Measuring Dependency Freshness in Software Systems

Joël Cox*, Eric Bouwers†, Marko van Eekelen*‡ and Joost Visser*†
*Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands
Email: joel@joelcox.nl, marko@cs.ru.nl and j.visser@cs.ru.nl
†Software Improvement Group, Amsterdam, The Netherlands
Email: e.bouwers@sig.eu and j.visser@sig.eu
‡Computer Science Department, Heerlen, Open University of the Netherlands
Email: marko.vaneekelen@ou.nl

***Abstract*—Modern software systems often make use of third-party components to speed-up development and reduce maintenance costs. In return, developers need to update to new releases of these dependencies to avoid, for example, security and compatibility risks. In practice, prioritizing these updates is difficult because the use of outdated dependencies is often opaque. In this paper we aim to make this concept more transparent by introducing metrics to quantify the use of recent versions of dependencies, i.e. the system's "dependency freshness".**

We propose and investigate a system-level metric based on an industry benchmark. We validate the usefulness of the metric using interviews, analyze the variance of the metric through time, and investigate the relationship between outdated dependencies and security vulnerabilities. The results show that the measurements are considered useful, and that systems using outdated dependencies four times as likely to have security issues as opposed to systems that are up-to-date.

I. Introduction

The use of third-party-components allows organization to develop systems at a lower cost in a shorter period of time [1]. Once a third-party component is added, it becomes a *dependency* of the system.

Throughout the lifetime of a system, developers need to invest time to update dependencies for a variety of reasons. For example, older versions of a dependency can contain security issues which can put the system using that version of a dependency at risk. Moreover, more recent versions of dependencies often include fixes related to the stability of the dependency. Lastly, using up-to-date versions normally makes it easier to upgrade the dependencies, which makes the entire system more flexible in terms of upgrading.

However, updating dependencies can come at a high cost [2], for example because manual testing is required to check for regressions after a dependency update. In addition, the team working on the software system may have little influence on the development process of the dependency, yet it relies on the developers of the dependency to provide non-trivial security and bug fixes [3]. Finally, bug fixing and adding features to a system is often prioritized over preventive maintenance. Dependency updates may contain trivial changes not directly affecting the system, which makes it hard to justify the update effort at the time of the release of the dependency.

Deciding to update the dependencies of a system is thus a double edged sword and requires a careful balance of the effort needed to update the dependencies and the benefits gained by updating. While there exist ways to estimate the costs of upgrading dependencies, there is currently no way to quantify the long-term benefits of upgrading. In this paper, we take one step towards this type of quantification by proposing a way to quantify whether the dependencies of a system are up-to-date.

Before defining a system-level measurement, we first define a measurement to quantify the "freshness" of a single dependency. In this paper, the term "freshness" is used to denote the difference between the used version of a dependency and desired version of a dependency. The freshness values of all dependencies, are aggregated to the system-level using a benchmark based approach.

This system level metric is validated along three axis. First, we investigate the assumption that up-to-date versions of a dependency are desired by investigating the relationship between known security vulnerabilities and the system-level dependency freshness. Secondly, the usefulness of the metric is evaluated by interviews with practitioners. Lastly, the variability of the metric is assessed to make sure the changes in the value of the metrics can be placed into context.

II. Background

Recent research acknowledges the importance of proper dependency handling, as it is the most common build problem [4]. This study investigates one of the aspects of dependency management, namely the updating of dependencies.

Our research is inspired by the concept of "update lag" as described by Raemaekers et al. [5] while researching API stability of popular open source libraries. In a later study it was concluded that this lag is slightly correlated with the amount of change introduced in a new version of a dependency [6].

To the best of our knowledge, no research has been done towards quantifying the freshness of dependencies. However, others have studied various aspects of the update cycle of dependencies.

For example, a longitudinal study of changes to the dependencies of a project has been performed by Businge et al. [7], in which several quality aspects of Eclipse plugins were analyzed. This analysis also involved the changes made to the external dependencies of the plugins through time. The change in dependencies is expressed in a churn-like metric, counting the number of dependencies added and removed.

Moreover, research has been done on the API usage of dependencies, for instance to automate the migration between versions of a dependency [8]. Migration patterns between dependencies that provide comparable functionality were also studied, showing how dependencies are swapped within a system when the used dependency is no longer to be found suitable [9].

More closely related to our topic, Mileva et al. [10] performed an analysis of different version of external dependencies so see which version was the most popular. The "wisdom of the crowd" is used to determine the "best" dependency version, categorizing some of the dependency users as earlier adopters or late followers. They also observed the processes of migrating back to an old version of a dependency in case of compatibility issues.

III. Problem statement

This study is performed at the Software Improvement Group (SIG), an independent advisory firm that evaluates the quality of software systems. Our study builds upon previous research and practices developed at SIG. Based on this context we define the goal of our study following the guidelines of Basili [11]:

> To quantify the dependency freshness of a given software system from the point of view of external quality evaluators.

In the context of our research a dependency is a third-party component that is added and directly called by a system. Given our goal we define the following research questions to guide our research:

RQ1 How can we measure the dependency freshness of a single dependency?

RQ2 How can we measure the dependency freshness of a system as a whole?

The remainder of this paper is structured as follows; in Section IV we investigate different dependency-level metrics to quantify the difference between two versions of a dependency. Based on the values of these metrics in practice (Section V), one metric is selected to be aggregated to the system level in Section VI. The different validation studies are presented in Section VII, Section VIII discusses the results, after which Section IX concludes.

