Vulnerability Risk Modelling in Open Source Software Systems

Master's Thesis

Rens Heddes

Vulnerability Risk Modelling in Open Source Software Systems

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Rens Heddes
born in Hoorn, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Software Improvement Group
Fred. Roeskestraat 115
Amsterdam, the Netherlands
www.sig.eu

Cover picture: The call-graph associated with the `https://github.com/authorjapps/zerocode` project.

Vulnerability Risk Modelling in Open Source Software Systems

Author: Rens Heddes
Student id: 4453859
Email: r.c.a.heddes@student.tudelft.nl

Abstract

Recent large scale cyber security incidents such as the Equifax data breach, where the personal information of around 160 million Americans leaked, demonstrate the current risk of security vulnerabilities libraries which software projects depend on. The usage of libraries forms an integral part of modern software development and is a widespread practice across software projects. Libraries make it possible to use proven implementations of certain functionalities without duplicating it. However, this means the usage of libraries creates a set of dependencies for software projects. While using libraries allows for increased development speeds by reusing existing code, these dependencies can also propagate problems which exist in dependencies. Therefore, a security vulnerability in a dependency can have a major impact on the software project as a whole.

Currently, there are analysers which perform a high level analysis which can identify vulnerable dependencies. However, these analysers are limited to the package level, where either a whole library is considered vulnerable or safe. In reality, the situation is often more nuanced, where only certain functions of a library pose a security risk. Considering the growing number of dependencies of software projects and the increasing number of vulnerability disclosure, the dependency update management process is currently a difficult task.

Therefor a more fine-grained type of analysis could help developers in identifying and mitigating actual security risks. In this thesis, we propose a new risk modelling approach which uses fine grained analysis to concentrate these efforts as best as possible and increase security of software applications. Further we perform an extensive evaluation to compare it to existing risk approaches to investigate the accuracy of the proposed approach. We find that the new risk model is more accurate in prioritising risk mitigation strategies, with an average increase of 8% of current state of the art risk models. The model does require function level vulnerability information which does not exist for all disclosed vulnerabilities and is an active area of research.

Thesis Committee:

Chair:	Prof. Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	Dr. S. Proksch, Faculty EEMCS, TU Delft
Company supervisor:	Dr. Ir. M. Živković, Software Improvement Group
Committee Member:	Dr. J.E.A.P. Decouchant, Faculty EEMCS, TU Delft

Contents

List of Figures

Chapter 1

Introduction

In recent years we are witnessing an incredible amount of digital transformation. Activities that used to require a great deal of manual interaction, can now be completed easily with the tap of a finger. For example, making a reservation at a restaurant can now swiftly be done on a website and receive a confirmation via email in a matter of minutes. While it is incredibly convenient for users, these systems require extensive development time and effort. Modern software systems reuse proven code that has been created by others to solve an existing problem. For example, such a library could provide the functionality to send emails after a reservation.

The usage of libraries creates dependencies for software projects and is the backbone of modern software engineering. Further, each library which is used can have its own set of dependencies, thus creating transitive dependencies. The amount of imported code for any software project therefore likely exceeds the written code by orders of magnitude. While this reuse opportunities are a boon for productivity, they come at a price. From the security perspective, this can have negative consequences since each of those dependencies can contain vulnerable code which can impact the security of the whole project. A recent example of this risk is the vulnerability found in the popular Java logging library Log4J. As a logging library, Log4J sees prevalent usage in a wide variety of Java projects. Further, the ”Log4Shell” vulnerability is very easy for a malicious actor to exploit and compromise the security of not only the whole project, but the entire host which runs the vulnerable project.

The consequences of vulnerabilities in dependencies can be seen in recent incidents such as the *Equifax* data breach, where the maintainers of a system at Equifax failed to update a vulnerable version of a dependency, namely Apache Struts. This resulted in a massive data breach and the private records of approximately 160 million North Americans leaked [28]. These incidents exemplify the importance of keeping dependencies of software projects up to date. However, with the growing size of dependency networks, this becomes increasingly difficult since updating a dependency is not always trivial [7].

Further, not only the total number of dependencies of a software project makes it difficult to keep up to date and thus mitigate against the risk associated with vulnerable dependencies. The sheer number of disclosed vulnerabilities makes it infeasible to manually stay up to date with new security issues in dependencies. In 2020 there were a total of $18,325$

vulnerabilities disclosed as a Common Vulnerability Exposure (CVE)[1], and is on the rise.

In order to make the process of identifying vulnerable dependencies in software projects easier than manually inspecting vulnerability databases, there are vulnerability checking tools such as *NPM Audit* [2] & *SIG OS Health* [3]. This is a massive improvement over manually checking for vulnerabilities in dependencies. However, these tools are operating package level, meaning these tools are limited to identifying entire dependencies as vulnerable instead of the actual functions in that dependency which are vulnerable. This can result in numerous vulnerability warnings without the vulnerable code actually being reachable from the application in which these dependencies are being used, and thus are flagged incorrectly as a security issue. These false positive warnings are a huge problem in software development since it has a large negative impact on the adoption, and active utilisation, of static code analysis tools [5]. In order to reduce the number of false positive security warnings, however, a different method of vulnerability analysis is required since it is impossible to inspect if vulnerable code is reachable in a project without a more fine grained approach than package level inspection.

Recently, Ponta et al. [26] made an effort to reduce the number of such false positive security warnings present in a vulnerability audit for Java projects. This was done in a code-centric approach, where abstract syntax tree (AST) comparison was used to verify whether the installed version of the dependency contained the vulnerability as opposed to being limited to metadata about the dependency. Using the list of vulnerabilities that are present in the application's dependencies, a combination of static and dynamic analyses were used to generate a call graph for the application which could then be used to inspect whether vulnerabilities were reachable from the application or not. While this is a great improvement for developers, it has two concerns. One is that this method does not provide insight in which vulnerable dependency should be updated first from a security perspective, and the other is that it is only available for Java projects.

For the first concern, there is a method to estimate the severity of a vulnerability in the form of a Common Vulnerability Scoring System score (CVSS score)[4]. For the majority of disclosed security vulnerabilities, such a score is assigned and thus the estimated severity of vulnerabilities is available. However, CVSS scores are a very generalised estimation of risk which in and of itself does not take into account how the vulnerability relates to other parts of the software project in which it could potentially be exploited.

A good example of how the context in which the vulnerable code is used can have a large effect on the risk posed by a vulnerability, is the vulnerability which was used in the Equifax data breach. That vulnerability affected Apache Struts setups which allows execution of Object-Graph Navigation Language from user input. Recent versions of CVSS scores allow for environmental factors to be incorporated in the severity score. However, the developer/maintainer of the software project in which the vulnerable dependency is used should do those assessments manually, and for a large project with many dependencies this quickly becomes infeasible. So, while CVSS scores can serve as a starting point in the

[1]`https://www.cvedetails.com/browse-by-date.php`, *accessed on 17 Sept. 2021*

[2]`https://docs.npmjs.com/cli/v6/commands/npm-audit`

[3]`https://www.softwareimprovementgroup.com/capabilities/sigrid-open-source-health/`

[4]`https://www.first.org/cvss/specification-document`

selection process to prioritise vulnerable dependency updates, they are currently not suitable for risk estimation.

The goal of this research is to help developers mitigate the risk of vulnerabilities in dependencies by providing actionable insight/advise for the vulnerabilities which actually have an effect on the security of their software projects.

In order to properly understand the problem, we will first analyse current update behaviour of developers regarding security updates of dependencies. The goal is to investigate the current state of dependency update management processes, and the extent to which vulnerability information plays a role in those processes. In this thesis we first focus on the following research question to properly understand the current state of vulnerable dependency update behaviour and how we can help developers mitigate against the risk of vulnerable dependencies. **RQ.1** *Is vulnerability information currently taken into account by developers of software projects in the update management process?*

We investigate this with an analysis on historical dependency update data to establish the role of risk of vulnerabilities in the dependency update management process. In this analysis we find that there is no correlation between the severity of a vulnerability in a dependency, and the time it takes to update that dependency. Further, depending on the actual vulnerable functions also has no effect on the time it takes to update the dependency of the vulnerable function. This suggests that, in accordance with previous research [21], a majority of developers seem to be unaware of potential risks posed by vulnerable dependencies.

Ultimately, the goal of this research is to help developers of software projects understand and mitigate against risk associated with vulnerabilities in dependencies of software projects. A new risk model can help in filtering vulnerability warnings which are not relevant for the project using the dependencies, and provide insight in which vulnerable dependencies should be updated first. The improved precision and actionability should have a positive effect on the adoption of such a risk model by developers of software projects, which will help in raising awareness of the risk these models estimate.

We propose four different objective quantitative risk models that use function level analysis in combination with CVSS scores to assess the security risk for an application that has open source dependencies. The model needs to take into account transitive dependencies as well as direct ones when determining the security risk for a software project. In order to asses whether we can help developers in this process, we investigate the following question. **RQ.2** *Can a risk model identify which vulnerable dependencies pose the highest risk for a software project?*

We perform an established method of evaluation to compare the models to current state of the art risk models which exist in different domains on a set of open source projects which have depended on vulnerable code in dependencies. In this evaluation, we first establish that a risk model based on relative betweenness centrality and CVSS scores seems the most suitable out of the four models we propose. Further, the most suitable proposed risk model performs significantly better than current risk models.

In order to perform the function level analysis for different programming languages, this research uses a preliminary version of the infrastructure that is being provided by the

FASTEN project[5], funded by EU within Horizon 2020 program, which aims to provide a new framework for static code analysis for OSS. The project is focused on using statically generated call-graphs to perform analysis on the method level.

The scientific contributions made in this thesis can be summarised as follows:

- A dataset of $193,542$ dependency updates performed over 927 open source repositories, containing version, time and vulnerability information. As well as a method of establishing such a dataset.
- A risk model which uses the call-graphs of a software project and its dependencies in combination with function level vulnerability information to estimate the risk disclosed vulnerabilities pose for the software project, as well as clearly pinpoint which vulnerabilities/dependencies are responsible for the most risk.

The implementation and data can be found on `https://doi.org/10.5281/zenodo.5777851`

[5]https://www.fasten-project.eu/view/Main/Introduction

Chapter 2

Related work

In this chapter, existing scientific literature is discussed. We have identified related work in three areas: Existing scales for risk assessment; Estimating (base) risk for found vulnerabilities; Risk propagation in connected networks; Risk model evaluation methods. The goal of the literature research is to have the necessary knowledge to define, build, and evaluate models for vulnerability risk propagation in application call-graphs.

The literature research started with several papers provided as a start for this thesis topic. Ponta et al. [26] which describes a method of using reachability to restrict the list of vulnerabilities in dependencies to the ones which could actually be exploited, Kotzanikolaou et al. [20] a paper about risk propagation in networks, and White and Borgatti [32] about relative betweenness centrality in graphs.

Another important area is establishing an inherent risk associated with the vulnerabilities in software systems. A good starting point for this step is investigating the Common Vulnerability Scoring System (CVSS). This system is an industry-standard for estimating the severity of a vulnerability based on its principal characteristics Team [30]. In the user guide it is stated, however, that it is wrong to use CVSS on itself as a risk estimate. Therefor, the goal is to incorporate the context of how vulnerable code is used in combination with the CVSS score to generated a risk estimate. The context of vulnerability usage is derived from the call-graph of the application

Using graphs in a risk calculation is an established concept, such as in [14, 18, 2]. Most of these methods of modelling rely on graph centrality measures as an estimate for the importance of a node in the graph. Therefor we also identified papers which analyse and compare graph centrality measures [12, 32, 15] Lastly, colleagues from SIG suggested [4] as literature for scaling software measurements.

2.1 Estimating intrinsic risk for vulnerabilities

The majority of disclosed vulnerabilities are in the format of a Common Vulnerability Exposure (CVE). These are disclosed in public databases such as the NVD[1]. Nearly all CVE's

[1]`https://nvd.nist.gov/`

are accompanied by a CVSS score, and these are determined by a group of experts. As stated in the CVSS user guide[2], a CVSS score alone should not be used as a risk score.

In order to assess whether CVSS scores can serve as a basis for estimating risk for vulnerabilities in software projects, it is good to establish what CVSS scores are, how they are calculated, and their uses. CVSS is a system that attempts to objectively estimate the severity of disclosed vulnerabilities on a scale from 1 to 10. This system is currently the most used metric for assessing the impact of discovered vulnerabilities. Several studies have been conducted towards evaluating this scoring systems, or specific versions there of. In Scarfone and Mell [27], version 2 of CVSS is extensively compared to version 1. This is done by comparing how the output for specific input values and ranges differ between these versions. Further, it investigates how input categories contribute to the final score.

Further Aksu et al. [1] provide a quantitative risk assessment methodology based on the several components which define the CVSS score, such as the Access Complexity or Authentication, to make a risk prediction as opposed to CVSS itself which provides a severity estimate. The assessment method also allows for graph-based risk assessment, by formulating a consolidated algorithm. Although few examples of the proposed methodology are given to evaluate this methodology, it forms an interesting perspective on how a risk prediction could be made from the information available from CVSS.

Another interesting research regarding CVSS is performed by Allodi et al. [3], where it is studied how different educational and professional expertise backgrounds impact how someone scores a vulnerability form a textual description. This study finds that security experts indeed are more accurate in their estimate.

2.2 Risk propagation in connected networks

In order to tailor the risk estimate to the specific application, an analysis needs to be performed on how the risk of a single vulnerability propagates through the call-graph. In Kotzanikolaou et al. [20], a model is proposed which calculates cascading risks in networks, mainly how failures in power grids have cascading effects in which a small problem can lead to huge problems by overloading other parts in the grid. Some parts of this model can be carried over to cascading risk in call-graphs, especially how the risk of vulnerabilities of nodes in the call-graph propagate to other elements of the application. In order to use this model, it is necessary to determine the likelihood that a risk of one node impacts the next node, for which a formula needs to be defined for vulnerabilities in call-graphs.

