section 5.2.2, several checks were excluded due to lack of data as a result of the implementation of the checks in OpenSSF Scorecard and the construction of the BigQuery dataset, including the Signed Releases and Packaging checks mentioned in the previous paragraph. We exclude them from our analysis and from our discussion and conclusions to eliminate the threat to conclusion validity. However, we introduce a threat to external validity since we are limited in our ability to determine whether our findings generalize to checks that were excluded. As noted in the previous paragraph, at least two of the checks (Signed Releases and Packaging) may be difficult to examine at all within the Go ecosystem, making them difficult to examine within our scope. As discussed in Section 5.2.2, a third check, the SAST check, may have posed additional problems due to its current implementation. Further research examining the tool in other ecosystems and analyzing and evaluating improvements to the tool implementation would mitigate this threat to validity, but was outside our scope.

11 Ethical Considerations

The preliminary interviews and survey were performed following NCSU IRB Protocol 26676. This protocol was exempt from full board review since we are not collecting or reporting sensitive information. We obtained informed consent from all participants, structured our interview and survey to minimize the amount of personal information we collected, and anonymized all responses. We are not identifying new security weaknesses or vulnerabilities, and therefore do not have a vulnerability disclosure process.

12 Conclusion

Using a combination of data analysis and feedback from practitioners via interviews and surveys, we have determined that some metrics, such as the *Branch Protection* metric, may be more helpful to practitioners if information from throughout the dependency tree is provided. Others, such as the *Code Review* metric, may be sufficient if only the root package is evaluated. Future work may expand the study to examine how the Scorecard metrics relate to other measures of security, such as exploitability or time-to-fix. Similarly, further research is needed to fully understand how the information should be presented, such as whether presenting both initial and aggregate information provides too much information.

Acknowledgments

This work was supported and funded by the National Science Foundation Grant No. 2207008. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. We want to thank our colleagues in the Secure Software Supply Chain Center (S3C2) for valuable feedback.

References

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why Do Developers Use Trivial Packages? An Empirical Case Study on Npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 385–395. doi:10.1145/3106237.3106267

[2] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. 2020. On the impact of using trivial packages: An empirical case study on npm and pypi. *Empirical Software Engineering* 25 (2020), 1168–1204.

[3] Authors. 2025. Our dataset. https://osf.io/x9hzs/?view_only=482febb8d30141aea8abbe1f8ad461d8.

[4] Jacob Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences* (2nd ed.). Routledge.

[5] Thomas D. Cook and Donald T. Campbell. 1979. *Quasi-Experimentation: Design and Analysis Issues for Field Settings.* Rand McNally College Publishing.

[6] Russ Cox, Robert Griesemer, Rob Pike, Ian Lance Taylor, and Ken Thompson. 2022. The Go programming language and environment. *Commun. ACM* 65, 5 (apr 2022), 70–78. doi:10.1145/3488716

[7] Fernando López de la Mora and Sarah Nadi. 2018. An Empirical Study of Metric-Based Comparisons of Software Libraries. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering* (Oulu, Finland) *(PROMISE'18)*. Association for Computing Machinery, New York, NY, USA, 22–31. doi:10.1145/3273934.3273937

[8] Jay L Devore. 2014. *Probability and Statistics for Engineering and the Sciences* (ninth ed.). Cengage Learning.

[9] Trevor Dunlap, John Speed Meyers, Bradley Reaves, and William Enck. 2024. Pairing Security Advisories with Vulnerable Functions Using Open-Source LLMs. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Federico Maggi, Manuel Egele, Mathias Payer, and Michele Carminati (Eds.). Springer Nature Switzerland, Cham, 350–369.

[10] Olive Jean Dunn. 1961. Multiple comparisons among means. *Journal of the American statistical association* 56, 293 (1961), 52–64.