IV. Dependency freshness at the dependency-level

In this work we define *dependency freshness* as the difference between the currently used version of a dependency, and the version of a dependency the system would ideally use. As discussed, the "ideal" version of a dependency can be context-specific; one can choose for the most stable version, a specific long-term support version, or the latest version of a dependency. For our study, we equate the ideal situation with using the latest version of the dependency. In other words, as soon as a new version of a dependency is released this version should be used. Although we make this assumption here, the defined metrics are also applicable to other definitions of the "ideal" version.

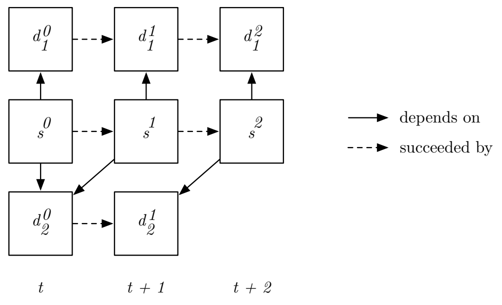


Fig. 1. Example of a system with two dependencies. Dependency d_1 is kept up-to-date with every version of system s, while the system only updated to dependency d_2^1 in system version s^2

As an example of our situation consider Figure 1 which shows a system s depending on two dependencies d_1 and d_2. On the top we see that dependency d_1 is kept up-to-date with every release of s, while the new version of d_2 was not immediately used in s. Thus at time $t + 1$ the freshness of d_1 is higher than d_2. In the second release of s the new version of d_2 is used, making the freshness of both dependencies equal again.

In this section we first lay out the criteria for selecting an appropriate metric given our research context (Section IV-A). We then define three different component-level metrics (Section IV-B, IV-C and IV-D) and test them to the criteria we explained earlier (Section IV-E).

A. Metric criteria

To select appropriate metrics for our research context we consider the criteria defined by Heitlager et al. [12]. These criteria are the minimal requirements that most be fulfilled in order to result in a practical metric model.

Technology independent A metric should be applicable to a wide range of systems and programming languages.

Ease of implementation The definition of a metric should be straight forward, easy to compute, and implement.

Simple to understand The metric should be easy to explain to non-technical staff and management.

Enable root-cause analysis Causative relations between system properties and quality factors should be clear.

To satisfy the criteria above we set out to define three relatively simple metrics that only take into account properties of a dependency release, rather than more complex metrics that take into account how a dependency interacts with a system. In the next section we describe the following metrics and their considerations: *version sequence number* (Section IV-B), *version release date* (Section IV-C), and *version number delta* (Section IV-D).

TABLE I
EXAMPLE OF HOW TO COMPUTE THE VERSION NUMBER DELTA DISTANCES BETWEEN SEVERAL VERSIONS OF A DEPENDENCY.

Version	Version number	Delta	Cumulative delta
d^n	$(1,2,0)$		
d^{n+1}	$(1,2,1)$	$(0,0,1)$	$(0,0,1)$
d^{n+2}	$(1,3,0)$	$(0,1,0)$	$(0,1,1)$
d^{n+3}	$(1,3,1)$	$(0,0,1)$	$(0,1,2)$

TABLE II
COMPARISON OF THE DIFFERENT DEPENDENCY-LEVEL FRESHNESS METRICS. CRITERIA CAN EITHER BE ADHERE TO (+), OR NOT (–).

	Version sequence number	Version release date	Version number delta
Technology independent	+	+	–
Ease of implementation	+	+	+
Simple to understand	+	+	–
Enable root-cause analysis	+	+	+

B. Version Sequence Number

The difference between two separate versions of a dependency can be expressed by the difference of the version sequence numbers of two releases. This measurement does not necessarily take into account the version number of a dependency, but can also employ the release date of the dependency to order difference versions. I.e., for a dependency with the versions (d^n, d^{n+1}, d^{n+2}) ordered by release date, the version sequence distance between d^n and d^{n+2} is 2.

Consideration: Dependencies with short release cycles are penalized by this measurement, as the version sequence distance is relatively high compared to other dependencies.

C. Version Release Date

The distance between two releases of a dependency can also be expressed by the number of days between the release dates. I.e., let r be a function which returns the release date for a dependency version. The distance between $r(d^n) = 10/3/2014$ and $r(d^n + 2) = 30/6/2014$ is defined as 113 days. Unlike the other measures presented in this section, this measurement can be calculated without knowledge of intermediate releases.

Consideration: This measurement heavily penalizes dependencies that release a new version of a dependency after large periods of inactivity. This is often the case for dependencies with a high level of maturity.

D. Version Number Delta

Lastly, the distance can be computed by comparing the version numbers of the releases of a dependency. Comparing two version number tuples can be done by calculating the delta of all version number tuples between two releases. A version number is defined as a tuple (x, y, x) where x signifies the major version number, y the minor version number and x the patch version number. The function v returns the version numbers tuple for a version of a dependency.

The delta is defined as the absolute difference between the highest-order version number which has changed compared to the previous version number tuple. To compare multiple consecutive version number tuples, the deltas between individual versions are added like normal vectors.

For example, two consecutive versions of a dependency $v(d^n) = (1,2,2)$ and $v(d^{n+1}) = (1,3,0)$ results in the version delta distance $(0,1,0)$. A more elaborate example can be found in Table I.

Consideration: The main problem with this measurement is that there is no meaningful way of aggregating the tuple of major, minor and patch version numbers to come to with a single number to represent the version delta. Intuitively it can be said that $x > y > z$, but it is impossible to generalize the values of the variables as they are completely dependent on a dependency's individual release cycle.