Alhomidi and Reed introduce a method in [2] for risk assessment in an attack graph, which is a graph of all known attack paths of an application, by using genetic algorithms. Further in Shivraj et al. [29], a risk assessment framework is proposed for vulnerability propagation in internet of things (IoT) networks. The framework focuses on calculating the aggregated risk that a specific node in the IoT network gets compromised. This calculation is a combination of the risk that the node gets compromised directly, and the risk that the node is compromised via the connection with a neighbouring node. The risk calculation makes extensive use of attack paths which are identified. The framework works for a wide

[2]`https://www.first.org/cvss/user-guide`

variety of thread methodologies, such as STRIDE [23], DREAD [19] & LINDDUN [33]. The framework proposed in this paper can be applied to vulnerability propagation in a call-graph, and could also be combined with the propagation model proposed by Kotzanikolaou et al. [20].

Another important part of estimating risk for an application is to estimate the importance of each element for the application. Graph centrality measures are attempting to estimate that importance [12]. These measures are also proven to be an effective method of defining important nodes in attack graphs [17] and determining the order of priority for fixing vulnerable nodes [18]. Since a call-graph is a directed graph, the original betweenness centrality, as defined by Freeman, cannot directly be used for this use cases. However, White and Borgatti [32] introduce an adapted definition of betweenness centrality for directed graphs. Further, Hinne [15] proposes local versions of centrality measures, including betweenness, in order to reduce the run-time complexity for the calculation of those metrics. This proves to be effective for undirected graphs, however, for directed graphs, results vary and would therefore need further investigation in order to be used in this project.

2.3 Existing scales for risk assessment

Lastly, as this projects main goal is creating a method to alert developers of risk within the dependencies they use in applications they are building, it is relevant to know how to best present risk alerts. In [24], multiple methods of alerting are compared to see how users of these systems react to these messages. This study shows that a likelihood scale is, in general, more advantageous for security alerts compared to a binary/category system.

In Alves [4], an approach is introduced for scoring software quality metrics. This approach uses two levels of aggregation which can be used to estimate the quality of software components in terms of a 5 star rating score based on raw absolute software metrics, such as McCabe complexity. This approach heavily relies on establishing thresholds based on a benchmark of software projects. This score allows for a good comparison between quality of software projects, since the score is relative. Meaning, that a 5 star project has a far higher quality compared to other projects, and a 2 star project slightly below average. This approach could be fitted to a risk score, but would require a large representative sample of software projects to establish a baseline score.

2.4 Summary

In this chapter we identified a wide variety of previous works which is relevant for estimating risk for software projects. Several works have extensively investigated CVSS scores and these are accepted as the established standard for estimating the severity of disclosed vulnerabilities. The components which make up a CVSS score, such as Access Complexity, can be used in risk models to incorporate the context in which the vulnerability exists and how that influences the perceived risk.

Further, CVSS scores are widely used in several risk models. These risk models exist in several domains such as estimating the risk of vulnerable hosts in a (IoT) network. Other more general models also exist, but require a high level of manual estimation.

Lastly, a risk score should be represented on some form of likelihood scale in order to be as clear for users as possible. An approach could be the 5 star scale [4], which would require a large set of representative software projects for a proper benchmark. However, a more straightforward approach would be an approach which, in accordance with CVSS scores, uses a scale from 0 to 10, where 0 represents no risk at all, and 10 the maximum risk.

Chapter 3

Role of vulnerability information when updating dependencies

In this chapter, we delve deeper into RQ.1 (Is vulnerability information currently taken into account by developers of software projects in the update management process?) Previous research by Kula et al. [21] has shown that developers of software projects generally do not keep up to date with new versions of dependencies; 81.5% of the systems that were studied keep their outdated dependencies. They also investigated the update behaviour for dependencies for which a security advisory has been issued, which finds that still a majority keep using outdated vulnerable versions. Further, it finds that developers are also unlikely to act upon security advisories relating to vulnerabilities found in dependencies, with 69% of the developers that were interviewed stating that they were not aware of vulnerabilities in dependencies.

However, their analysis does not look into the effect CVSS scores of vulnerable dependencies have on the update behaviour of developers. Considering CVSS scores are used in automatic security advisory notifications such as Dependabot. It could be the case that dependencies containing a vulnerability with a high CVSS score are updated more quickly than those with a lower score. Further, their research does not make a distinction between projects actually depending on the vulnerable pieces of code, and the projects having a vulnerable dependency but not depending on the vulnerable code. In this chapter we perform an analysis to find if making this distinction has an affect on the update behaviour of developers regarding vulnerable dependencies.

In order to investigate the effect of CVSS scores and the usage of vulnerable code on the update behaviour we perform the following investigation.

We first create a dataset of updates to dependencies of open source Java projects to compare update behaviour of developers. A new dataset is created in order to investigate the update behaviour from the perspective of a developer of an open source repository. The data and metrics used by Kula et al. [21] are concentrated around the perspective of the usage of a library version across software projects. This makes it hard to measure changes in update behaviour within a single repository. The dataset creation is focused around the following question: *How can we construct a dataset to further investigate changes in update behaviour in relation to vulnerabilities in individual projects?*

We first perform an analysis of software projects which have used vulnerable dependencies to establish an update behaviour baseline of developers. This includes all vulnerable dependencies, regardless whether or not the vulnerable code was used. In this analysis we focus on answering the following question. *Do maintainers update vulnerable packages with a high CVSS score quicker compared to vulnerable packages with a lower CVSS score or no vulnerabilities?*. Subsequently, we can investigate the update behaviour regarding vulnerable dependencies where the vulnerable code was used, in order to determine whether or not depending on vulnerable code has an affect on this behaviour. *Does the actual usage of vulnerable code affect the update delay of vulnerable dependencies?*

3.1 How can we construct a dataset to further investigate changes in update behaviour in relation to vulnerabilities in individual projects?

In order to investigate the effects CVSS scores and usage of vulnerable code have on the update behaviour of developers of software projects we need to construct a dataset which allows measuring the update behaviour and contains the necessary vulnerability information (CVSS scores and effected functions). We construct the dataset in a few comprehensive steps. The first step is making a selection of open source repositories to scan. The second step is scanning all dependency updates which occurred in that repository, in which we are mostly interested in the timing and the different versions. Finally, we normalise the data to be able to make comparisons in update behaviour between repositories.

The selection of projects to scan was determined by the vulnerabilities for which we can perform fine-grained analysis on a function level. In order to determine the effect of using vulnerable code on update behaviour it is necessary to have function level vulnerability information. Not all disclosed vulnerabilities have that level of information associated with them. Therefore we create a list of vulnerabilities with necessary function level information in the FASTEN Knowledge Base.

The criteria for the selection of vulnerabilities were:

- It should have been disclosed in the last 3 years, so that projects which update to a fixed version of the vulnerable dependency have done so recently. This is done so that the projects we analyse are under active development.
- Only select one vulnerability per library. We select one vulnerability to keep the analysis feasible and comprehensive. Libraries can usually be effected by multiple vulnerabilities over different versions. We select the vulnerability which affects the most functions in the library.

For each of those selected vulnerabilities, select the last version of the package which was affected by that vulnerability. Subsequently find open source repositories which have depended on that vulnerable library version, but have updated to a fixed version. The repositories are found using the *libraries.io* API. *Libraries.io* provides a structured database of existing libraries and there versions. Further, for each library version it maintains a list

Table 3.1: The number of scanned repositories per vulnerability

vulnerability	found repositories	found fix updates for
CVE-2020-8840	104	84
CVE-2019-12402	101	57
CVE-2018-10936	100	37
CVE-2020-11612	92	53
CVE-2020-10683	82	39
CVE-2020-1745	35	19
CVE-2019-5427	33	12
CVE-2019-12400	32	12
CVE-2020-5398	32	19
CVE-2019-10086	32	22
CVE-2019-12401	31	16
CVE-2019-12415	31	17
CVE-2020-5408	30	15
CVE-2020-26945	30	16
CVE-2020-26168	30	9
CVE-2020-5421	30	19
CVE-2019-13990	30	11
CVE-2020-9489	27	4
CVE-2020-13935	26	3
CVE-2020-26883	21	1
CVE-2020-7692	20	4
CVE-2020-13933	19	2
CVE-2021-22134	18	9
CVE-2020-13920	14	3
CVE-2020-8908	13	10
CVE-2019-3802	13	6
CVE-2020-25633	12	5
CVE-2021-22112	9	3
CVE-2020-17530	5	1
CVE-2020-17523	5	3
CVE-2021-29505	5	5
CVE-2021-28168	3	1
CVE-2020-7226	2	2
CVE-2019-9212	2	0
CVE-2021-26117	2	1
CVE-2021-27807	2	1
CVE-2021-28165	2	0
CVE-2020-13949	1	0

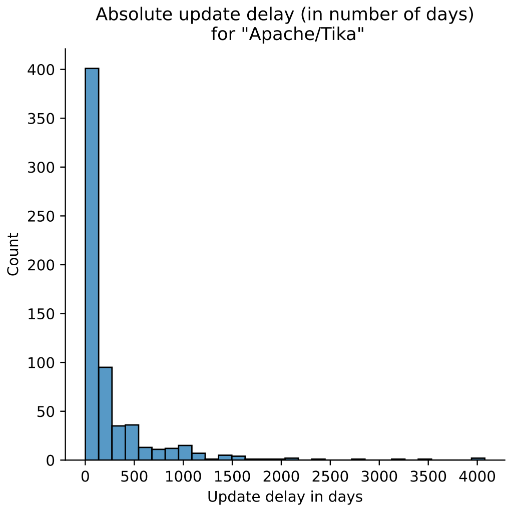

Figure 3.1: The distribution of days between a new release of a dependency being made available and ”Apache/Tika” updating to that newer version.

metric	update delay in days
count	646
mean	251
standard deviation	453
minimum	1
25th percentile	28
50th percentile	84
75th percentile	265
maximum	4077

Table 3.2: Statistical values of *update delay* distribution.

of open source repositories which depend on that library version. This list is ordered by activity and popularity.

The second step is to scan all identified repositories for the data needed to analyse the update behaviour. We scan the history of each repository for updates to its dependencies. In this scan we consider all dependency updates per repository which are found using the git history of the repository. When analysing the update behaviour of a repository, one of the most interesting aspects is the speed with which updates are performed on dependencies. We express this as the time it takes between a new version of a dependency being released and the dependency being updated, we define this as *update delay*. We use the released date from Maven central as the release date of a library version. From the git history it is known which commit is associated with a specific dependency update, and thus we can use the commit timestamp to determine when an update is performed. Further, We mark the commit that updates the dependency which contains the vulnerability we identified in step 1 as the fix commit. Additionally we define the *update delay* to the fixed version of that dependency as *fix update delay*.

Repository scanning results We scanned the update histories for a total of 703 open source repositories which used 1 of 37 vulnerabilities. An extensive overview of the number of scanned repositories per vulnerability can be seen in Table 3.1. Note that for some vulnerabilities the number of open source repositories using the vulnerable version was limited in the *libraries.io* database. Now, in order to compare the update behaviour across these repositories, we need to ensure we can compare the *update delays*, for which the data

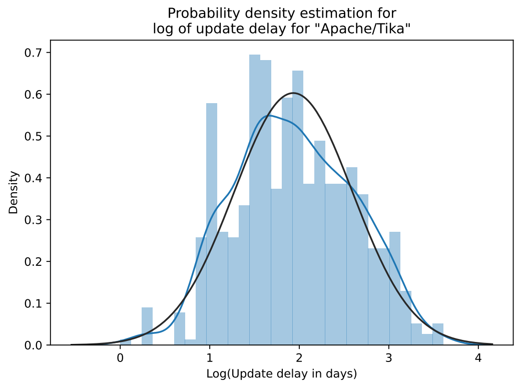

Figure 3.2: The log (base 10) over distribution of update delays of dependencies in Apache/Tika (bars), with its associated kernel density estimation (blue line), and the fitted normal distribution (black line).

needs to be normalised.

A first important observation is that the absolute *update delays*, are limited to non-negative values, since a repository can only update to a new version of a dependency after it is released. As a result, the distribution of *update delays* for any scanned repository is a heavy tailed distribution containing only non-negative values, as seen in Figure 3.1. Most of the updates occur within a relative short time frame (< 50 days), but some updates occur in larger time frames, such as that of ”Apache/Tika”, with an update delay of 4077.

Repositories can have vastly different dependency update management processes. For instance a project could actively use automatic dependency update tools to continuously stay up to date with new versions of dependencies. While another project only periodically manually performs dependency updates. Therefor the distribution of updates for repositories is not directly comparable. In order to be able to compare the dependency update behaviour between multiple different repositories, we need to transform the *update delay* data to normal-like shape. Specific updates can then be compared to the general update behaviour of each project to find if that update was performed relatively quickly or slowly.

The general shape of the distribution of *update delays*, such as seen in Figure 3.1, for a repository seems to be a logarithmic normal curve. This can be tested by taking the log (base 10) over the *update delay* distribution. In order to check whether it is indeed a logarithmic normal distribution, the Kolmogorov-Smirnov test [16] over the distribution of log *update delays* was performed. In this test we compare it to a normal distribution associated with the mean and standard deviation of the transformed distribution ($\mu = 1.93$ and $\sigma = 0.66$). This yields a maximum distance between a fitted normal distribution and the observed data, henceforth referenced as D-statistic value, of 0.063 and an associated p-value of 0.09, suggesting that the distribution of *update delays* is indeed a logarithmic normal curve.

