Risk in Open-Source Projects

Practicum Paper - 22-07-2024

Róisín Ní Bhriain
Student Number: 23269640

***Abstract*—Measuring risk is essential in ensuring the security, reliability and compliance of software systems. Open-source software is both transparent and accessible but may nevertheless be vulnerable to security risks such as software vulnerabilities. Open-source software thus requires analysis to identify and mitigate these risks. Developers and organisations need to understand the risks associated with their open-source supply chain to safeguard against any data breaches or other attacks. Making informed decisions on software adoption and continued use by analysing the risks in their open-source dependencies is an important part of this. This paper explores, proposes and evaluates methods for assessing risk in using open-source software. There are a number of objects of analysis used to rate risk such as measuring project activity and vulnerability data and our aim is to combine them to give a more comprehensive view of the risks of using a particular open-source project. We examine the literature to decide on an appropriate algorithm for each measure of risk. We develop a tool to forecast the risks in dependency trees using ARIMA prediction by analysing project activity and vulnerabilities. A dependency graph is generated to aid in identifying where the risk lies in Maven dependency trees.**

***Index Terms*—ARIMA prediction, software vulnerabilities, Maven dependency trees.**

I. Introduction

It is estimated that between 90% and 93% of organisations and businesses make use of open-source software in their software systems [1]. The popularity of open-source software in large software projects continues to grow and over 3.6 million repositories depend on the top 50 open-source projects [2]. Open-source software has many benefits. The reuse of libraries can reduce development efforts substantially for developers. The use of open-source libraries is relatively unrestricted and transparent meaning developers can modify and distribute the code to meet their needs. The use of this software can come with risks as there is reliance on software of uncertain quality [1]. The reuse of libraries creates a dependency list called a supply chain. The supply chain is a suite of dependencies, which can include open-source software. The supply chain involves a network of participants who perform activities on the dependencies [3]. As software projects expand so too do their associated supply chains. Projects may thus become difficult to manage leading developers to turn to build tools like Maven for Java projects. In Java, in particular, the average application depends on approximately 40 third-party libraries [4], in comparison to C/C++ repositories which rely on 6.3 third-party libraries on average [5].

With the increased use of open-source software, attackers have begun to exploit known vulnerabilities in trusted components in the supply chain by injecting malicious code into software packages [6]. These types of attacks are increasing with 34% of all recorded supply chain attacks taking place in 2020 and 66% occurring in 2021 [7]. These attacks come at a great cost to users of open-source software with data breaches in particular costing $388 million on average for a breach of 50 million data records [8]. Once vulnerabilities are discovered, it is critical that users of open-source software identify whether they are impacted by the reported vulnerabilities or by inactivity in the open-source projects.

Risk assessment is a technique that can be used to measure whether certain open-source projects are of high-security standard when considering the quality of the software [9]. Several objects of analysis are used to predict this type of risk: the code base itself, project activity metadata, and historical vulnerabilities. Some of the key techniques in assessing risk in open-source software include the prediction of vulnerabilities based on code metrics, the prediction of project activity based on version control metadata from repositories such as GitHub, and the prediction of vulnerabilities based on data maintained in databases such as the National Vulnerability Database (NVD). For the purpose of this paper, we focused on the prediction of project activity and the number of vulnerabilities per month as these have previously been found to be significant [10, 11].

Most of the existing research focuses on one specific object of analysis such as project activity or vulnerability data. This paper explores the prediction of risks in Maven dependency trees by examining both GitHub project activity and NVD Common Vulnerabilities and Exposures (CVE) data. We aim to use time series forecasting, or more specifically, we aim to use ARIMA prediction for both sets of data. A colour-coded graph based on the analysed risk is produced capturing their risk requirements based on user-preference risk thresholds. Our software will help users identify any vulnerable components in their supply chain and more secure alternatives [12].

This paper is structured as follows: We first explore the literature on risk prediction in open-source software focusing in particular on both project activity predictions and vulnerability predictions. We then present a selection of case studies where a tool to explore risk in dependency trees would have been useful for users. We describe the data from

APIs that we used as well as the different predictions we generate. Finally, we describe the calculations and algorithms we used when predicting risk according to user-assigned metrics and the results that we obtained.

We will answer the following research question:

- *RQ:* Can we provide developers with an effective risk measure when choosing between multiple candidate open-source components?

By answering the above question we aim to provide some new insights in the area of risk prediction for Maven projects.

II. Key Terminology

In this section, we will discuss some of the terminology that will be used throughout this paper.

A. Maven

Maven is an open-source build automation and project management tool used in Java applications. It can be used to manage project dependencies using a Project Object Model, also known as a pom.xml file [13].

B. Time series analysis

Predicting the data for the future based on past data. The observations are taken at equal intervals - in our case they are taken monthly. A fitted model is built using the past data and future data can be forecast based on this model.