E. Measurement overview

Table II shows how each of the three metrics adheres to the different criteria. The version number delta adheres to only two criteria. The reliance on a specific versioning scheme makes this metric technology dependent, while the additional aggregation step makes this metric harder to understand.

The other two metrics satisfy all four criteria and do not make any assumptions about versioning schemes of the software. Because of this, the version release date (VRD) and version sequence number (VSN) metrics are selected as candidate metrics in our study.

V. DEPENDENCY FRESHNESS IN PRACTICE

Although a high level of freshness is desired for a dependency, our experience tells us that in practice dependencies are not updated frequently. Like many other software system properties, we expect the distribution of freshness to follow a fat-tailed power-law distribution [13].

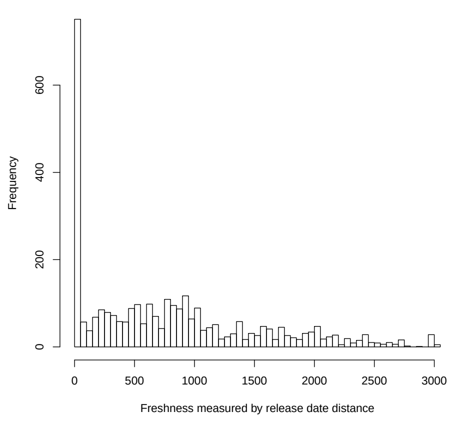
To asses this hypothesis the freshness information from a large group of systems is needed. To calculate this information two datasets are required, one containing the dependency information of a set of systems (Section V-A), and one containing the versions available for the used dependencies (Section V-B).

A. Dependency Information Dataset

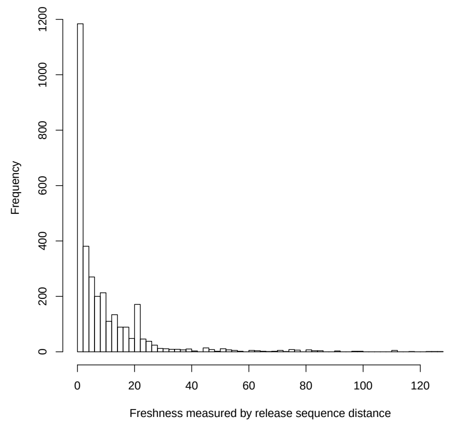
Out of the systems analyzed by SIG we selected 75 systems from 30 different clients. All of these systems were identified as Java systems that manage their dependencies through Maven[1], a tool for managing software dependencies. We chose the Maven package manager as it is the most prevalent package manager in the repository of industry systems.

Maven requires dependencies to be specified in a manifest file (`pom.xml`), located in the root directory of a system. Additional manifests files can be placed in subdirectories. To

[1]http://apache.maven.org



(a) Freshness by version release date.



(b) Freshness by version sequence number

Fig. 2. Distribution of dependency freshness at the dependency-level.

obtain a list of dependencies for a specific system we therefore need a two-step process:

1) Search the system for files called `pom.xml`.
2) Parse the XML files to retrieve the dependencies and remember the dependency if the dependency wasn't already specified for this system.

Variable version numbers are replaced by the value found for the key in the `<properties>` element. Dependencies with undeclared version numbers are discarded. This was only the case for 34 unique dependencies in our dataset.

A total of 3107 unique dependencies were retrieved from the dataset of industry system, consisting out of 8718 unique dependency versions. Of these dependency versions 5603 (64%) were classified as internal dependencies unavailable to us. A dependency is considered internal if the `groupId` of the dependency is equal to the `groupId` of the system.

B. Dependency Versions Dataset

To construct the database of dependency versions, e.g. the version number and release dates, the Maven.org repository is used. This repository is the most widely used repository for Java packages and used by the Maven package manager in its default configuration. For every unique dependency (combination of `groupId` and `artifactId`) found in our dataset of industry systems, a query was made to the Maven.org repository. If an exact match was found, all versions of that unique dependency were added to our database, together with their respective release dates.

Querying the Maven.org repository resulted in 2326 unique dependency versions with release dates out of 3115, a hit percentage of 75%. A total of 23431 unique intermediate dependency versions were retrieved. These are the dependency versions which were not found in the dataset of industry systems, but are earlier or later versions of unique dependencies which were found in this dataset.

C. Dependency Freshness distribution

Based on the datasets described above the dependency freshness in terms of the release date and release sequence was calculated, Figure 2 shows the results of this calculation.

When analyzing Figure 2 (a) the length of the tail stands out, showing that some dependencies are over 3000 days old. Upon further investigation these cases often involve rather mature dependencies such as those maintained by the Apache Commons project[2].

An example of such a dependency would be the release of the `commons-logging. commons-logging` package, version 1.1.1 on 2007-11-26. This release was followed by version 1.1.2 on 2013-03-16. Calculating the release date distance between the two versions would yield a number found in the tail of the overall distribution, while the actual perceived freshness is probably deemed to be higher.

The release sequence histogram (Figure 2 (b)) follows the anticipated power-law distribution more accurately. This measurement is more forgiving to mature dependencies which are updated after several years of inactivity, as described in the previous paragraph. However, it is more susceptible to dependencies which have a shorter release cycling. For instance, the `org.eclipse.jetty.jetty-client` package saw 128 releases in a timespan of less than 5 years.

Performing a Spearman correlation test between both metrics yields a value of 0.637, indicating a strong, significant ($P < 0.05$) correlation. This result confirms that as the version release date distance increases, it is highly probable that the version sequence number distances increases, too (and vice versa).