One thing that stands out in the logarithmic distribution, is how the right side of the

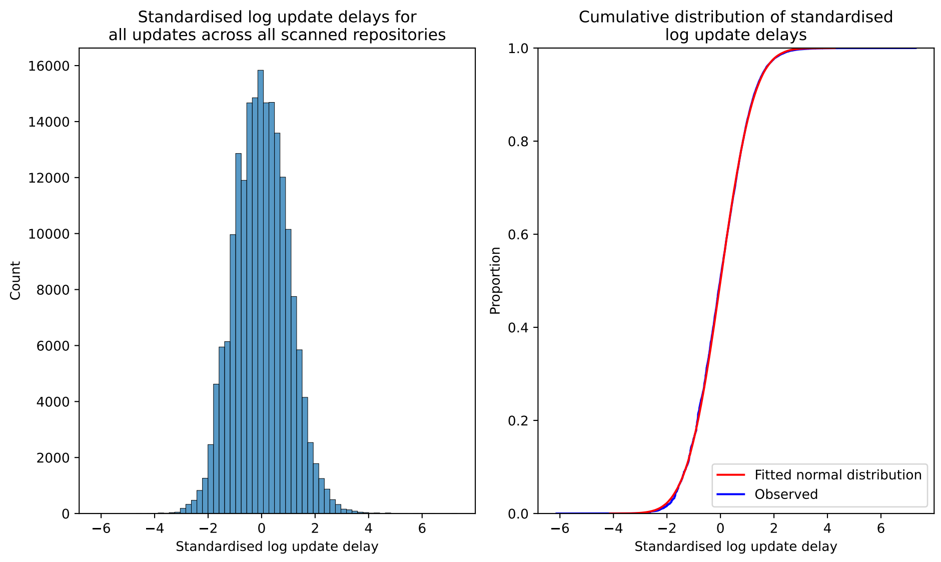

Figure 3.3: Standardised *log update delays* over all updates across all repositories

Table 3.3: Percentage of datapoints that lie within 1, 2 & 3 standard deviations from the mean in the observed distribution

range	percentage of data within range
$\mu \pm \sigma$	68.3%
$\mu \pm 2\sigma$	95.9%
$\mu \pm 3\sigma$	99.7%

distribution is quite a bit smoother than the left side. This can, at least partially, be attributed to the granularity of the *update delays*. Logarithmic function are very sensitive to slight deviations for lower input values. The *update delays* are calculated in the number of days between release date and date of update, which has as effect that quick updates ($<$ 10 days) result in distinct values which are mapped to individual bins in the logarithmic distribution with gaps in between these individual bins.

Now we have a normalised distribution of logs of *update delays* for every scanned repository. However, each of those distributions has a different mean and standard deviation, and therefore comparisons are not meaningful yet. However, these normal distributions can be standardised to be able to compare the update behaviour across repositories. For this we can use the formula associated with Z scores, with μ being the mean of the log normal distribution for a single repository, and σ the associated standard deviation.

$$Z(x) = \frac{x-\mu}{\sigma} \tag{3.1}$$

The formula ensures the resulting distribution have a mean around 0 and a standard deviation of 1, in which the distance from the mean for each value across repositories can be compared in terms of standard deviations. In order to check if these values can indeed be used to compare update delays across repositories we can check if the distribution of *standardised log update delays* are indeed normal-like. Now, since the sample size of all updates is vast ($N = 193\,112$), any slight deviation from a normal distribution will yield in a rejection of the null hypothesis (that the observed distribution is indeed normal). The D-statistic for a Kolmogorov-Smirnov test is for this distribution is 0.01, and thus seemingly follows the fitted normal distribution quite well as is also seen in the two cumulative distribution functions of these two distribution in Figure 3.3. However, the p-value associated with the test is extremely low ($3.94 * 10^{-10}$), which suggest the resulting distribution is not normal, and it can indeed be seen that there are deviations between the fitted normal distribution and the actual distribution. Firstly, as explained earlier, because of the lack of granularity of updates occurring within the first few days of an update being made available, the left tail of the distribution is somewhat rugged, and does not follow the fitted normal cumulative distribution as nicely as would be expected with an actual normal distribution of this sample size.

Further, the observed distribution has some extreme values which would not be expected in an actual normal distribution; the $min = -6.1$ and $max = 7.3$) while values smaller then -4 and larger than 4 would be extremely unlikely if the data was an actual normal distribution.

While the observed data is not an actual normal distribution, it does share numerous properties with a normal distribution. For instance, the percentage of values observed within one, two & three standard deviations from the mean matches with what would be expected from a normal distribution. Further, if we take a sample of the observed data ($n = 5000$) and perform a Kolmogorov-Smirnov test on that sample, we still find a D-statistic value of 0.01 but associated with a p-value of 0.68 which would suggest that sample is drawn from a normal distribution.

It is clear that the distribution of *standardised log update delays* is not an actual normal distribution. However, for the purpose of comparing dependency update behaviour across software projects, the main goal is to be able to compare the timing of one specific update to the general update behaviour of the project in which the update was performed. A distribution with a well defined mean and standard deviation are needed. Thus, we assume the data resembles a normal distribution sufficiently to be able to use *standardised log update delays* as a method of comparing update behaviour in software projects.

3.2 Do maintainers update vulnerable packages quicker compared to regular dependency updates?

The first aspect we investigate is the effect of CVSS scores on the update behaviour of developers. In general, in accordance with the CVSS specification document, which states that CVSS scores can be used as a help in prioritisation process for vulnerability management. The expectation is that the presence of a vulnerability with a higher CVSS score,

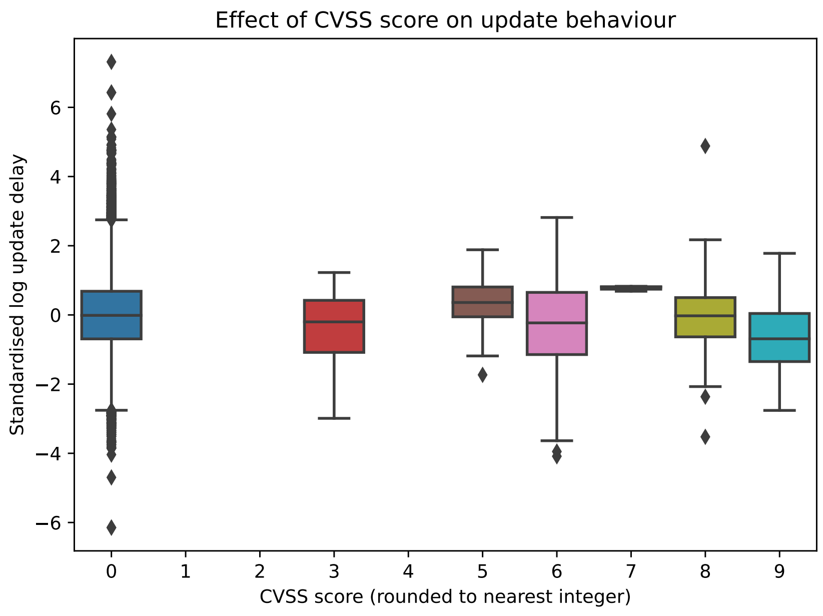

Figure 3.4: Relation between CVSS score and update to a fixed version of a dependency corresponding with the CVE with that score. Updates to non-vulnerable dependencies are assigned a score of 0. CVSS scores are rounded to the nearest integer

should correspond to a relatively quicker update in that repository, if the maintainers of the project are aware that the vulnerability is present. This analysis therefore further determines if higher CVSS scores have an effect of the awareness of security risk in dependencies.

However, as visible from Figure 3.4 there is no clear difference in *standardised update delay* based on the CVSS score of a vulnerability in a dependency. This is further motivated by an R-squared test over the *standard update delays* in to relation with CVSS scores of vulnerabilities in dependencies. This yields a score of $R^2 = 0.040$, thus nearly all variance in the *standardised update delay*, and to a further extend the update behaviour, is not related to the severity of a vulnerability.

From this analysis it seems that CVSS scores are not actively used to prioritise security updates accordingly. These findings are mostly in line with the findings from Kula et al. [21]. However, one factor which is not taken into account in this analysis is the development cost associated with performing a specific update. An update to a vulnerable dependency can take longer to perform based on the associated work with updating the dependency and ensuring all code still runs properly. Nevertheless considering the low R-squared score between CVSS scores and *standardised update delay*, it is unlikely that the perturbation associated with the development cost of updates has enough of an impact to completely hide the measured correlation.

This analysis further makes clear the lack of awareness in relation to vulnerabilities in dependencies of software projects, motivated by the apparent lack of awareness of severe

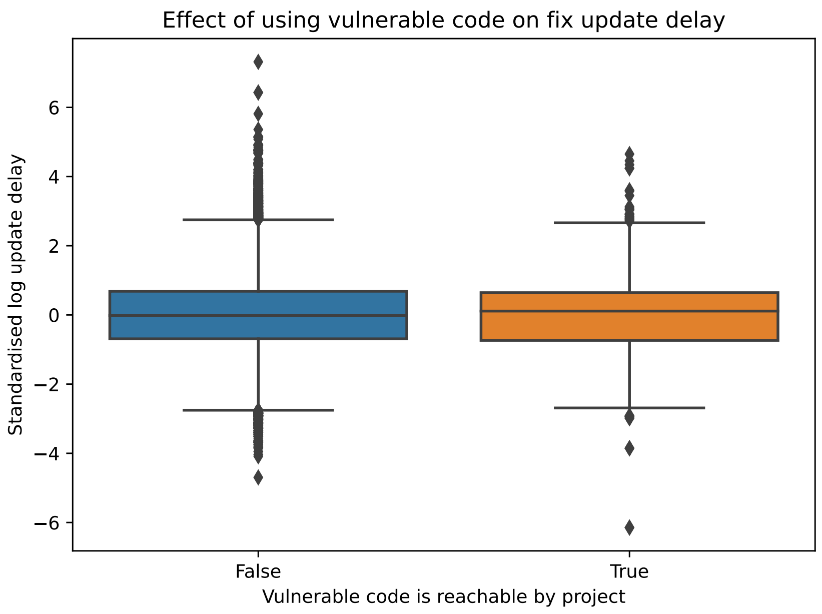

Figure 3.5: A boxplot showing the update behaviour for dependency versions where vulnerable code was used to a fixed version, compared to normal update behaviour.

vulnerabilities. However, there is a possibility that developers are aware of severe vulnerabilities but upon further investigation determine that their projects are not effected by the vulnerability. We investigate this possibility further in the next section.

3.3 Does the actual usage of vulnerable code affect the update delay of vulnerable dependencies?

An interesting aspect with regards to prioritising security updates is whether maintainers of software projects are aware of if the software project actually depended on exploitable functions in a vulnerable dependency. If that is the case, it should likely follow that projects which depended on the vulnerable code prioritise updating the vulnerable dependency in question over projects which required the vulnerable dependency but did not used the vulnerable code.

In order to investigate this aspect, we use the detailed vulnerability information present in the FASTEN knowledge base. Each of the repositories identified in section 3.1 has depended on a library containing a vulnerability for which the function level vulnerability information is present in the FASTEN Knowledge Base. We can compare the *standardised update delays* for updates which update from the vulnerable version of that dependency to a fixed version. Then, by producing call-graphs for these projects using the `fasten-maven-analyzer`[1], we can perform a reachability analysis to see whether or not the software project depended on the vulnerable code.

The results of this analysis are shown in Figure 3.5, which shows the *standardised log update delay* for fix updates performed on the dependencies which contained one of the selected vulnerabilities as described in section 3.2. Each bar in the graph represents how long it took for a single project to update to a fixed version of the selected dependency compared to the mean update delay for that project and in terms of standard deviations from the mean.

Similarly, for each of the selected vulnerabilities in section 3.2, we can plot the distribution for a more detailed view, as well as showing the names of the repositories, which were actually using the vulnerable code according to the call-graphs in combination with the function level vulnerability information from the FASTEN knowledge base.

From these figures it seems there is no strong correlation between the prioritisation of security updates, and whether or not the repository which uses the dependency is actually using the vulnerable code. In combination with the analysis regarding the CVSS scores on update behaviour and previous work by Kula et al. [21], it is clear that the lack of awareness in relation to vulnerable dependencies is widespread. This further motivates the need of a novel approach to alert the developers of software projects of the actual risks associated with vulnerable dependencies to raise awareness and increase software security.

3.4 Summary

In this chapter, we concentrated on answering **RQ.1** *Is vulnerability information currently taken into account by developers of software projects in the update management process?*. We have performed a comprehensive analysis to show that, currently, the majority of maintainers of software projects do not take vulnerability information actively into account and are unaware of the potential risk posed by vulnerabilities in dependencies. Building upon the findings by Kula et al. [21], we show that even when projects depend on vulnerable pieces of code, the update behaviour is not significantly different from updates to vulnerable dependencies in general. Additionally, the CVSS score associated with a vulnerability also has no significant impact on the update behaviour. This shows that there is a need to raise awareness for the potential risks software projects are at regarding depending on vulnerable code.

[1]`https://github.com/fasten-project/fasten-maven-plugin`

Chapter 4

Risk Model

In the previous chapter we have shown that, currently, vulnerability information is barely taken into account when planning and performing updates to dependencies of software projects. In order to ensure the security of software projects, it is imperative that vulnerable dependencies need to be updated as quick as possible, therefor project maintainers need to be aware of the risks introduced by vulnerable dependencies. However, considering more and more dependencies are being used in software projects, combined with the fact that each year the number of common vulnerability exposures is increasing, it is clear that the number of security impact notifications is quickly running out of hand. It is infeasible to keep up to date with all notifications, and is thus important to help maintainers focus their efforts on the security problems which matter most.

In this chapter, we propose a model that will help maintainers of software projects focus their efforts on the dependencies which pose the highest security risk for the application. This model uses function level vulnerability information in combination with the call-graph of the software application. More concretely, we focus on **RQ. 2** (*Can a risk model help in identifying which methods are of the highest risk for a software project?*), by proposing a model and evaluating its properties to show that it does indeed assess the risk of a software application to the best of our abilities.

In order to properly assess the risk associated with security vulnerabilities in open source dependencies in applications, it is necessary to properly define what risk is being estimated. In a business setting, risk is often expressed in terms of potential financial loss, or in more extreme scenario's such as estimated risk in national security expressed as loss of human lives. There are more abstract scales for other scenarios which range from, for example, *very high* to *low* risk. The goal of the risk assessment method of this thesis is to use openly available information, such as vulnerability severity estimates and source code for call-graphs, to make a risk prediction. Therefore, the risk model should work independently of the scope of the application in which the open source dependencies are used. This also means that the scale of the risk prediction will remain quite an abstract concept.

As stated in the CVSS specification, the CVSS score of a vulnerability should not be interpreted as the risk that a vulnerability poses for software projects which depend on software containing that vulnerability. This is mostly due to the fact that the context of the vulnerability is important [30], for example a cross-site scripting vulnerability in a dependency could be a non-issue for a project with proper input and output sanitation. Previous research has been performed to incorporate the context of a vulnerability into a risk score. Various works investigate the risk of vulnerabilities in the context of computer networks. In these situations the computer network is modelled as a graph, and network centrality measures are used to identify which hosts in the network are important from a security perspective [17][18][2]. When analysing software applications, the call-graph of the software application allows to include the context of the software project in which the vulnerability is present. The goal is to create a risk model using function level vulnerability information in combination with centrality measures on the call-graph to establish which vulnerable dependencies need to be fixed first.