C. ARIMA

Autoregressive integrated moving average (ARIMA) is a statistical model used to predict future data in a time series. There are three parameters - p, d, and q. The AR stands for autoregression which is associated with the p parameter - the number of lagged values. The I stands for integrated as it observes the difference taken until the original series becomes stationary - this is represented with the d parameter. The MA stands for the moving average which is the dependency between an observed value and residual error from the model applied to previous observations - this is represented in the q parameter which is the size of the moving average window. The optimal p and q parameters can be determined by examining the Partial Autocorrelation Function and the Autocorrelation Function. The d parameter can be determined by differencing the data until the p-value of the Augmented Dickey-Fuller test is below a certain threshold [14].

D. AutoARIMA

Auto ARIMA is a library which cycles through and tests a number of parameter combinations to determine the best model for the provided data. The model is optimised based on the Akaike Information Criterion (AIC) and the Bayesian Information Criterion (BIC) with a focus on minimising these values [15].

III. Related Work

In this section, we review relevant work in the prediction of risk in open-source software. We examine the machine learning models used as well as the different objects of analysis used in the predictions.

A. Vulnerability Propagation

Roughly 1.2% of projects directly use vulnerable code in their dependencies [4]. However, a small number of vulnerable packages such as *jackson-databind* and *netty-codec-http* could affect up to 375,607 packages in the Maven ecosystem. This means CVEs can affect a large number of Maven projects. In [16] the authors present a dependency vulnerability graph of the entire NPM ecosystem. In investigating the JavaScript NPM ecosystem the authors propose an algorithm to resolve dependency trees as well as vulnerability propagation paths. They find that vulnerabilities affect up to 16.7% of third-party libraries in their current versions. Some widely known CVEs exist in the current dependency trees of a significant number of packages. As packages grow larger and more dependencies are added the complexity of discovering vulnerable dependencies increases. The complexity can be reduced from an exponential to a polynomial level depending on the number of relationships through a search algorithm based on lazy strategy [17]. The authors also propose the use of optimal blocking analysis to complete vulnerability repair for a project with minimal blocking. Blocking analysis finds the cost of repairing an entire propagation path where each dependency depends on an affected module - each dependency must be updated after a repair is made.

B. Project Metadata Analysis

One of the main indicators of an open-source project's survival is project activity. In addition to project activity level, community size and time-to-fix bugs, the subscriber base was identified as a measure of the success of an open-source project [18]. Measuring project activity in open-source software provides a means of predicting the success and security of projects. In [19] the authors examine the contributors in open-source projects and use several machine learning and deep learning techniques to predict their activity. They attempt to differentiate between newcomers who leave a project after a short time and those who stay for a long time. Some of the most important features are the number of project followers, the programming language and the average number of commits per developer. A dataset from GitHub as well as a survey conducted among developers were used to identify factors affecting long-term contributors. Predicting the health of an open-source project is explored in [10]. Data from over one thousand GitHub projects was used to predict multiple health indicators using machine learning and the error rate was reduced using hyperparameter optimisation. Predictions included the number of contributors, the number of commits, and the number of pull requests and issues. These metrics are indicative of the project's overall activity level. The authors also conducted a survey which indicated that the number of contributors, commits, and pull requests are real-world concerns for developers. The commit activity of individuals can also be forecasted as active developers play a significant role in the survival of open-source projects. Fewer than 50% of abandoned projects find new contributors so contributor retention is a major indicator of a project's health. A simple probabilistic model can be used to predict the time until the next contribution for specific contributors in a project [20]. Committing is an important activity in open-source projects and the standard measurement is *number of commits per month*. ARIMA is the most popular prediction method in predicting commits per month [21].

C. Vulnerability CVE Data Analysis

Time series forecasting is a popular method for predicting vulnerabilities. In [22] the authors analyse the use of an autoregressive integrated moving average (ARIMA) model to predict vulnerabilities. Data on vulnerabilities from the NVD was analysed [23]. The National Vulnerability Database (NVD) is a database maintained by the National Institute for Technology which maintains details on known vulnerabilities. The authors aimed to predict the number of Android vulnerabilities per month. The authors compared ARIMA's performance with the long short-term memory model (LSTM), artificial neural network (ANN) and recurrent neural network deep learning techniques and found ARIMA gives better results. They found the LSTM to have comparable error rates with ARIMA and to have more successful results than the two deep learning techniques. Another paper discusses the use of ARIMA to predict vulnerabilities [24]. They use a linear and a non-linear approach comparing ARIMA and Artificial Neural Networks and Support Vector Machines (SVM) again using reported vulnerabilities from the NVD for three different operating systems. The results indicated that there were sharp fluctuations in the number of vulnerabilities, the non-linear approaches were a better predictor, however, the ANN did not have sufficient data to perform well. The authors of [25] used two methods of time series analysis to predict vulnerabilities for web browsers: ARIMA and exponential smoothing. The trend, level and seasonality of the vulnerabilities were considered to help in prediction. They note that level was the only significant parameter. Datasets from the NVD were also used in this study. The prediction of the number of CVEs published in the NVD per month can be done using a number of different machine-learning models but it must be noted that the best predictor may change over time as the data changes. In [26] they explore 11 different models to forecast the number of CVEs and they note that most models fall flat at the three-month mark. These predictions are done on the NVD and predict the vulnerabilities for every project per month rather than focusing on just one project or dependency.