When looking at the overall state of dependency freshness using the version sequence number metric we conclude that only 16.7% of the dependencies display no update lag at all; the most recent version of a dependency is used. Over 50% of the dependencies have an update lag of at least 5 versions, which is considerable. The version release date distance paints an even worse picture. The large majority (64.1%) of the dependencies has an update lag of over 365 days, with a tail up to 8 years. Overall we conclude that it is not common practice to update dependencies on a regular basis.

While both measurements are sensitive to outliers, we decided to use the version sequence number distance as

[2]http://commons.apache.org

the basis for our system-level metric, as we rather discount dependencies on fast release cycles (and thus more subject to change), rather than mature projects that receive potential minor updates.

VI. Dependency freshness at the system-level

Given the measurement of freshness on the dependency level, a system level metric can be defined by aggregating the lower level measurements. As discussed before, there is no absolute threshold available for the "ideal" freshness of dependencies. Therefore, we opted for a benchmark based aggregation approach. Given the power-law distribution of the data (see Figure 2) we use the risk based aggregation approach of Alves et al. [13][14].

This aggregation method works with a so-called *risk profile* which, in our case, describes which percentage of dependencies falls into one of four risk categories. To determine the thresholds for the risk profile a two-phase calibration process needs to be followed. In the first phase the thresholds for the four risk categories are derived from a benchmark (Section VI-A), this enables the creation of a risk profile on a per system basis.

In the second phase, thresholds are established to assign a rating to each risk profile (Section VI-B) based on the percentage of dependencies in each risk category. This allows the risk profiles to be sorted, which enables the comparison between systems.

A. First-level calibration

In the first-level calibration phase the thresholds for the risk profiles are determined. Based on these thresholds, each dependency is assigned to one of four risk categories. For example, if the thresholds are defined as $\langle 5, 10, 20 \rangle$ for the categories "low", "moderate", "high", and "very high", a dependency with a freshness of 9 would be assigned to the "moderate risk" category. Performing this classification for all dependencies of a system can result in a risk profile like $\langle 27, 9, 5, 2 \rangle$, meaning that a system has 27 "low risk", 9 "moderate risk", 5 "high risk" and 2 "very high risk" dependencies.

The threshold values for this classification are determined by plotting a cumulative density function, containing all dependency-level freshness measurements across a dataset. Figure 3 shows the overall distribution of dependency freshness as well as the distribution for each system extracted from the dataset as described in Section V-A. Three red lines are plotted to indicate the 70th, 80th and 90th percentile, which are used to derive the values of the thresholds for the risk categories. The exact numeric thresholds corresponding to the quantiles are listed in Table III.

One additional decision that needs to be made in this phase is whether each dependency is of equal importance, e.g. whether each dependency is assigned the same weight or whether some dependencies need to be weighted differently. Following the advice of Alves et al. [13] we assign each dependency a weight of 1. Determining whether a more elaborate weighting scheme provides more value is left as future work.

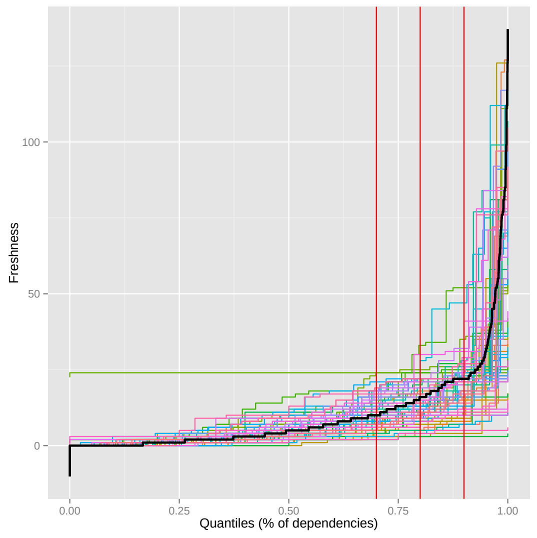


Fig. 3. Dependency freshness per system, measured by release sequence distance. The black line indicates the average. Red vertical lines are placed on the 70th, 80th and 90th percentile.

TABLE III
First-level thresholds for dependency risk profiles.

	Risk category			
Risk category	Low	Moderate	High	Very high
Interval	$[0, 10)$	$[10, 16)$	$[16, 22)$	$[22, \infty)$

After establishing these thresholds, risk profiles for each separate system can be created. As explained, the classification of a dependency is done by computing the version release sequence distance, after which the corresponding risk category is determined. Given the current thresholds, a dependency with a version release sequence distance of 12 is classified as a "moderate risk" dependency. This process is repeated for each dependency with a known version release sequence distance and results in a system-level risk profile.

B. Second-level calibration

While the risk profiles provide an indication of the distribution of undesirable dependency versions for a single system, it is difficult to intuitively compare them across systems. For example, given two relative risk profiles (containing percentages instead of absolute counts) $A = \langle 20, 45, 30, 5 \rangle$ and $B = \langle 15, 55, 22, 8 \rangle$ it is unclear which one should be preferred.

To solve this issue, Alves et al. [14] proposed an algorithm for ranking these risk profiles as well as a way to derive a rating from such a profile. To accommodate this, a mapping has to be created that assigns a rating based on the relative size of the risk categories. This rating scale can be completely arbitrary. However, to embed the rating into our research context we follow the already used approach by defining a 5 star rating system using a $\langle 5, 30, 30, 30, 5 \rangle$ distribution. In other words, 5% of the systems is assigned a rating of 1, 5% of the systems is assigned a rating of 5, while the other 90%

TABLE IV
NORMAL RISK PROFILES ARE TRANSFORMED INTO CUMULATIVE RISK PROFILES BY ADDING THE DEPENDENCIES OF HIGHER RISK CATEGORIES.