In order to do this, we first establish a list of criteria which a risk model should follow, then investigate which centrality measures are suitable for the risk model. Next, propose a set of models which adhere to those criteria. In the next chapter we perform a more in-depth analysis to properly compare the models and select the most suitable one.

4.1 Prerequisites

In order to properly define our models we will use the following notations.

- A call graph for software projects denoted by $\mathcal{G} = (V, E)$ with order $|V|$ and size $|E|$
- A set of vulnerabilities $\mathcal{V}$, with a CVSS score (v3) for every vulnerability $v \in \mathcal{V}$ given by $cvss(v)$.
- For any vertex $n \in V$, a set of vulnerabilities which are present in n defined by $vulnerabilities(n)$.
- The set of vertices in the graph which can be reached from a given vertex n defined as
  $reachable(s) = \{t | \exists \{v_0 = s, v_1, ...v_k = t | (v_{i-1}, v_i) \in E, v_i \in V\}\}$.
- Similar to CVSS scores, the risk score for an application should be on a scale from 0 to 10, which will make it easier to compare relative risk of software projects in a portfolio.

In order to avoid confusion as to what properties a measurement should and should not have, several works have investigated properties which should hold for risk measurements (Denuit et al. [11]), and in a greater scope metrics for software projects (Briand et al. [6]). It is important that any risk score calculated by the proposed models adheres to the following criteria to ensure the score are intuitive and is comparable. These criteria are:

Non-negativity The risk of any software project should never be negative considering the minimum risk of an application is no risk at all when no vulnerabilities are present.

Null value In the case that no vulnerabilities are present the risk score should reflect that in a 0 value.

Monotonicity with respect to vulnerabilities Another important property is that the risk score for the application should be monotonic, in other words, additional risks in the application should only increase the risk score, the only ways to decrease the risk score should be removal of vulnerable nodes, or decrease of vulnerability scores.

Traceable It should be traceable which pieces of vulnerable code contribute to the software projects' risk, and to which extent, to ensure efforts to decrease the risk of an application can be focused on high profile targets.

Further, we work under the assumption that if a vulnerability is present and reachable from the application, it can be exploited.. More specifically, a vulnerability of a transitive dependency is just as important as a vulnerability of a direct dependency and thus the distance from project code to vulnerable code has no direct impact on the risk score.

Our risk model is created in three steps. First, we need to quantify the importance of a function. Then we use the importance in combination with CVSS scores to estimate the risk of vulnerable functions. Lastly, we need to propagate these risks to obtain a risk estimate for the entire application. These three different parts are covered in the next three sections that discuss centrality measure as a way to express importance, how to establish risk for a single function, and finally how to derive the overall risk of a system.

4.2 Measuring the importance of a function

Incorporating the context of a function in terms of its importance is done by investigating its respective call-graph, which represents the flow between different functions of the system. Further, risk is often calculated for networks, in which threat propagates between nodes. In software development the closest equivalent are call-graphs.

Using call-graphs in combination with the function level vulnerability analysis, we will be able to identify exactly which nodes in the call-graph are vulnerable and which ones are not. In order to then estimate the risk associated with reachable vulnerable nodes we need to determine how important a vulnerable node is from an attack perspecive. We can use the theory regarding attack paths in graphs for this purpose. In this theory the multiple different paths which result in a vulnerable node being exploited is used to calculate the different possible ways in which a vulnerable node can be exploited [2]. However, since calculating all possible (attack)paths in a call-graph can quickly grow out of hand with an exponential run-time complexity, this quickly becomes unfeasible for software projects. Centrality metrics have long been used to estimate the importance of a node within a graph [12], they are widely used across multiple domains, such as, investigating weaknesses of the power grid [20] and finding prominent members of social groups [10]. Hong and Kim [17] propose a method of security analysis in which they utilise centrality measures to find which vulnerable hosts in a computer network are the most critical to fix first.

In this study, we focus on the centrality measures most commonly used in risk estimation papers ([2, 17, 18, 29]): node degree, PageRank, betweenness and closeness as well as co-reachability centrality, which was is extensively used in the FASTEN project. The simplest of these is the in or out degree of a node. While this metric is very efficient computationally, it lacks in describing the importance of a node with respect to the call-graph as a whole since it considers only the direct neighbours of a node and is therefore a very local metric.

PageRank Another frequently used metric to estimate the relative importance of a node in a graph is the PageRank algorithm [25]. While this is a much better approximation than the degree of a node, it also has its drawbacks which makes it less suited for estimating the importance of a node with respect to its potential security risks. This is mainly because PageRank estimates the behaviour of internet browsing. Since a user on the internet is most likely to only click one link on a webpage, an incoming edge from a node with a high out-degree is not assigned a high score. While this might model the behaviour of browsing on the internet, the actual method calls in the call-graph of a software application often visit multiple outgoing edges.

$$C_{PR}(n) = \frac{1-df}{N} + df \sum_{j \in d^{+}(n)} \frac{C_{PR}(j)}{|d^{-}(j)|} \tag{4.1}$$

where: df = damping factor, usually 0.85
$d^{+}(j)$ = Set of nodes with an edge to j
$d^{-}(j)$ = Set of nodes with an edge from j

Co-reachability A more suitable candidate could be the *co-reachability* of a node (Equation 4.2), which is defined for node n as the number of nodes in the graph which can reach node n. This is a relatively simple but effective metric to estimate how well a node is connected to other parts of the graph, and therefore other parts of the software application. Co-reachability is quite similar to closeness centrality (Equation 4.3). The key difference between these two is that closeness centrality takes distance into account whereas co-reachability does not.

$$C_{CR}(n) = |\{w | \exists \{v_0 = w, v_1, ... v_k = n | (v_{i-j}, v_i) \in E, v_i \in V\}\}| \tag{4.2}$$

$$C_C(n) = \frac{1}{\sum_{u \in V} d(u,n)} \tag{4.3}$$

Betweenness centrality Lastly, there is also betweenness centrality which estimates the centrality of a node based on the number of shortest paths that pass through a given node. This more closely represents the number of different paths possible which can exploit this node and is therefore a good candidate. Usually, betweenness centrality only counts the number of shortest paths passing *through* a node, and therefore leaf nodes always have a betweenness centrality of 0, which does not represent the impact on risk very well considering a leaf node in the call-graph could still be exploited and has therefore a non-zero risk.

However, it can also be altered to included the endpoints of shortest paths to ensure every connected node has a non-zero centrality score.

$$C_B(n) = \sum_{s,t \in V} \frac{\sigma_{st}(n)}{\sigma_{st}} \tag{4.4}$$

Gonda et al. [14] compare several different centrality measures in an analysis of attack graph representations in order to find which vulnerability to be patched first, in which betweenness centrality shows to be the best estimator to be used in that scenario. However, since *co-reachability* was not evaluated in that paper it is unclear exactly how that measure would perform. Therefore for the proposed risk models we will compare both *betweenness centrality* as well as *co-reachability*.

4.3 Inherent risk of vulnerable functions

In order to use the centrality measures to calculate the inherent risk of a vulnerable node, we can, in the simplest form, multiply the CVSS score of the vulnerability of the node with its centrality score Equation 4.5.

$$R_{inherent}(v) = centrality(v) * CVSS(v) \tag{4.5}$$

However, this has a few shortcomings. Firstly, it could be the case that there are multiple vulnerabilities present in a single node, in which case the inherent risk formula needs a method to account for that. Since the risk score should be monotone, taking the arithmetic mean of the CVSS scores of the multiple vulnerabilities is not an appropriate method of combining multiple scores considering the risk of a high vulnerability should not be decreased by the additional presence of a lower risk vulnerability. A simple other method could be to take the sum of CVSS scores, but that results in a situation where the potential risk of a node has no theoretical maximum. This would make it hard to interpret resulting risk scores and would skew high risk scores to larger projects purely based on code volume. The simplest method for calculating the inherent risk with a theoretical maximum, is to use the max CVSS score of vulnerabilities in a node. However, a more suitable method would be to use the formula which is also used to calculate the probability of exploitation in the FASTEN deliverable 3.2[13].

$$CombinedCVSS(n) = 10 * (1 - \prod_{v \in vulnerabilities(n)} (1 - cvss(v)/10)) \tag{4.6}$$

Another shortcoming is that the sum of centrality scores in a graph does not always add up to one, which would mean the range of risk scores for an application is limited by the chosen centrality. Therefore we can calculate the normalised centrality.

$$NormalisedCentrality(n) = \frac{centrality(n)}{\sum_{w \in V} centrality(w)} \tag{4.7}$$

A different approach would be to compare the centrality measures for different nodes in the graph with the node with the highest centrality score. However, do note that this also means that the sum of centrality scores in the graph does not sum to one. However, using a propagation function which uses elements to combine CVSS score the appropriate range of risk scores can still be achieved.

$$RelativeCentrality(n) = \frac{centrality(n)}{max(\{centrality(w) : w \in V\})} \tag{4.8}$$

These formulae then allows calculate the inherent risk of a vulnerable node in the call-graph with the following equations.

$$R_{inherent,normalised}(n) = NormalisedCentrality(n) * CombinedCvss(n) \tag{4.9}$$

$$R_{inherent,relative}(n) = RelativeCentrality(n) * CombinedCvss(n) \tag{4.10}$$

4.4 Aggregating inherent function risk to risk of a system

The last step is to use the inherent risks as calculated for the vulnerable nodes by Equation 4.9 or Equation 4.10, and aggregate the risk for the entire application. In an incremental model, the propagated risk for node n could be the combined risk of direct reachable neighbours of n, in which the risk could be combined in a similar fashion as combining multiple CVSS scores in a vulnerable node, or as described in a formula, where $d^-(n) = \{w|(n,w) \in E\}$.

$$R(n) = R_{inherent}(n) + \sum_{w=1}^{d^-(n)} R(w) \tag{4.11}$$

This approach has as drawback, however, that risks in a node can be counted twice. For example, in Figure 4.1 node D could contain a vulnerability, and has therefore a non-zero risk. Since there are two separate paths from node A to node D, the risk associated with that vulnerability would be counted twice in the propagated risk for node A, because the risk from vulnerable node D propagates through node B to node A as well as through node C to node A.

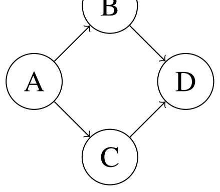

Figure 4.1: Simple graph to demonstrate how risk could be counted twice in an incremental model.

Therefore a more suitable calculation would be to combine the inherent risks of all the methods which are reachable by the software application. The methods reachable by the software application is the set of methods which are reachable by any of the ”entrypoints” of the application, in which entrypoints are defined as the functions which are run first when the application starts, and should have an in-degree of 0

$$V_{reachable} = \bigcup_{n \in entrypoints(app)} reachable(n) \tag{4.12}$$

$$Risk_{normalised}(app) = \sum_{n \in V_{reachable}} R_{normalised}(n) \tag{4.13}$$

In the case relative centrality scores are used for the inherent risk score, instead of summing the values, the inherent risk values should be combined in a similar fashion to how multiple CVSS scores in a single node are combined. Since the total risk of the application under test should be propagated through these nodes, combining them in a similar way as combining multiple CVSS scores in a vulnerable node, will yield a single risk score which should adhere to all criteria stated in section 4.1.

$$Risk_{relative}(app) = 10 * (1 - \prod_{n \in V_{reachable}} (1 - R_{inherent,relative}(n)/10)) \tag{4.14}$$

4.5 Proposed models

Based on the two different suitable centrality measures and two methods of risk aggregation described in Sections 4.2 to 4.4, we propose 4 different models. The first difference is the centrality measure which is being used in inherent risk calculation, and secondly how the centrality measure is transformed in order to compare graphs of different sizes and ensure the risk scale form 0 to 10 is achieved.

model	centrality measure	inherent risk calculation
A	*co-reachability* (Equation 4.2)	normalised (Equation 4.9 and Equation 4.13)
B	*co-reachability* (Equation 4.2)	relative (Equation 4.10 and Equation 4.14)
C	*betweenness* (Equation 4.4)	normalised (Equation 4.9 and Equation 4.13)
D	*betweenness* (Equation 4.4)	relative (Equation 4.10 and Equation 4.14)

Table 4.1: The set of proposed risk models to compare

4.6 Illustrative model calculations

In this section, we will take a look at a small example on how the different models calculate risk scores to get an indication into their properties and how suitable they are as risk models. We designed the example to clearly showcase the differences between the four different risk models. The graph used in the example is shown in Figure 4.2. This graph contains a combination of factors which are interesting with respect to the risk modelling. These factors are: a node with high number of incoming edges, a chain of nodes, and a highly connected

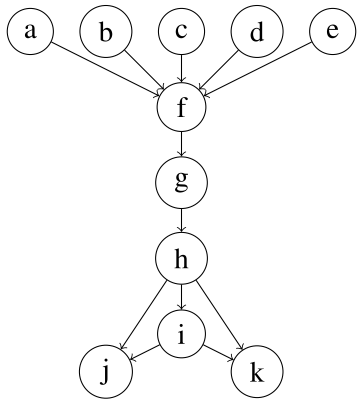

Figure 4.2: Example graph to show properties.

node	co-reachability	betweenness
a	0	0.05
b	0	0.05
c	0	0.05
d	0	0.05
e	0	0.05
f	5	0.32
g	6	0.31
h	7	0.28
i	8	0.09
j	9	0.08
k	9	0.08
sum	44	1.44
max	9	0.32

Table 4.2: Associated centrality scores.

component. Further, we calculate the risk score with a different number of vulnerabilities in the graph to highlight how vulnerability density affects the risk score of a model.

4.6.1 Centrality measures

In the graph shown in Figure 4.2, a clear difference between *co-reachability* and *betweenness* centrality measures can be observed. Where the *co-reachability* measure has a bias towards nodes close to leaf nodes. This is because those nodes can inherently be reached by more nodes than its ancestor nodes, since each node that can reach its ancestor(s) can also reach the leaf node.