[11] Peter K. Dunn and Gordon K. Smyth. 2018. *Chapter 1: Statistical Models.* Springer New York, New York, NY, 1–30. doi:10.1007/978-1-4419-0118-7_1

[12] Sarah Elder, Md Rayhanur Rahman, Gage Fringer, Kunal Kapoor, and Laurie Williams. 2024. A Survey on Software Vulnerability Exploitability Assessment. *ACM Comput. Surv.* 56, 8, Article 205 (April 2024), 41 pages. doi:10.1145/3648610

[13] Robert Feldt and Ana Magazinius. 2010. Validity threats in empirical software engineering research-an initial survey.. In *Seke*. 374–379.

[14] Go. 2024. Why Go -> Case Studies (Website). https://go.dev/solutions/case-studies Online; Accessed: 22-July-2024.

[15] GO. 2025. Why Go > Case Studies. https://go.dev/solutions/case-studies.

[16] Jelle J Goeman and Aldo Solari. 2014. Tutorial in biostatistics: multiple hypothesis testing in genomics. *Statistics in Medicine* (2014).

[17] Jawad Golzar, Shagofah Noor, and Omid Tajik. 2022. Convenience sampling. *International Journal of Education & Language Studies* 1, 2 (2022), 72–77.

[18] Google. 2024. Open Source Insights (website). https://deps.dev/ Online; Accessed: 22-July-2024.

[19] Google. 2024. Open Source Insights (website). https://docs.deps.dev/faq/ Online; Accessed: 22-July-2024.

[20] Jinchang Hu, Lyuye Zhang, Chengwei Liu, Sen Yang, Song Huang, and Yang Liu. 2024. Empirical Analysis of Vulnerabilities Life Cycle in Golang Ecosystem. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 212, 13 pages. doi:10.1145/3597503.3639230

[21] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. 2020. Selecting Third-Party Libraries: The Practitioners' Perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 245–256. doi:10.1145/3368089.3409711

[22] Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, and Emad Shihab. 2023. Not All Dependencies Are Equal: An Empirical Study on Production Dependencies in NPM. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 73, 12 pages. doi:10.1145/3551349.3556896

[23] Xiaodan Li, Xiaolin Chang, John A. Board, and Kishor S. Trivedi. 2017. A novel approach for software vulnerability classification. In *2017 Annual Reliability and Maintainability Symposium (RAMS)*. 1–7. doi:10.1109/RAM.2017.7889792

[24] Xiaozhou Li, Sergio Moreschini, Zheying Zhang, and Davide Taibi. 2022. Exploring factors and metrics to select open source software components for integration: An empirical study. *Journal of Systems and Software* 188 (2022), 111255. doi:10.1016/j.jss.2022.111255

[25] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[26] Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. 2023. What are the characteristics of highly-selected packages? A case study on the npm ecosystem. *Journal of Systems and Software* 198 (2023), 111588. doi:10.1016/j.jss.2022.111588

[27] Sarah Nadi and Nourhan Sakr. 2023. Selecting third-party libraries: the data scientist's perspective. *Empirical Software Engineering* 28, 1 (2023), 15.

[28] OpenSSF. 2024. About - Open Source Security Foundation. https://openssf.org/about/ Online; Accessed: 08-Jun-2024.

[29] OpenSSF. 2024. OpenSSF Scorecard (Website). https://securityscorecards.dev [Online; Accessed: 30-Sept-2024].

[30] OpenSSF. 2025. OpenSSF Scorecard Checks Summary (Website). https://github.com/ossf/scorecard?tab=readme-ov-file#scorecard-checks [Online; Accessed: 23-Aug-2025].

[31] OpenSSF. 2025. OpenSSF Scorecard Checks (Website). https://github.com/ossf/scorecard/blob/main/docs/checks.md [Online; Accessed: 12-July-2025].

[32] OpenSSF. 2025. OpenSSF Scorecard README (Website). https://github.com/ossf/scorecard/blob/main/README.md [Online; Accessed: 09-July-2025].