	Risk category			
Risk profiles	Low	Moderate	High	Very high
Non-cumulative	62.2%	20.7%	13.3%	3.8%
Cumulative	100%	37.8%	17.1%	3.8%

TABLE V
SECOND-LEVEL CALIBRATION THRESHOLDS FOR CUMULATIVE RISK PROFILES.

Rating	Moderate risk	High risk	Very high risk
⋆⋆⋆⋆⋆	8.3%	0%	0%
⋆⋆⋆⋆	30.4%	14.3%	7.7%
⋆⋆⋆	38.9%	30.6%	19.7%
⋆⋆	60.0%	46.3%	27.8%

is equally divided over the 2, 3, and 4 ratings. In this system, higher ratings are given to systems with less risk, e.g. more dependency freshness.

The first step to aggregating these risk categories is transforming the risk profile into a cumulative risk profile. This is done by including the dependencies of higher risk categories into the original risk category. Intuitively this makes sense; dependencies which are of "very high risk" are of at least "high risk", too. An example of this transformation is shown in Table IV.

Once the risk profiles are transformed, exact threshold values for the categories have to be set. Using the set of systems in the benchmark and their associated risk profiles, thresholds are computed so that our desired distribution of $\langle 5\%, 30\%, 30\%, 30\%, 5\% \rangle$ of system ratings is reached.

To assign a discrete rating to a risk profile the rating has to be found for which no cumulative risk category from a risk profile is greater than the threshold. It is also possible to assign a more fine-grained rating to a risk profile by using linear interpolation. This is done by first computing the discrete rating of a system and subtracting the absolute volume of dependencies in this risk category, normalized by the length of the risk category's interval. The lowest value of this computation for each risk category is taken and 0.5 is added for ease of arithmetic rounding when the rating is translated to a star rating.

As an example, the risk value in Table IV would be assigned four stars for the "very high risk" rating, three stars for the "high risk" rating, and three stars for the "moderate risk" rating. Taking the lowest of these ratings results in a rating of a maximum of three stars. Interpolating the moderate risk category gives $(38.9 - 37.8)/(38.9 - 30.4) = 12$, resulting in an interpolated score of $2.5 + 0.12 = 2.62$.

VII. VALIDATION

To validate the system level freshness rating three different types of studies are defined. First of all, we asses the relationship between dependency freshness and security vulnerabilities found in dependencies (Section VII-A). A positive relationship (e.g. less vulnerabilities are found in newer dependencies) would strengthen the argument that upgrading libraries is important. Secondly, we interview practitioners to determine the usefulness of the system level metric (Section VII-B). Finally, we perform a longitudinal analysis to analyze whether the metric can be used for long-term monitoring (Section VII-C).

A. Vulnerabilities in dependencies

To determine the relationship between the dependency freshness rating and security vulnerabilities we calculate the rating for each system and determine how many of the dependencies used by a system have a known security vulnerability. To determine whether a specific dependency versions contains a security vulnerabilities we use data obtained from a Common Vulnerabilities and Exposures (CVE) system. Component maintainers use these systems to spread security advisories to the users of their software.

1) Data Gathering: Because CVEs are mostly unstructured, the data obtained from the CVE system has to be processed before it can be matched to a specific dependency. In this study the dataset from Cadariu [15] is used in which dependencies found in the Maven.org ecosystem were matched with CVEs using the DependencyCheck[3] tool.

This dataset contains information about which version of a dependency is linked to a CVE identifier and is formatted as follows:

```
org.apache.wicketwicket6.5.050484787010570
26157.txt (cpe:/a:apache:wicket:6.5.0) :
CVE-2013-2055
```

The first part of each line is the path to a file, containing the `groupId`, `artifactId`, version number, and a random number. The token between parentheses is the name of the matches software package according to the CPE naming convention, while the last token references the CVE-ID which was matched.

It is clear that this data is not formatted in such a way that is can be incorporated in the dependency version dataset directly. The following steps were performed to recreate the fully qualified name of the dependencies in the dataset.

1) Find the longest repeating token in the file path. This would be "wicket" in the example.
2) Split the file path after this token, resulting in the dependency's `groupId` "org.apache.wicket" and `artifactId` "wicket6.5.05048478 701057026157.txt".
3) Find the version number either from the CPE definition, or perform a pattern match on the file path. In the example the version number can be taken from the CPE definition: "6.5.0".
4) Strip the version number from the `artifactId`, resulting in"wicket".

By using this algorithm we were able to fully qualify 1642 out of 1683 items from the original dataset, the remaining 41 records were discarded to keep the steps reproducible. After manual inspection 339 dependency versions were marked as containing at least one reported security vulnerability, this

[3]https://github.com/jeremylong/DependencyCheck

TABLE VI
SYSTEM VULNERABILITY DATA GROUPED BY STAR RATING.

Rating	No vulnerable dependencies	Vulnerable dependencies
⋆⋆⋆⋆⋆	2	1
⋆⋆⋆⋆	11	3
⋆⋆⋆	18	4
⋆⋆	11	15
⋆	4	2

low number is attributed to a high number of duplicate and mismatches in this used dataset as well as noise in the CVEs themselves. An example of such a mismatch is a non-Java project, matched to a Java project with comparable names.

Analysis result

Figure 4 (a) shows the distribution of dependency freshness grouped by the number of security vulnerabilities found. The main observation is that the majority of the systems were found to have no reported vulnerabilities. The data becomes increasingly sparse as the number of vulnerable dependencies increases.