This is in contrast to *betweenness* centrality which concentrates its higher scores to nodes which are in the chain from node f to h. Since the edges between these nodes form the bridge between the two larger connected components in the graph. Therefor, nearly all shortest paths in the graph pass through these nodes and thus yields higher centrality score.

4.6.2 Maximum risk score

In order to gain more insight into the behaviour of the different models, it is a good experiment to demonstrate how the max risk score for each model is obtained. Model A & C both need all nodes with a centrality score > 0, to have a inherent risk score $R_{inherent}$ of 10. This would mean for model A that all nodes except a, b, c, d, e would need to contain vulnerabilities which combine to an $R_{inherent}$, for example, each containing a vulnerability with CVSS score of 10. Whereas model C would require *all* nodes in this graph to have an $R_{inherent}$ of 10.

For model B, which uses the relative Co-reachability score, the simplest case to obtain the max risk score is if node j or k would have an $R_{inherent}$ of 10, whereas that would be node f for model D.

4.6.3 Realistic scenario with traceability

In order to show how the risk model behaves in a more realistic scenario, and to demonstrate how to calculate the degree to which a vulnerability affects the risk score and should therefore be prioritised, we place two vulnerabilities in the call-graph. The nodes which the vulnerabilities affect have been chosen at random. Further, the CVSS scores for the vulnerabilities are randomly generated using the distribution of CVSS scores of all disclosed vulnerabilities in the NVD One vulnerability $v1$ in nodes h & k with a CVSS score of 5.8, and another vulnerability $v2$ in node k with CVSS score of 9.1. The first step is to calculate the inherent risk scores for these two vulnerable nodes: These example CVSS scores are randomly drawn from a distribution of CVSS scores associated with vulnerabilities disclosed between 2015 and 2020. It was chosen to have one node be affected by both vulnerabilities to demonstrate how the inherent risk is calculated with multiple vulnerabilities present in one node.

$$\begin{aligned}
CombinedCVSS(h) &= 10*(1-(1-\frac{5.8}{10})) = 5.8 \\
CombinedCVSS(k) &= 10*(1-(1-\frac{5.8}{10})*(1-\frac{9.1}{10})) \\
&= 10*(1-0.42*0.09) \\
&= 10*0.9622 \approx 9.62
\end{aligned}$$

The next step is to calculate the $R_{inherent}$ for both vulnerable nodes for the different risk models, using the the combined CVSS score and the associated relative/normalised centrality score.

$$\begin{aligned}
R_{inherent,a}(h) &= 5.8*\frac{7}{44} = 0.92 & R_{inherent,a}(k) &= 9.6*\frac{9}{44} = 1.97 \\
R_{inherent,b}(h) &= 5.8*\frac{7}{9} = 4.51 & R_{inherent,b}(k) &= 9.6*\frac{9}{9} = 9.62 \\
R_{inherent,c}(h) &= 5.8*\frac{0.28}{1.44} = 1.13 & R_{inherent,c}(k) &= 9.6*\frac{0.08}{1.44} = 0.54 \\
R_{inherent,d}(h) &= 5.8*\frac{0.28}{0.32} = 5.08 & R_{inherent,d}(k) &= 9.6*\frac{0.08}{0.32} = 2.41
\end{aligned}$$

For which the following application risk scores are calculated. Since the two vulnerabilities are placed in a leaf node and a direct ancestor of a leaf node, the difference between the two centrality metrics is immediately visible in the higher risk scores for the models which use *co-reachability* (A & B) compared to the models which use *betweenness centrality* (C & D). Further, in most call-graphs the number of vulnerable nodes is quite sparse, on average only 0.3% of nodes in a call-graph for an application are vulnerable for the scanned vulnerable repositories in section 3.3. Additionally, the sparse population of vulnerable nodes in a call-graph models A & C, which use a normalised sum over inherent risk scores, the risk score for the entire application is quite low. It seems therefore a better fit for the purpose of calculating risks of vulnerabilities in software to model B or D.

$$
\begin{aligned}
Risk_a(app) &= 0.92 + 1.97 = 2.89 \\
Risk_b(app) &= 10 * (1 - (1 - \frac{4.51}{10}) * (1 - \frac{9.62}{10})) = 9.80 \\
Risk_c(app) &= 1.13 + 0.54 = 1.67 \\
Risk_d(app) &= 10 * (1 - (1 - \frac{5.08}{10}) * (1 - \frac{2.41}{10})) = 6.27
\end{aligned}
$$

The last step is showing the relative contribution for both of the vulnerabilities to the risk score for the entire application, the proportional risk $PR(v)$. This shows that even though the normalised and relative centrality scores yield different risk scores for the application, the proportion to which a vulnerability contributes to that score is the same regardless. Further, both prioritise the vulnerability with a lower CVSS score but with two exploitable nodes over the vulnerability with a single exploitable node but a higher CVSS score.

$$
\begin{aligned}
RP_a(v1) &= \frac{0.92}{2.89} + \frac{5.8}{5.8+9.1} * \frac{1.97}{2.89} & RP_a(v2) &= \frac{9.1}{5.8+9.1} * \frac{1.97}{2.89} \\
&= 58.4\% & &= 41.6\% \\
RP_b(v1) &= \frac{4.51}{4.51+9.62} + \frac{5.8}{5.8+9.1} * \frac{9.62}{4.51+9.62} & RP_b(v2) &= \frac{9.1}{5.8+9.1} * \frac{9.62}{4.51+9.62} \\
&= 58.4\% & &= 41.6\% \\
RP_c(v1) &= \frac{1.13}{1.67} + \frac{5.8}{5.8+9.1} * \frac{0.54}{1.67} & RP_c(v2) &= \frac{9.1}{5.8+9.1} * \frac{0.54}{1.67} \\
&= 80.3\% & &= 19.7\% \\
RP_d(v1) &= \frac{5.08}{5.08+2.41} + \frac{5.8}{5.8+9.1} * \frac{2.41}{5.08+2.41} & RP_d(v2) &= \frac{9.1}{5.8+9.1} * \frac{2.41}{5.08+2.41} \\
&= 80.3\% & &= 19.7\%
\end{aligned}
$$

4.7 Validation of model properties

As stated in section 4.1, a risk measurement should adhere to a set of criteria to prevent confusion as to what the measurement exactly represents and how it should be interpreted. In this section we demonstrate how the proposed models adhere to these criteria.

- **Non-negativity:** If that the risk models does not adhere to this property, then there should be an inherent risk score $R_{inherent} < 0$, for which either a negative *Normalised-Centrality*, *RelativeCentrality* or *CombinedCvss* needs to exist. Since centrality scores are non-negative in nature and the normalised and relative centrality scores only alter the scales, the altered centrality score $NormalisedCentrality$ & $RelativeCentrality$ cannot be negative. Further, CVSS scores are all on a scale from 0 to 10, thus the *combinedCvss* can also not be negative. Therefore, no negative $R_{inherent}$ can exist, hence no negative risk scores can exist.

- **Null value:** If this property would not be upheld by the risk models, there would be a scenario where an application call-graph G exists with an empty set of vulnerability $\mathcal{V} = \emptyset$ which yields a risk score $Risk(app) > 0$. For that to occur, there would need to exist a node n such that $R_{inherent}(n) > 0$. However, since the inherent risk is a product of centrality scores & combined CVSS scores, and the combined CVSS score for all nodes is 0 since no vulnerabilities exist, the inherent risk for all nodes is 0. Therefore, $Risk(app) = 0$. Alternatively, if there are no nodes in the call-graph $G = (V, E)$ & $V = \emptyset$, it logically follow that the sum of inherent risk scores over all nodes is 0.

- **Monotonicity with respect to vulnerabilities:** Assuming this property would not hold for the risk score, there should be a vulnerability $\sqsubseteq$, which when added to the set of vulnerabilities in a call-graph $\mathcal{V}$, the risk score for that application $Risk(app)$ would decrease. Since the combination method of CVSS scores in nodes is a monotonic function for all inputs ≥ 0, and all CVSS scores are in the range $[0, 10]$, no inherent risk score can decrease and therefore the $Risk(app)$ cannot decrease. It is good to note that adding notes/edges to the call-graph can decrease the risk score, since that could alter the relative/normalised centrality score for vulnerable nodes.

- **Traceable:** Since the risk score associated with a software application is a combination of the inherent risks of all methods reachable by the application, the contribution of vulnerable methods to the application risk score can be prioritised by the associated inherent risk scores. Further, for the *normalised* risk calculations, the inherent risk score divided by the application risk score, is the relative risk contribution in percentages. In order to find which vulnerabilities contribute most towards the risk score, the following formula can be used.

$$PR(v) = \sum_{n \in \{n | v \in vulnerabilities(n)\}} \frac{CVSS(v)}{\sum_{w \in vulnerabilities(n)} CVSS(w)} * \frac{R_{inherent}(n)}{\sum_{u \in V_{reachable}} R_{inherent}(u)} \tag{4.15}$$

4.8 Summary

In this chapter we have proposed four different models which can be used to calculate an estimation for risk regarding vulnerabilities in dependencies. These models are shown to uphold the criteria which we set for these models. Furthermore, from a small scale theoretical evaluation it is demonstrated how the different properties of the model affect the risk score, which shows that the there are different biases for the different methods in the models. The models using a normalised risk score tend to require a large portion of nodes in a call-graph to yield a significant risk score, whereas relative models are sensitive to only a few nodes being vulnerable but quite central. Further, the *co-reachability* centrality measure has a bias towards leaf nodes compared to *betweenness* centrality which is more sensitive to how uniquely a node connects components in the graph. In the next chapter we will further expand the evaluation to determine which model seems to be the most suitable for the risk estimation process, and measure its accuracy.

Chapter 5

Risk model evaluation

In the previous chapter, we have proposed four different models which could be used to estimate risk regarding vulnerabilities in dependencies of software projects. The goal of these risk models is to help developers of software projects be aware of security risks in their projects, and prioritise updates to dependencies which are most critical from a security perspective. In order to achieve that goal such a model needs to be accurate, since a high number of false positive warnings has a negative effect on the adoption rate of analysis tools [5]. More concretely we need to answer **RQ.2** *Can a risk model identify which vulnerable dependencies pose the highest risk for a software project?* Establishing the accuracy of risk estimators is a complex task, since risk is inherently subjective based on which aspects are important for the users and owners of a project. However, risk models have been proposed and evaluated in other domains.

One such model in a different domain is the model by Hong et al. [18] which estimates which vulnerabilities to fix first in a network of computers. In that paper, the proposed model is evaluated by comparing the prioritised list of vulnerabilities (PLV) calculated by the model, to the PLV as calculated by an *exhaustive search* over all possible attack scenarios. As the name exhaustive search suggests, this is a resource and time intensive process. In their paper, they analyse network graphs up to 100 nodes (hosts). The number of edges in their simulated graphs are not discussed, but it is very plausible that the topology of call-graphs generally differs from computer network graphs. Therefor we first perform an analysis, to determine the maximum call-graph size which can be used to feasibly compute an exhaustive search approach over, by investigating the question: *What is the maximum size call-graph for which exhaustive search can be computed?*

Then, we can use this to perform an in depth comparison of our four proposed risk models to establish which one is the most suitable for prioritising vulnerable dependency updates, for which we define the following question. *Which scalable risk model, based on graph topology and vulnerability information, is most suitable to estimate risk for software projects?*

Finally, we can compare our most suitable risk model with the model by Hong et al. [18] using their method of evaluation to determine accuracy and if it poses any benefit over existing approaches, which is the last question: *How accurate is our risk model in estimating risk of vulnerabilities in dependencies compared to other risk model approaches?*

5.1 What is the maximum size call-graph for which exhaustive search can be computed?

Computing all paths in a graph is a complex problem with a worst case run-time complexity of $O(n!)$ where n is the number of nodes in a graph and an exponential number of paths, if the graph is fully connected. Therefore, computing an *exhaustive search* over all paths in a call-graph is currently infeasible. For instance, the average number of nodes in the call-graphs which we analysed in section 3.3 is 14375 nodes, see Table 5.1. Computing an exhaustive search over graphs with 200 nodes can already take well over 3 hours [18]. In order to be able to compare our risk models to an *exhaustive search* ground truth we need to work with sub-graphs of the original call-graph. We base our evaluation on the method used by Hong et al. [18]. In their case, they use a maximum graph size of $n = 100$. Since call-graphs can differ in topology, we further investigate the performance of *exhaustive search* for samples of the call-graphs of the vulnerable projects as found in section 3.3. The goal is to have a single graph size threshold which is representative of the whole call-graph with regards to risk of vulnerabilities, so that we can use it to determine the accuracy of the proposed models. Using the same graph size threshold for all projects ensures that we can compare the risk calculation results between projects.

Methodology In order to investigate the performance of *exhaustive search* on samples of call-graphs, we first need to generate these sub-graphs. Generating representative sub-graphs can be done using a *forest fire sampling* method with a 70% burn rate [22]. Since we are interested in risks posed by vulnerabilities, we can start the *forest fire sampling* from the vulnerable nodes in the call-graph. Further, the evaluation method used in [18] performs an exhaustive search over all possible attack paths. Attack paths are paths in the graph which contain vulnerable nodes. The risk calculations for exhaustive search then consider the sum of maximum vulnerability scores per node in each path. Since only vulnerable nodes have a non-zero vulnerability score, we can restrict the *exhaustive search* to all paths from a vulnerability to another vulnerability (including paths of length 1).

We compute all attack paths in sub-graph sizes in vicinity of the threshold used in the evaluation by Hong et al. [18] to establish whether the threshold of 100 nodes is suitable for samples of call-graphs. Concretely, we investigate the exhaustive search over all attack paths for sub-graph sizes ranging from $n = 10$ to $n = 250$. However, if the exhaustive search for a single sub-graph takes longer than half an hour we terminate the search for that sub-graph, and subsequent sub graph sizes for that project. This timeout is maintained to keep the run-time of the analysis manageable, while still being able to use the resulting curves of run-time over number of nodes to establish the a proper graph size threshold. This results in a curve corresponding to how the run-time of *exhaustive search* increases as the graph size grows. Further, we investigate the number of attack paths identified by *exhaustive search* for the different sub-graph sizes. If the number of attack paths remains stable at a certain graph size, we can use that size for all further investigations.