[33] OSV. 2024. Open Source Vulnerabilities (Website). https://osv.dev/ Online; Accessed: 22-July-2024.

[34] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable Open Source Dependencies: Counting Those That Matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Oulu, Finland) *(ESEM '18)*. Association for Computing Machinery, New York, NY, USA, Article 42, 10 pages. doi:10.1145/3239235.3268920

[35] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A Qualitative Study of Dependency Management and Its Security Implications. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1513–1531. doi:10.1145/3372297.3417232

[36] Qualtrics. 2024. Qualtrics (Website). https://www.qualtrics.com/ Online; Accessed: 22-July-2024.

[37] Paul Ralph and Ewan Tempero. 2018. Construct validity in software engineering research and software metrics. In *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018*. 13–23.

[38] Cong Wang, Hao Sun, Yiwen Xu, Yu Jiang, Huafeng Zhang, and Ming Gu. 2019. Go-Sanitizer: Bug-Oriented Assertion Generation for Golang. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 36–41. doi:10.1109/ISSREW.2019.00039

[39] D. Wermke, J. H. Klemmer, N. Wöhler, J. Schmüser, H. Sri Ramulu, Y. Acar, and S. Fahl. 2023. "Always Contribute Back": A Qualitative Study on Security Challenges of the Open Source Supply Chain. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1545–1560. doi:10.1109/SP46215.2023.00191

[40] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[41] Bowen Xu, Le An, Ferdian Thung, Foutse Khomh, and David Lo. 2020. Why reinventing the wheels? An empirical study on library reuse and re-implementation. *Empirical Software Engineering* 25 (2020), 755–789.

[42] Nusrat Zahan, Parth Kanakiya, Brian Hambleton, Shohanuzzaman Shohan, and Laurie Williams. 2023. OpenSSF Scorecard: On the Path Toward Ecosystem-Wide Automated Security Metrics. *IEEE Security & Privacy* 21, 6 (2023), 76–88. doi:10.1109/MSEC.2023.3279773

[43] Nusrat Zahan, Shohanuzzaman Shohan, Dan Harris, and Laurie Williams. 2023. Do Software Security Practices Yield Fewer Vulnerabilities?. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 292–303. doi:10.1109/ICSE-SEIP58684.2023.00032

[44] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. 2022. On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empirical Software Engineering* 27, 5 (2022), 107.

[45] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem.. In *USENIX security symposium*, Vol. 17.

A Go Package Collection Validation

An overview of the package selection process is shown in Table 2. In this section, we discuss additional processes used to validate our selection criteria.

As discussed previously in Section 5.1, we begin with the Open Source Insights dataset, which lists 768,115 Go packages from the Open Source Insights database. We then map those packages to GitHub repositories using data stored in the Open Source Insights dataset, and remove all packages for which we could not locate a GitHub repo. This resulted in the removal of 26,088 packages, with 742,027 packages remaining. To focus on packages that were being used as third-party components, we narrowed our selection to packages that were dependents of other packages that were from a different organization. This further reduced the number of packages examined to 57,613. We then removed 13,385 packages for which we do not have dependency data. The resulting dataset contains 44,237 root nodes of dependency trees of third-party libraries.

We then validated the selection process due to the large number of packages removed. One researcher randomly selected 100 packages that did not have any dependents and 50 packages that only had dependents from the same organization, while a second researcher randomly selected 200 packages across both categories. The researchers found one package that appeared to be a library in use by practitioners; the remainder were not[4].

While we were able to map the package name of the root node for each of these 44,237 packages to a GitHub repository, we also need to be able to map each of their dependencies to a GitHub repository to map to OpenSSF Scorecard scores. We identified 34,617 packages for which we had a GitHub repository name for the entire dependency tree, and continued with the selection process as discussed in Section 5.1.

[4]For example, one of the libraries examined was a "toy" library designed to make fun of Kubernetes