The box plot shows that systems with a high median dependency freshness rating have a lower number of dependencies with reported security vulnerabilities. Systems with a low median dependency freshness rating have a higher number of vulnerable dependencies. However, due to the sparsity of the data the statistical tests do not show statistically significant results. To make statistically significant claims about the data the results have to be aggregated further.

Figure 4 (b) shows the distribution of systems without vulnerable dependencies and with vulnerable dependencies. This plot clearly shows the shift in distribution, which is significantly different (Wilcoxon rank-sum test, $W = 25$, $n = 71$, $P < 0.05$ two-sided). Systems with vulnerable dependencies have a mode of 2.2 and systems without vulnerable dependencies have a mode of 3.3. The mode is used to describe the central tendency of the distributions as they are clearly skewed.

When the data is tabulated (Table VI) and grouped by the star rating a tipping point can be observed between two and three stars systems. The distinction that can be made here is that dependencies with three or more stars are considered fresh, while others are considered stale, from a security perspective. A change in non-vulnerable versus vulnerable ratio can also be seen for systems rating two stars, although this can not be observed for one star systems. This can be attributed to the small number of systems in this group.

After making this subdivision, the effect size can be expressed through an odds ratio by simple cross-multiplication: $\frac{(2+11+18)/(1+3+4)}{(11+4)/(15+2)} = 4.3$. This effect size means that systems which score less than three stars are more than four times as likely to have vulnerable dependencies.

B. Interviews on Usefulness

Five semi-structured interviews were conducted with five technical consultants at SIG. Technical consultants support general consultants on client projects and develop the tooling to perform their analysis. Technical consultants are generally highly educated (Master degree or higher) and experienced software engineers. All 30 minute interviews were conducted by the first author at the SIG premises.

The technical consultants were asked about two types of systems, known and unknown systems:

Known system A system from a client that the technical consultant is assigned to. The exact system was selected by the first author from the set of systems assigned to the consultant which contain at least 20 dependencies.

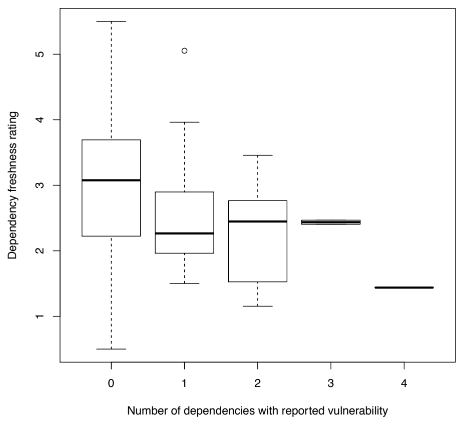
Unknown systems Five systems from the dataset of industry systems, with a similar number of dependencies and distinctive star ratings.

Printouts containing the dependencies and version distance of each dependency were brought to the interviews. Each line contains the `groupId` and `artifactId` of the dependency, together with the version number, followed by the component-level dependency freshness. The last line shows the total amount of dependencies in the system, the amount of dependencies that were classified as internal, and the number of dependencies with unknown histories (e.g. not found in the central Maven.org repository). The following is an example of such a printout.

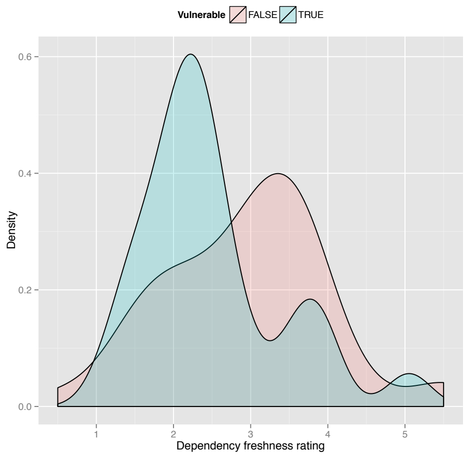
```
- commons-lang.commons-lang at 2.6.0 with
dependency freshness 0
- org.springframework.spring-web at 3.2.3
with dependency freshness 10
- javax.mail.mail at 1.4.1 with dependency
freshness 11
Found 58 dependencies, 15 with unknown
histories, 10 internal-only.
```

We asked the following questions to each of the interviewees. (The printout of the known system was only provided after question 5).

1) Are you aware of any issues – current or past – related to dependency management in the known system?
2) Is the development team responsible for the known system aware of issues arising from bad dependency management?
3) Do you have an idea about the dependency freshness of the known system?
4) If you were to rate the known system – considering the $\langle 5\%, 30\%, 30\%, 30\%, 5\% \rangle$ distribution – what rating would you give?
5) If you were to rank the following unknown systems from a low to high dependency freshness rating, how would you rank them?
6) Do you think this is a valuable metric considering its impact on security, stability and flexibility of the known system?
7) Would this metric translate to an advice to the customer?
8) Would you take any specific action given the printout of the known system?



(a) Dependency freshness rating by number of dependencies with reported vulnerabilities.



(b) Dependency freshness rating for systems with and without reported vulnerabilities.

Fig. 4. Dependency freshness rating and dependency vulnerabilities

TABLE VII
SYSTEMS AS RANKED DURING THE INTERVIEWS AND RANK PRODUCED BY THE DEPENDENCY FRESHNESS METRIC.

System			**Subject**				
#	**Rating**	**Rank**	**1**	**2**	**3**	**4**	**5**
1108	5.053	5	5	5	5	5	5
1994	4.105	4	4	**2**	4	4	4
850	3.248	3	3	3	3	3	3
362	2.188	2	2	**4**	2	2	2
181	1.427	1	1	1	1	1	1

Quantitative Interview Results

During the interview interviewees were presented with 5 printouts of systems and asked to rank these systems from a low-level to a high-level of dependency freshness. Printouts were fully anonymous; only a unique identifier was displayed on the paper. The results of this test are shown in Table VII. Numbers in bold indicate systems ranked differently than the metric's ranking.