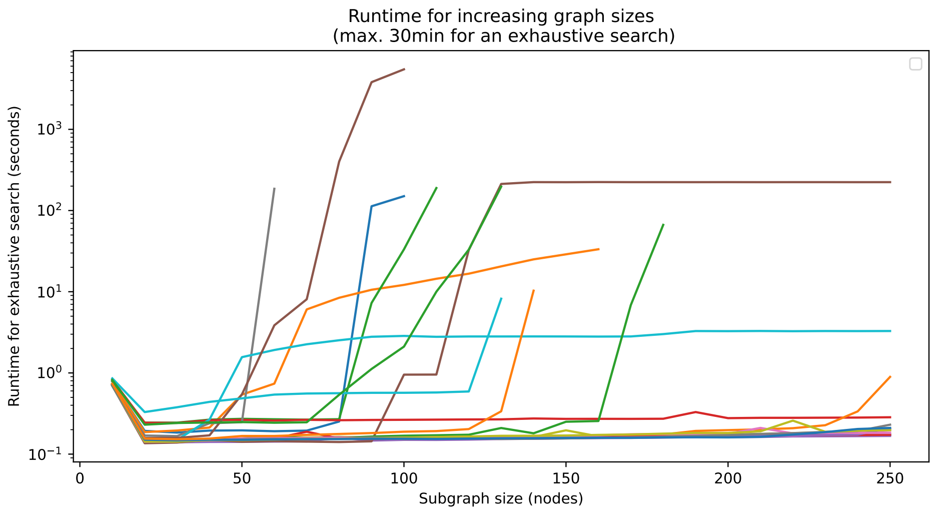

Figure 5.1: Run-time of calculating all paths between all vulnerable nodes for increasing graph sizes (in terms of nodes in the graph). Each line depicts the runtime curve of a single project.

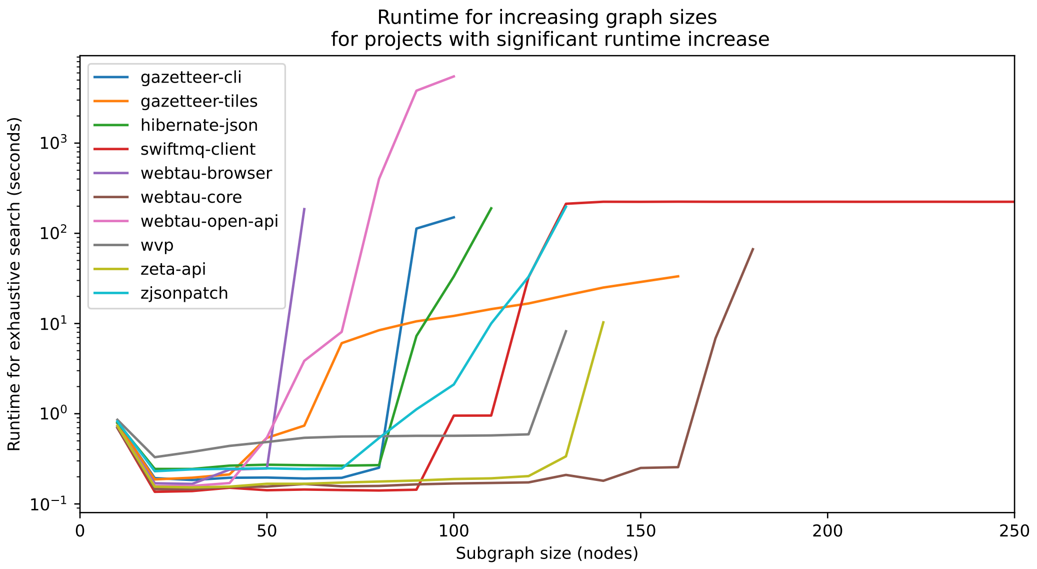

Figure 5.2: Run-time of calculating all paths between all vulnerable nodes for increasing graph sizes (in terms of nodes in the graph). Each line depicts the runtime curve of a single project. Plotted for projects with significant increases in runtime to identify those projects.

Table 5.1: Call-graph properties of the analysed projects

project	nodes	edges	vulnerable nodes	vulnerabilities
wvp	52148	838500	63	31
pg2k4j	18893	169381	8	18
hibernate-json	19826	245458	8	18
zjsonpatch	6397	70934	7	17
bigtable-autoscaler	66284	1190803	8	15
gazetteer-tiles	26893	271166	3	8
gazetteer-cli	28801	297380	3	8
webtau-cli	6016	31917	6	4
unity-types-api	14112	79983	3	4
webtau-http	6654	33662	6	4
webtau-browser	24258	383499	6	4
webtau-open-api	21742	226606	6	4
webtau-report	4615	26009	6	4
webtau-cache	5806	31192	6	4
webtau-core	6238	31495	6	4
webtau-test-server	7944	38250	6	4
webtau-config	4450	22688	6	4
webtau-console	4173	21841	6	4
webtau-utils	4362	22055	6	4
webtau-java-runner	4565	25927	6	4
wechat-mp	20789	396669	3	2
x8l	28686	930876	22	1
spring-boot-swing-reservations	21970	269271	1	1
zeta-api	50622	498365	20	1
docker-client	29525	537244	1	1
swiftmq-client	18965	127737	22	1

Results As can be seen in Figure 5.1, a majority of projects have no problem calculating all possible attack paths up to graph sizes with 250 nodes. However, there are a number of projects for which calculating all attack paths for graphs smaller than 250 nodes takes longer than the 30 minute run-time limit. These projects are plotted separately for clarity in Figure 5.2. There are two major factors which cause the rapidly increasing run-time: the edges per node density in the sub-graph, and the number of vulnerable nodes. From previous work, it is known that generating all paths for a fully connected graph is bounded by $n!$, where n is the number of nodes. It follows that the more edges per node in a graph, the run-time for generating all paths increases. Further, since we generate only possible attack paths, paths originating and ending in a vulnerable node, higher number of vulnerable nodes in a graph increases the run-time of the exhaustive search.

As mentioned before, the worst case for the run-time of an exhaustive search over all paths in a graph ($O(n!)$) occurs when the graph is fully connected. To further investigate

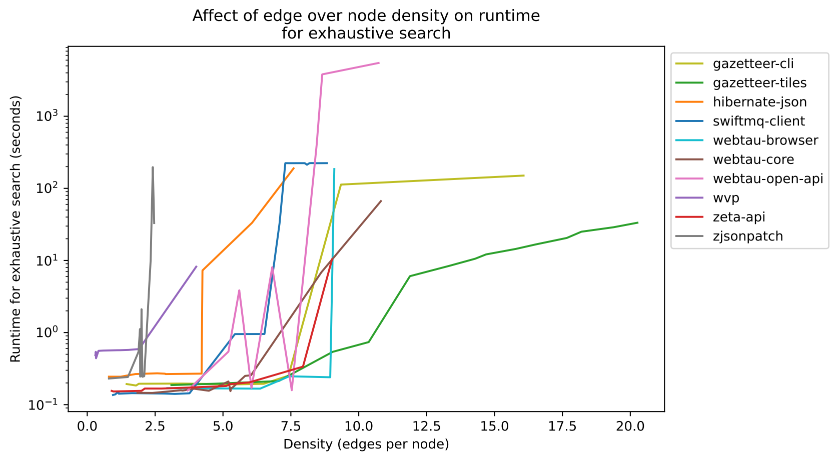

Figure 5.3: The affect of edges per node density on the runtime of an exhaustive search over all attack paths, for increasing sub-graph sizes in terms of nodes.

the differences in run-time for exhaustive search for the different projects, we can inspect the relation between the density in a graph in terms of edges per node and the run-time for exhaustive search for that graph. This is shown in Figure 5.3. In this figure it can be seen that an increase in edges per node means an increase in run-time for exhaustive search. However, there are instances where the run-time does increase for similar edge per node densities such is the case for *webtau-browser* project. Therefore, it is possible there are other factors which play a major role in the increasing run-time, and future investigations could look into these factors.

In order to establish an appropriate maximum size for sub-graphs it is important to understand how an increase in sub-graph size corresponds with additional attack paths in the aforementioned sub-graph. The number of attack paths found in increasing sub-graph sizes is shown in Figure 5.4. An important observation is that the number of attack paths remains stable for $n > 75$. From [22], it is known that larger sub-graphs are more representative of the original graph. Further, computing an exhaustive search over graphs of size $n \leq 100$ is trivial. Therefore, we use 100 nodes for all sub-graph sizes, hich also matches the threshold as used by Hong and Kim [17] in their evaluation.

5.2 Which scalable risk model, based on graph topology and vulnerability information, is most suitable to estimate risk for software projects?

In the previous chapter we have proposed four different risk models based on centrality metrics and CVSS scores. In order to find which one of these models is most suitable to estimate risk regarding vulnerabilities in dependencies of software projects, we perform the following analysis. First we compare how the *normalised* and *relative* aggregation ap-

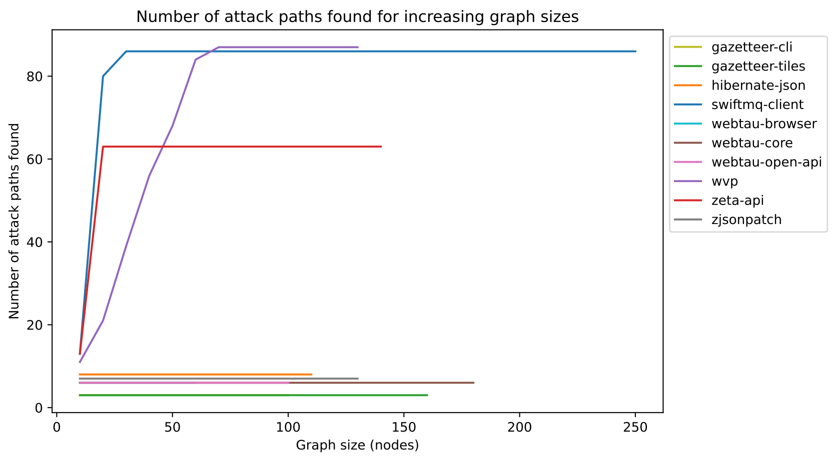

Figure 5.4: The number of attack paths found by exhaustive search in increasing sizes sub-graphs.

proaches relate to the density of vulnerable nodes in a call-graph. Further, we compare the *co-reachability* and *betweenness* centrality metrics with an exhaustive search approach to determine which is most accurate for estimating importance of nodes in an attack path.

Methodology In order to determine which of our four risk models is most suitable for estimating risk of vulnerabilities in dependencies using call-graphs, we use two metrics which determine the suitability of a risk model based on the topology and characteristics of a call-graph. These metrics are the density of vulnerable nodes in the call-graph, and similarity of centrality measures to the importance of nodes as computed by an exhaustive search. We perform this analysis in the following steps.

From section 4.6, we know that the *relative* models are more suitable for graphs where the number of vulnerable nodes is sparse, whereas *normalised* models work better with a higher density of vulnerable nodes. We can investigate the call-graphs of the vulnerable projects identified in section 3.3 to get a representative idea of the density of vulnerable nodes in the call-graph, which we can use to determine which of these two approaches is the most suitable.

In order to find the most suitable centrality metric for the risk model, we can compare the centrality measures on which they are based with an exhaustive search calculation over all paths in the graph. The exhaustive search can serve as a ground truth for the application of centrality measures for risk estimation as described in Gonda et al. [14]. Since exhaustive search over all paths in a graph has a run-time complexity of $O(n!)$ for n nodes in the graph, we cannot compute this for entire call-graphs. Therefore, we will need to construct sub-graphs from original call-graphs with a more limited number of nodes. In accordance with the findings in section 5.1 we use sub-graphs of size $n = 100$, which are sampled from the original call-graph using forest fire sampling with a burn rate of 70%

For each of the vulnerable projects identified in section 3.3, we generate a sub-graph for

Table 5.2: Spearman correlation between the *betweenness* and *co-reachability* centrality metrics and an exhaustive search over all paths in subgraphs with 100 nodes

		Betweenness		*Co-reachability*	
project	vulnerability density	correlation	P-value	correlation	P-value
wechat-mp	1.44×10^{-4}	0.81	0.00	0.63	0.00
swiftmq-client	1.16×10^{-3}	0.88	0.00	0.57	0.00
hibernate-json	4.04×10^{-4}	0.23	0.02	0.32	0.00
webtau-http	9.02×10^{-4}	0.68	0.00	0.52	0.00
webtau-utils	1.38×10^{-3}	0.93	0.00	0.43	0.00
gazetteer-tiles	1.12×10^{-4}	0.69	0.00	0.57	0.00
webtau-report	1.30×10^{-3}	0.82	0.00	0.68	0.00
webtau-console	1.44×10^{-3}	0.81	0.00	0.56	0.00
unity-types-api	2.13×10^{-4}	0.84	0.00	0.05	0.61
zeta-api	3.95×10^{-4}	0.96	0.00	0.03	0.79
wvp	1.21×10^{-3}	0.99	0.00	0.69	0.00
webtau-config	1.35×10^{-3}	0.92	0.00	0.37	0.00
webtau-core	9.62×10^{-4}	0.88	0.00	0.61	0.00
webtau-cli	9.97×10^{-4}	0.87	0.00	0.62	0.00
pg2k4j	4.23×10^{-4}	0.97	0.00	0.55	0.00
gazetteer-cli	1.04×10^{-4}	0.83	0.00	0.34	0.00
webtau-cache	1.03×10^{-3}	0.87	0.00	0.63	0.00
mean	7.95×10^{-4}	0.82		0.48	

the entire graph of size 100, and calculate the *co-reachability* and *betweenness* centrality. Further, for each of these graphs we generate all paths in the graph, and then assign an *exhaustive search score* based on the number of paths which pass through a node (including start and endpoints) over the total number of paths in the graph.

These two steps will yield the combination of aggregation approach and centrality measure which is most suited for the purpose of estimating risk of vulnerable nodes in a call-graph. The best model will be the one which combines the results of analyses performed by the aggregation approach and centrality measure.

Results The results for the relevant projects is described in Table 5.2. The first thing to note is the overall low density of vulnerable nodes in the inspected call-graphs. The mean percentage of vulnerable nodes in the call-graphs for these projects is quite low, with only on average 0.08% of the nodes being vulnerable. It is clear that vulnerable nodes only sparsely populate call-graphs of software projects with vulnerabilities. Considering the sparsity of vulnerable nodes, risk models based on the *normalised* risk aggregation method are less suitable for vulnerability risk analysis.

The second step in this analysis looks into the best centrality measure (*Betweenness* centrality or *co-reachability*) to use in the risk model. In order to investigate this issue, we calculated the Spearman correlation between the *exhaustive search score* and the two centrality measures. This calculated correlations can be seen in Table 5.2. *Betweenness* centrality shows an overall strong correlation with the *exhaustive search scores*, with a mean correlation of 0.82 over all projects. *Co-reachability* shows a much weaker correlation with a mean correlation score of 0.48. Further, in two cases there is no clear correlation between co-reachability and all paths (P-value of 0.61 and 0.79), and thus do not estimate the importance of a node.

This is not unexpected considering *betweenness* centrality is based on the number of shortest paths passing through a node and quite comparable to the approach of the *exhaustive search scores* which considers all paths. *Co-reachability*, however, is only affected by the portion of the graph which can reach a specific node and is thus quite different.

From this information it is clear that *betweenness* centrality is more suitable for a scalable risk model. This is also in line with intuition since it is important to incorporate in how many different ways a single method can be called, which will increase with more subsequent nodes. Thus, from the four risk models we proposed in the previous chapter, model D based on *betweenness* centrality scores with a *relative* risk aggregation approach is the most suitable.

5.3 How accurate is our risk model in estimating risk of vulnerabilities in dependencies compared to other risk model approaches?

A key feature of our risk model is to prioritise which vulnerable dependencies needs to be updated first. In order to evaluate this aspect of the risk model we compare our model to the Hierarchical Attack Representation Model (HARM) by Hong et al. [18]. The HARM introduced in that paper can calculate risk for an entire computer network, based on the possible attack paths a malicious actor could take from their host to a specified goal host. Further, we use the same method of evaluation by comparing the Prioritised List of Vulnerabilities of these two models to an *exhaustive search* over samples of the call-graphs of the vulnerable projects identified in section 3.1.

The goal of our risk model is very similar to the model by Hong et al. [18], determining which vulnerability needs to be fixed first in order to reduce risk as much as possible. The model and evaluation in that paper are in a different domain. Between these domains there exist several differences. For instance, one key difference is that while one vulnerability can infect multiple hosts in a network, fixing that single vulnerability requires fixing it on all hosts individually. Whereas a single vulnerability affecting multiple nodes in a call-graph can be fixed with one update.

However, both domains proof to be quite similar. In both approaches nodes in the graph can contain any number of vulnerabilities. Further, computer networks can be modelled as graphs where hosts are nodes in the network and a connection between hosts is an edge in the graph. This is similar to how we denote call-graphs, but in said model, methods are nodes and method-calls are the edges. As such, we think that this is a good methodology for our evaluation and a great baseline.

Methodology In order to determine the accuracy of our model and compare it to existing vulnerability prioritisation methods, we can compare our risk model to the model and exhaustive search approaches proposed by Hong et al. [18]. Therefore we propose the following methodology:

In order to map the problem from computer networks to call-graphs, we have to establish which nodes serve as entrypoints and which ones are the target nodes between which all simple paths are calculated. In the computer network scenario, these are clearly defined. Entrypoints are the hosts to which a malicious attacker has access and the target nodes are private servers such as user information databases. These entrypoint and target nodes are not as clearly defined for the call-graphs, especially in sub-graphs.

However, the exhaustive search in this evaluation is used to obtain all potential attack paths a malicious actor can use. The risk score for the exhaustive search is then the sum of Impact Scores of vulnerable nodes in the attack path with the maximum sum. Since only inherently vulnerable nodes contribute to this risk score, the exhaustive risk can be restricted to the portion of the path containing vulnerable nodes. Further, because all nodes in the call-graphs we analyse have been tested for reachability, each node can be reached. Therefor we can compute all possible paths from each vulnerable node to every other vulnerable node, including path length 1, while maintaining the same risk score.

We compare three approaches (i). Exhaustive search, (ii). The HARM, and (iii). Our risk model D. Each of those approaches yields a Prioritised List of Vulnerabilities (PLV), the list of vulnerabilities ordered by which vulnerability needs to be fixed first. In order to compare these PLVs, we can calculate the Rank Biased Overlap (RBO) [31]. Rank Biased Overlap [31] is suitable similarity metric for this purpose, which places high weight on the highest priority items, and can handle non-overlapping lists. Further, this metric takes into accoun the order of elements which is of importance since we compare prioritised lists. The high weight on the items with highest priority is also essential as is clear from the findings of Christakis and Bird [8]. From those findings it is clear developers usually only act upon short list of actionable advice which means the first few items in the PLV are the most important to be correct. From the RBO similarity scores, we can evaluate which approach is the most suitable for calculating risk related to vulnerable nodes in a call-graph.

In short, we have three different models (*Model D*, *HARM*, and *Exhaustive Search (ES)*) each calculates a list of vulnerabilities prioritised by which need to be fixed first in order to reduce the risk as calculated for that model as much as possible. In order to evaluate the accuracy of these models we compare *Model D* and *HARM* to the *Exhaustive Search* approach, by calculating the similarity of the prioritised list of vulnerabilities to fix. These similarity scores serve as an estimation of the accuracy of the model and can therefore determine which of the two models is most suited for helping developers mitigate risk of vulnerabilities

Results The results for the comparison can be seen in Figure 5.5. In nearly all cases Model D matches the PLV for Exhaustive Search completely. Further, it consistently outperforms the HARM, suggesting that our risk model has a high accuracy and is well suited to help developers prioritise updating the vulnerable dependencies which actually pose the highest risk. This is not to surprising considering the HARM was created for an other domain,

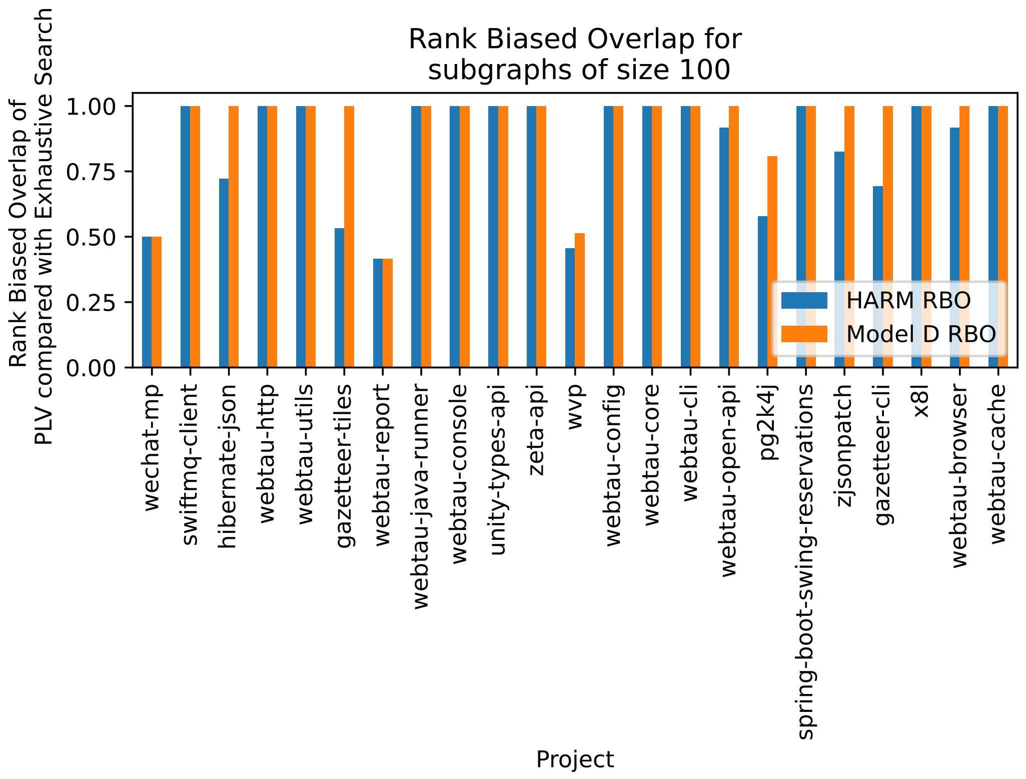

Figure 5.5: Similarity of Prioritised Set of Vulnerabilities between the model by Hong et al and Exhaustive Search (blue), our Model D and Exhaustive Search (orange) for sub-graphs with 100 nodes.

further exemplifying the need for a risk model for estimating the risk of vulnerabilities in dependencies of software projects.

However, for most projects in this analysis there are only a limited number of vulnerabilities present in the call-graph, see Table 5.1. In most cases, there are therefore not many degrees of freedom in the PLV, making it quite easy to generate matching PLVs. This is further demonstrated in the cases where Model D and Exhaustive Search do not generate the same PLV.

An important factor for the cases where our model differs from exhaustive search approach, is the number of distinctive vulnerabilities present in the graph. Most graphs have at most 8 distinct vulnerabilities, while *wvp*, *hibernate-json*, *pg2k4j* contain 31, 18, 18 distinct vulnerabilities respectively (see Table 5.1). Therefore in general for software projects we would expect our risk model to not be 100% accurate, but should still pose an improvement in the majority cases compared to the HARM.

In addition, the number of nodes of the original call-graph is also an important factor. For example, *zjsonpatch* contains 17 vulnerabilities but the PLV of our risk model is the same as the PLV of the exhaustive search model. The key difference being that the original

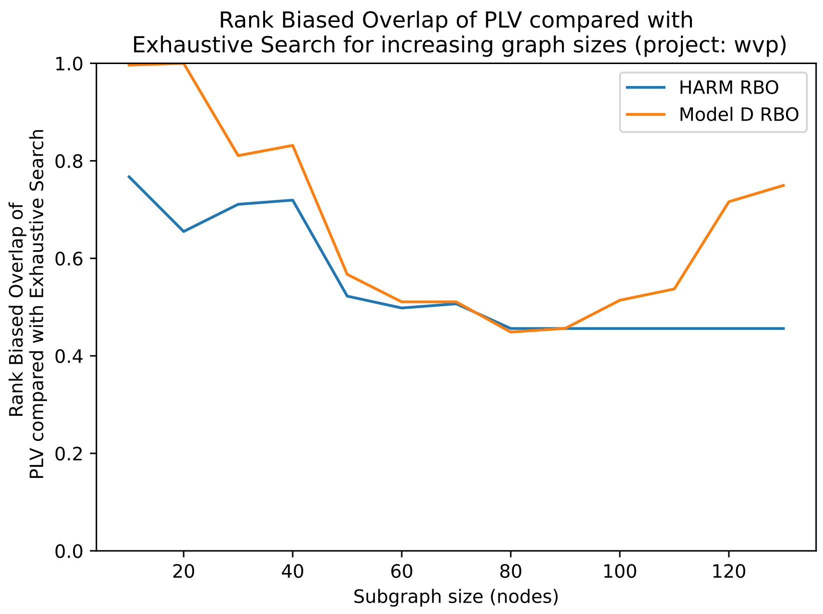

Figure 5.6: The affect of different sub-graph sizes in terms of nodes on the RBO between the PLV's of the three models.

call-graph of *zjsonpatch* only contains 6400 nodes, while the aforementioned projects all have around 20 000 nodes.

Lastly, to verify whether using sub-graphs with 100 nodes for the analysis is not impacting the results we investigated the effect of increasing graph sizes on the RBO scores. In order to investigate this, we compare the RBO scores for increasing sub-graph sizes of the call-graph of *wvp* which contained the most vulnerabilities. This is visualised in Figure 5.6, in which can be observed that the RBO score for the HARM remains constant, while the RBO for our risk model D does increase with larger sub-graphs though this increase is limited.

From these findings it is clear that our risk model D is well suited for prioritising vulnerability updates. The proposed model has a mean RBO of 0.92 when comparing the list of prioritised vulnerabilities with an exhaustive search approach. This is a significant 8% increase over the mean RBO of 0.85 for the HARM which we compared our model to.

5.4 Summary

In this chapter we set out to find if risk models can help developers in mitigating risk by identifying which vulnerable dependencies pose the highest risk for a software project. In section 5.2, we show which of the four risk models defined in the previous chapter is most suited for risk estimation for software projects. Based on the density of vulnerable nodes in the call-graph and comparison with exhaustive search to centrality metrics, model D is the most suitable.

Further, we analysed how our risk model perform compared to existing risk estimation models by investigating the accuracy of our model and the accuracy of the HARM proposed by Hong et al. [18]. In order to measure the accuracy of each model we compared the prioritised lists of vulnerabilities which should be fixed for each model to an exhaustive search approach. In this analysis we show that our model is has a mean similarity to the exhaustive search approach of 0.92 in terms of identifying which vulnerable dependencies must be updated first, which is an 8% average increase to the results of the HARM. This indicates that our most suitable proposed risk model can determine with high accuracy which vulnerable dependencies must be update first. However, this evaluation was limited to a theoretical comparison and therefore future work should investigate how well our risk model performs in practice.

Chapter 6

Discussion

In this work we set out to help developers in maintaining secure software projects by estimating the risk of vulnerabilities present in dependencies of such projects. Vulnerable dependencies are a growing problem in software development, and should therefore be minimised. In order to understand how we can help developers in mitigating vulnerable dependencies, we performed an investigation into the current update behaviour of developers.

First, we investigated the extent to which CVSS scores play a role in the current update management process. We investigated this by comparing the relative speed with which dependencies are updated to the CVSS score of vulnerabilities present in dependencies. From this investigation into CVSS scores, we found that CVSS scores currently do not affect the update behaviour of developers. Further, we find that actually depending on the vulnerable pieces of code similarly has no clear effect on the update behaviour. This is in line with the findings by [21], which finds that developers are generally unaware of vulnerabilities in the dependencies of their project. However, Cox et al. [9] have found a relation between a dependency containing a vulnerability and the freshness of that dependency, which would suggest vulnerability information is actually used in this process. A future study could compare the *relative update delay* with the dependency freshness proposed in that paper to investigate this discrepancy in order to determine which factors are responsible for the difference in measured update behaviour.