As shown in Table VII, all interviewees except for one ranked the systems identical to the metric, which indicates a relative good agreement between the consultants and the rating from the metric.

Qualitative Interview Results

Several of the technical consultants consider dependency management an issue within the projects they evaluate. This is especially true for older systems with a relative large number of dependencies. Organizations with critical application and rigid processes often impose restrictions on the dependencies that can be used by the (internal) development team, either at the library level or runtime environment.

Some consultants were not aware of any issues. They either attributed the absence of problems to the volume, maturity or environment of the system. Smaller and newer systems may have a higher dependency freshness because when dependencies are added to a system, the most recent version of the dependency is included. Internal systems often have lower security requirements and system owner thus may find dependency management not too important from a security perspective.

The consultants were able to accurately rate the system, as shown in question 4 and 6 – in the proximity of one star. This indicates that the rating is quite well aligned with the perceived dependency freshness of the system. It also shows that the version distance is a good mapping from individual dependencies to the system dependency freshness, strengthening the predictive quality of the metric.

Some of the consultants were able to reason about the dependency freshness using their knowledge of the system, for instance the system age and owner. A few had actually inspected the system dependencies earlier, by hand.

When the consultants where asked to rank the unknown systems, different issues and questions came up:

- Are dependencies that are used more often throughout the system weighed differently?
- Are dependencies of different types weighed differently? For instance Hibernate (a database abstraction layer) may be more crucial than JUnit (a testing framework).
- How are dependencies on fast release cycles weighed?
- Are transitive dependencies taken into account?

These questions were not answered until after the ranking was finalized. In spite of these concerns, the consultants were able to accurately rank the system, as shown in Table VII.

All interviewees were favorable towards the usefulness of the metric. Some saw the metric as an indicator of the client's or the supplier's work processes, or the quality of their work processes. One consultant elaborated on how the metric served as an indicator of security, stability and flexibility, ultimately agreeing with all three aspects.

Throughout the interviews the importance of good dependency management was stressed, but every interviewee also acknowledged how difficult this really is. “This is a typical problem developers know about, but there is no clear overview of the problem and it only becomes apparent when a new feature has to be implemented and x has to be updated”, was noted by one of the interviewees.

C. *Longitudinal Analysis*

To perform a longitudinal analysis the dependency freshness rating of all available versions of systems in our dataset was calculated. Systems with fewer than five versions were not included, reducing the dataset to 50 systems. No restrictions were placed on the interval between versions of a system.

Analysis Results

The median variance of the rating for the different systems is quite low, only 0.04 in a range of 5. Only nine systems have a variance of 0.2 or greater, meaning that the majority of the systems sees little changes to its dependency freshness rating. When excluding the systems without dependency churn the median variance improves only slightly, to 0.06.

Although the rating variance is low, quite some change can be observed when looking at Figure 5. Snapshots that include dependency updates are clearly visible in the line plot, displayed as an ascending line. If a snapshot contains no updated dependencies the line is stable, or slightly declining if new versions of the dependencies used in the system were released. The following types of systems with regards to dependency freshness can be distinguished:

Stable Systems with a stable dependency freshness rating. The system dependencies see little to no updates.

Improving Systems with an increasing dependency freshness rating. Dependencies are updated faster than they are released.

Declining Systems with an decreasing dependency freshness rating. Dependencies are updated slower than they are released.

VIII. Discussion

The reported security vulnerabilities study shows that systems with a low dependency freshness score are more than four times as likely to contain security issues in these dependencies. This confirmed the relationship between the security quality factor of a system and its dependency freshness rating, although no claims of causation could be made.

All interviewees considered the system-level metric useful and considered that it also serves as an indicator for several other properties of the system and it’s development process. We found that technical consultants are able to accurately distinguish systems with a high level and low level of dependency freshness. Moreover, the metrics showed to be good mapping for the perceived dependency freshness of a system.

While the longitudinal analysis showed a low variance in ratings, the analysis shows an interesting concept, namely that the dependency freshness of a system can be characterized using two dimensions: the speed at which the dependencies are updated and the speed at which the dependencies are released.

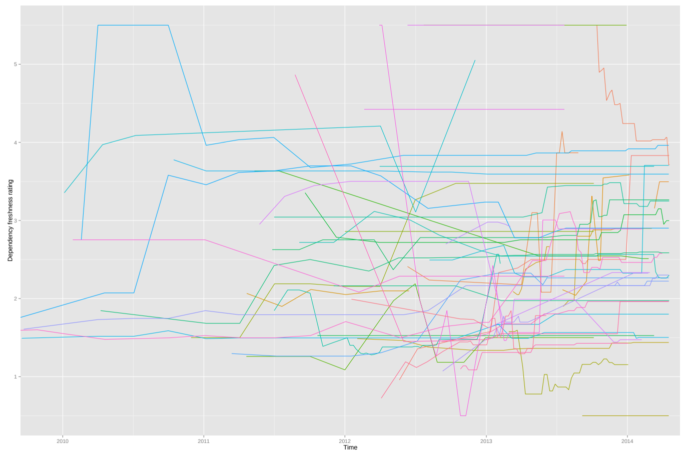


Fig. 5. Dependency freshness rating per snapshot grouped by systems with more than 5 versions.