In the investigation into changes in update behaviour with respect to vulnerable dependencies we used our own metric *standardised update delay* instead of an established metric such as *dependency freshness* [9] or one of the metrics by Kula et al. [21]. We have not used one of these established metrics because there are slight differences between what we want to measure, and what the metrics in these works measure. The metrics by Kula et al. [21] are concentrated around the number of projects which rely on a specific library version, from which they can determine how developers react to a new version of a library. Further *dependency freshness* is mostly used to estimate how up to date a project is with new versions of its dependencies. These are slightly different from what we need in that we want to measure the priority assigned to a specific dependency update in a software project. Which we estimate using the speed with which an update is performed to a dependency relative to other dependency updates in a software project in the *standardised update delay*. While the results of our investigation using this metric are in line with previous work, future research could be performed to determine if the *standardised update delay* does indeed accurately estimate the priority of a dependency update. This could be done by forming a dataset with dependency updates and their *standardised update delays* as well as a priority label, which could perhaps be derived from pull-request or associated issue labels on GitHub.

The prevalent non-usage of CVSS scores in the update management process begs the question if CVSS scores are even suitable for the purpose of helping developers mitigate security issues in their dependencies. As stated by the CVSS user guide, the score alone should not be used on itself as a risk estimate. Maybe the scores are interesting for maintainers of the libraries in which vulnerabilities are disclosed.

However, most project maintainers seem to be unaware when an update to a dependency fixes a security vulnerability, and were therefore at risk before the update. With the ever increasing number of dependencies of software projects, this is an alarming insight, and any effort into raising awareness should be top priority. As stated, considering a large number of vulnerability warnings are not actually relevant for developers, considering the dangerous pieces of code in a vulnerable library are often not depended upon. Ultimately, we tried to answer the question: *Is vulnerability information currently taken into account by developers of software projects in the update management process?* From the analyses in chapter 3, we know that, currently, vulnerability information is not taken into account in this process. Therefore, to help developers be aware of the actual risk posed by vulnerable dependencies and how to most efficiently mitigate against that risk, we proposed a risk model. By decreasing the false positive rate as well as providing a short actionable list, this model can raise adoption rate and thus awareness.

Our proposed risk models use call-graph topology and CVSS scores as input for the risk estimation. This approach is therefore limited to vulnerabilities that have been disclosed, and cannot detect insecure code in and of itself. Since the security advisories are only available for publicly available software, insecurities cannot be detected in private dependencies.

In order to asses if the models we proposed can help developers in the future in their dependency management process to reduce risks for their software projects, we performed a similar analysis as used in [18], where we compare the results of our risk model with their model and an *exhaustive search* approach. The *exhaustive search* approach in this evaluation serves as ground truth to compare the models to. This approach prioritises the highest score vulnerabilities in the most risky attack path. A key assumption in this evaluation is that risk calculations for call-graphs of software projects are similar enough to risk calculations in computer networks. There are numerous resemblances between these domains in how their graph structure models the communication between pieces of software. However, there are also key differences such as Firewalls and VLANs in computer networks which can restrict access to certain parts of the graph. From this assumption, it also follows that it is currently not entirely clear if the exhaustive search does indeed correspond fully with the real world risks, future work should further test if this theory holds up in practice.

We have investigated what a suitable sub-graph size is on which to perform the accuracy of our risk model to the *exhaustive search approach*, which is in line with the sub-graph size used in the evaluation by Hong et al. [18]. However, in the run-time investigation exhaustive search over all attack paths was trivial to calculate for 14 of the 23 analysed projects on sub-graphs with 250 nodes. Further investigation could investigate the run-time complexity for attack paths in a call-graph to better understand the factors that contribute to the run-time increase.

We proposed the risk models to determine the following: *Can a risk model identify which vulnerable dependencies pose the highest risk for a software project?* With the evaluation from chapter 5, we know that the proposed model is an accurate estimator for determining which vulnerabilities pose the highest risk. This follows from the high overall Rank Biased Overlap similarity between the PLVs our risk model and the *exhaustive search*. Therefore, this model should theoretically be able to help developers in the update process, and thereby see increased adoption and thus raised awareness. In the investigation into the accuracy of our risk model we have used a fixed threshold of 100 nodes for each project subgraph. However, with these differences in run-time between projects it could be interesting to use a different threshold per project, and investigate if that has a meaningful impact on the accuracy results.

However, in this research we were not able to investigate the practical usage of the risk model. There are two assumptions which the current method of evaluation was not able to verify. The first is that distance in a call-graph is irrelevant for vulnerabilities; vulnerabilities in direct and transitive dependencies are treated equally. Further, we assume that the call-graph topology does indeed have an influence on the actual risk of a vulnerability in a software application. Additionally, using only call-graphs and function level vulnerability information as inputs for the risk model does not fully incorporate the context in which the software application is being used. Software applications are used in numerous different settings, and this can have a high effect on the perceived risk.

Future research should look into a more practical evaluation to verify these assumptions. A study can be performed where developers are given access to the model and investigate whether that significantly changes their update behaviour and increases awareness with respect to vulnerable dependencies. In this experiment, a group of developers of projects could be asked to use the risk estimation in their dependency management process, and another group asked to use a tool which reports the CVSS scores of present vulnerabilities. This would allow to compare the dependency update behaviour of these groups and assess whether or not risk scores serve a better basis for vulnerability information in the dependency management process.

Another interesting idea for future work which could be conducted using the proposed risk model, is to use the model to analyse whether they could have been useful in scenarios where vulnerable code was actually exploited. By running the models on projects which were actually affected by vulnerabilities and verifying whether the models correctly estimate a high risk score for the vulnerable dependency which was exploited.

Threats to validity Threats to *construct validity* mainly concern the differences between theory and real observations. In this work, these threats are mostly comprised of the following.

Implementation Bugs. Our implementation for the risk models, and the reachability analysis for CVSS scores might contain bugs that cause these models to falsely report on vulnerabilities.

Vulnerability information. The proposed risk models rely on the function level vulnerability information and reachability analysis thereof. However, this information is not necessarily complete and could therefore yield false negatives. A good example of this property is the vulnerability which was responsible for the Equifax data breach (CVE-2018-11776). The function level vulnerability information for this weakness was present in the FASTEN KB. However, considering it was a configuration issue, the call-graph analysis could not have pointed out that the vulnerability is affecting projects, but, as is clear from the data breach, it in reality was affecting projects and exploitable by malicious actors. If a high enough percentage of vulnerabilities are such vulnerabilities, the analysis based on call-graphs and function level vulnerability information is not entirely correct.

Threats to *internal validity* which consider if other extraneous factors could explain causal inferences in our study. In particular, the CVSS score investigation as well as the quantitative risk model evaluation could be affected by our selection of projects. The main way we have mitigated this is by selecting a random diverse set of open source projects for the analysis from libraries.io[1]. Further, we selected for maturity for these projects by using the popularity based on the *SourceRank*[2] metric from Libraries.io.

Another threat to internal validity is the fact that we draw our conclusions regarding update behaviour based on the relative speed with which projects perform dependency updates. However, as stated before other metrics such as dependency freshness by Cox et al. [9] exist, and in this work a relation is observed between the *dependency freshness* and whether that dependency contains, or used to contain, a vulnerability. It is thus possible that the metric we used skewed our findings, for example due to the associated development cost of performing a dependency update which the used metric does not take into account.

Lastly, the threats to *external validity* are focused on factors which influence the extent to which the findings of this work can be generalised to different situations.

Sample Size and Diversity. Our evaluation would surely benefit from the analysis of a larger sample of projects. In this work, the analysis of projects had technological limitations (Java, MAVEN) and do not affect the relevance of the detected anti-patterns, as explained in Section IV.

[1]`https://libraries.io`

[2]`https://libraries.io/data#repositories`

Chapter 7

Summary

In this thesis we set out to help developers in their dependency update management process by researching risk models based on CVSS scores as well as centrality measures from a call-graph from the application. We first investigated the update behaviour of developers for vulnerable dependencies based on CVSS scores and the usage of vulnerable code. Based on this investigation, as well as previous work, we find that developers of software projects are generally unaware of security risks posed by vulnerabilities in dependencies of such projects. Further to answer **RQ.1**: *Is vulnerability information currently taken into account by developers of software projects in the update management process?*. We conclude that vulnerability information is currently not actively taken into account for prioritising and performing dependency updates.

Our proposed risk model can help in notifying developers when their projects actually are at risk. One major problem in using CVSS scores for this purpose is that these are scores generalised to describe severity across all different use cases, and that a project can contain a vulnerability on a package level, but not actually depend on the vulnerable code. These are two areas in which the proposed risk models can help. Using function level vulnerability information from the FASTEN Knowledge Base in combination with call-graphs of software projects, only the vulnerabilities which actually affect the software project are reported, which should result in less false positives compared to CVSS. Further, the proposed models do use a combination of CVSS scores and centrality measures on call-graphs to estimate the risk of associated with a vulnerability, and propagate that to the risk for an entire software project. In our analysis we find that a risk model based on *betweenness* centrality in combination with a relative risk score aggregation approach is the most suitable for vulnerability risk estimation. Further, when compared to existing state of the art risk models existing in different domains that our best model is around 8% more accurate. The evaluation was performed to answer **RQ.2**: *Can a risk model identify which vulnerable dependencies pose the highest risk for a software project?* In the theoretical evaluation we show that our most suitable risk model can accurately identify highest risk vulnerabilities and should therefore serve useful for developers in prioritising updates to high risk vulnerable dependencies.

Bibliography

[1] M. U. Aksu, M. H. Dilek, E. İ. Tatlı, K. Bicakci, H. İ. Dirik, M. U. Demirezen, and T. Aykır. A quantitative cvss-based cyber security risk assessment methodology for it systems. In *2017 International Carnahan Conference on Security Technology (ICCST)*, pages 1–8, 2017. doi: 10.1109/CCST.2017.8167819.

[2] Mohammed Alhomidi and Martin Reed. Attack graph-based risk assessment and optimisation approach. *International Journal of Network Security & Its Applications*, 6 (3):31, 2014.

[3] Luca Allodi, Marco Cremonini, Fabio Massacci, and Woohyun Shim. Measuring the accuracy of software vulnerability assessments: experiments with students and professionals. *Empirical Software Engineering*, 25(2):1063–1094, 2020.

[4] Tiago Miguel Laureano Alves. Benchmark-based software product quality evaluation. *Univerisy of Minho, Portugal*, 2012.

[5] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481. IEEE, 2016.

[6] Lionel C Briand, Sandro Morasca, and Victor R Basili. Property-based software engineering measurement. *IEEE transactions on software Engineering*, 22(1):68–86, 1996.

[7] Kevin Burke. Maybe automatically updating dependencies isn't a great idea. `https://kevin.burke.dev/kevin/dont-update-dependencies/`, 2015.

[8] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 332–343, 2016.

[9] Joël Cox, Eric Bouwers, Marko Van Eekelen, and Joost Visser. Measuring dependency freshness in software systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 109–118. IEEE, 2015.

[10] Kousik Das, Sovan Samanta, and Madhumangal Pal. Study on centrality measures in social networks: a survey. *Social network analysis and mining*, 8(1):1–11, 2018.

[11] Michel Denuit, Jan Dhaene, Marc Goovaerts, Rob Kaas, and Roger Laeven. Risk measurement with equivalent utility principles. *Statistics & Risk Modeling*, 24(1): 1–25, 2006.

[12] Linton C Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[13] Bergmans L. Gao C., Zivkovic M. A model for quality and risk propagation and aggregation. *FASTEN deliverables*, 2020.

[14] Tom Gonda, Tal Pascal, Rami Puzis, Guy Shani, and Bracha Shapira. Analysis of attack graph representations for ranking vulnerability fixes. In *GCAI*, pages 215–228, 2018.

[15] Max Hinne. Local approximation of centrality measures. *Radboud University Nijmegen, The Netherlands*, 2011.

[16] John L Hodges. The significance probability of the smirnov two-sample test. *Arkiv för Matematik*, 3(5):469–486, 1958.

[17] Jin B Hong and Dong Seong Kim. Scalable security analysis in hierarchical attack representation model using centrality measures. In *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 1–8. IEEE, 2013.

[18] Jin B Hong, Dong Seong Kim, and Abdelkrim Haqiq. What vulnerability do we need to patch first? In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 684–689. IEEE, 2014.

[19] Michael Howard and David LeBlanc. *Writing secure code*. Pearson Education, 2003.

[20] Panayiotis Kotzanikolaou, Marianthi Theoharidou, and Dimitris Gritzalis. Cascading effects of common-cause failures in critical infrastructures. In *International Conference on Critical Infrastructure Protection*, pages 171–182. Springer, 2013.

[21] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2018.

[22] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636, 2006.

[23] Paerit Garg Loren Kohnfelder. The threats to our products. *Microsoft Security Blog*, 1999.

[24] Joachim Meyer, Omer Dembinsky, and Tal Raviv. Alerting about possible risks vs. blocking risky choices: A quantitative model and its empirical evaluation. *Computers & Security*, 97:101944, 2020. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2020.101944. URL `http://www.sciencedirect.com/science/article/pii/S0167404820302200`.

[25] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[26] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, pages 1–41, 2020.

[27] Karen Scarfone and Peter Mell. An analysis of cvss version 2 vulnerability scoring. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 516–525. IEEE, 2009.

[28] David Shepardson. Equifax failed to patch security vulnerability in march: former ceo. `https://www.reuters.com/article/us-equifax-breach/equifax-failed-to-patch-security-vulnerability-in-march-former-ceo-idUSKCN1C71VY`, 2017.

[29] V. L. Shivraj, M. A. Rajan, and P. Balamuralidhar. A graph theory based generic risk assessment framework for internet of things (iot). In *2017 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*, pages 1–6, 2017. doi: 10.1109/ANTS.2017.8384121.

[30] C Team. Common vulnerability scoring system v3.1: Specification document. *First. org*, 2019.

[31] William Webber, Alistair Moffat, and Justin Zobel. A similarity measure for indefinite rankings. *ACM Transactions on Information Systems (TOIS)*, 28(4):1–38, 2010.

[32] Douglas R White and Stephen P Borgatti. Betweenness centrality measures for directed graphs. *Social networks*, 16(4):335–346, 1994.

[33] Kim Wuyts and Wouter Joosen. Linddun privacy threat modeling: a tutorial. *CW Reports*, 2015.