In turn this shows that good dependency management (i.e. a high dependency freshness rating) can be achieved by making the correct combinations of these dimensions. Systems which use dependencies with a short release cycling thus require a high update speed in order to achieve a high rating, while systems with dependencies on a long update cycling can get away with a slower update speed. Using more mature (i.e. dependencies on a long update cycle) dependencies thus requires less effort to maintain a certain dependency freshness rating. On the other hand, systems with a dependency freshness rating that has been stable for an extensive period of time might also be an indicator of problems.

A. *Treats to validity*

A relationship has been found between the dependency freshness rating of a system and whether this system has a dependency with a security vulnerability. However, this does not imply causation as counter examples can also be found: systems with a higher dependency freshness can also have security issues. This should not be surprising, as newer software sometimes also contain reported security vulnerabilities.

Additionally, the data obtained for determining whether dependencies contained reported vulnerabilities was generated automatically, which has been shown to be difficult to do accurately [15]. Because the actual names of the dependencies are matched to a known set of systems, there is a high level of confidence that the resulting data set is correct (no false positives), but not complete. For instance, popular Java projects like Spring and Hibernate are not included. It is expected that adding these projects reinforce the conclusions made in our research as they contain several vulnerabilities and are widely used.

Another issue is that determining whether a metric is useful depends on context. This study tried to approach this concept from two different perspectives, namely whether the metric serves as an indicator of quality attributes of the system and whether the metric will translate into advice to the system owners. The authors believe that these two perspectives serve the most prevailing use-cases in which this metric may be used.

There are also several confounding variables which may have influenced the results of the interviews. For instance, if technical consultants recently inspected the dependencies of a system manually as part of their regular work, they are better able to predict the dependency freshness of a system. Asking the consultants about different known systems also introduced another variable as systems vary widely in size, age and functionality. Additionally, interviewer bias may have impacted the answers to the questions regarding the usefulness of the metric. These issues have been countered by interviewing a larger set of technical consultants, and explaining the purpose of the interviews.

IX. Conclusion

The overall result of this study is the definition of a metric to aid stakeholders in deciding on whether the dependencies of a system should be updated. In particular, the contributions of this paper are:

- The definition of several metrics to quantify dependency freshness at the component-level, including their advantages and disadvantages.
- An assessment of the current state of dependency management of a large set of industry systems.
- The definition of a system-level metric using a benchmark approach, so that a single rating can express the dependency freshness.
- The validation of this metric, showing the implications of low dependency freshness with regards to security, its usefulness in practice and its suitability for long-term monitoring.

Overall, the metrics show a great potential in quantifying the dependency freshness of a system. They provide a quantitative basis for discussions and thus can help stakeholders to make decisions about updating specific dependencies.

Furthermore, this study outlines several preconditions to enable a good dependency management process. These preconditions include a high-level of automated testing and agile processes. The relationship between these requirements and good dependency management seems worth investigating. Moreover, to improve the usefulness and applicability of the proposed metrics we envision three area's of future work.

Metric refinements: The current metrics do not take attributes of a dependency into account, such as its size, functionality or popularity. Neither does it look at how the dependency is used by the system, how tightly the dependency is coupled or how often it is called. These attributes could provide an additional layer of granularity and possibly reduce the outlier sensitivity of the dependency-level metrics.

Update effort estimation: While the metric gives a good indication of how a system is performing with regards to dependency freshness, it is still hard to put this number into context. As discussed in the introduction, the choice of updating dependencies is a balancing act between effort and risk. Yet, the metric does not give a clear indication what amount of effort has to be put into the system in order to counter the risk expressed in the dependency freshness rating.

Impact on software quality: Metrics are often used to monitor and improve the quality of software systems. Now that a metric is defined to monitor the dependency freshness of a system it would be interesting to see whether applying this metric has a positive impact on the dependency management of a system.

References

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[2] V. R. Basili and B. W. Boehm, "COTS-based systems top 10 list," *Computer*, vol. 34, no. 5, pp. 91–95, May 2001.

[3] D. J. Reifer, V. R. Basili, B. W. Boehm, and B. Clark, "Eight lessons learned during COTS-based systems maintenance," *IEEE Software*, vol. 20, no. 5, pp. 94–96, 2003.

[4] H. Seo, C. Sadowski, S. G. Elbaum, E. Aftandilian, and R. W. Bowdidge, "Programmers' build errors: a case study (at Google)." in *International Conference on Software Engineering*, 2014, pp. 724–734.

[5] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 378–387.

[6] ——, "Semantic versioning versus breaking changes: A study of the Maven repository," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, 2014.

[7] J. Businge, A. Serebrenik, and M. van den Brand, "An empirical study of the evolution of Eclipse third-party plug-ins," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, ser. IWPSE-EVOL '10. New York, NY, USA: ACM, 2010, pp. 63–72.

[8] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source Java projects," in *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 2011, pp. 1317–1324.

[9] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, "A study of library migration in Java software," *arXiv preprint arXiv:1306.6262*, 2013.

[10] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*. ACM, 2009, pp. 57–62.

[11] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*. Wiley, 1994.

[12] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*. IEEE, 2007, pp. 30–39.

[13] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.

[14] T. L. Alves, J. P. Correia, and J. Visser, "Benchmark-based aggregation of metrics to ratings," in *Software Measurement, 2011 Joint Conference of the 21st Int'l Workshop on and 6th Int'l Conference on Software Process and Product Measurement (IWSM-MENSURA)*. IEEE, 2011, pp. 20–29.

[15] M. Cadariu, "Tracking vulnerable components in software systems," Master's thesis, Delft University of Technology, 2014.