D. Research Gap

A research gap stems from the fact the papers above each focus on a single specific object of analysis (project activity or NVD) and thus do not produce a measurement of the overall health of a dependency. A dependency with many vulnerabilities and significant project activity may be deemed less risky by users than a dependency with low project activity and few vulnerabilities. The use of vulnerability prediction models to predict across multiple projects is also a gap in the research and it could certainly be useful when predicting risk in dependency trees.

IV. Case Studies

In this section, we will identify a number of case studies where a tool like ours would have been valuable in rapidly identifying high-risk dependencies in a project.

A. Log4j

In November 2021 a vulnerability was discovered in the popular open-source logging library Apache Log4j. This software is widely used in Java projects and web portals. The vulnerability in this package meant that hackers could run malicious code remotely on a victim's machine [27]. The main aim of the log4j exploit is to obtain Lightweight Directory Access Protocol information and this will serve as a gateway to full control of the machine. The attack consists of several steps: information gathering, weaponisation, delivery, exploitation and installation [28]. Numerous projects were unaware that they were affected by the Log4j vulnerability as their dependencies included the Log4j library and a graph of risk dependency risks such as that generated by our software would have helped them discover that they were vulnerable. Once the Log4j vulnerability was reported a user could have used our algorithm to verify that some of their dependencies have vulnerabilities and taken steps to find an alternative library.

B. Heartbleed Vulnerability

In April 2014 the Heartbleed vulnerability was discovered in OpenSSL versions 1.0.1 and 1.0.1f. OpenSSL is a popular open-source cryptography library. The vulnerability relates to the established connection being kept "alive" through *Heartbeat* messages. Hackers could craft a similar message to the official *Heartbeat* Message and exploit a buffer overflow to obtain private information in response [29]. It was estimated that 24-55% of HTTPS servers in the Alexa Top 1 million were vulnerable and even 48 hours after the disclosure of the vulnerability around 11% remained vulnerable as they did not take action to mitigate the attack. This attack could reveal sensitive data such as private keys and only 10% of affected servers changed their TLS certificates and of these 14% did not replace their private keys thus gaining no protection [30]. A number of affected servers were unaware they were affected so a tool such as the one we propose would have been useful in ascertaining whether they were vulnerable to Heartbleed.

V. Methodology

This paper explores the creation of a tool to predict risk in dependency trees. The tool aims to combine two (rather than focusing on one) of the commonly used objects of analysis as explored above to give a more complete analysis.

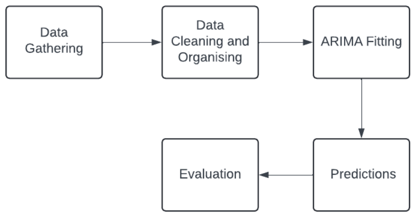

Fig. 1. The Time Series Pipeline.

TABLE I
LIST OF SOURCES USED

Data Source	Purpose	Type
GitHub projects	Find Dependencies	Maven-dependency trees
GitHub API	Project Meta-data	API
NVD API	Vulnerability Prediction	CVE API

In this section, we initially describe the data we used and subsequently the prediction methods as well as our approach to risk. For both predictions, seasonality was not considered as the papers reviewed above did not find it to be a significant parameter [25]. The time series pipeline used in this paper can be seen in Figure 1. We focus on two risk prediction methods here:

1) Project Activity Prediction
2) Vulnerability CVE Data Prediction

A. Data Gathering

The first set of test data used for this project was gathered from various Maven projects available on GitHub. Some small projects were gathered to test the feasibility of the graphing and prediction techniques. As explored in one of the case studies, a sample project that incorporated the Log4j logging library was tested.

The GitHub API was used for the project activity risk prediction and time-to-close issues and commits per month were determined as appropriate measures of risk for this project as they were identified as predictors of a project's health in the literature [10]. The project can be configured to predict either the issue data, the commit data or both. Each dependency was matched to the project's GitHub location using a mapping file and the GitHub API was used to gather the required GitHub issue or commit data.

The NVD API was used to gather data for the vulnerability risk prediction. For the purpose of this project, we did not discriminate between vulnerability severity levels as there was not much vulnerability data per month for each of the dependencies. The number of vulnerabilities reported per month was gathered for each of the dependencies using this API.

Data sources and their role in this project can be seen in Table I. The source of data, the purpose of the data, and the type of data that was used are displayed in the table.

Listing 1. Example GitHub extraction

```
Hamcrest Dependency in Maven Project:
org.hamcrest:hamcrest-core:jar:1.3:test

Keyword for Hamcrest Dependency:
hamcrest-core
```

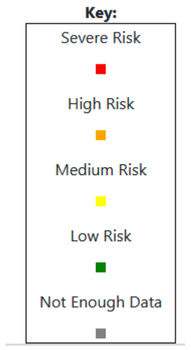

Fig. 2. The key for the risk dependency graph.

B. Dependencies Graphing

The dependency graph was constructed by analysing the Maven dependencies in the POM files. The tool then extracts a keyword to locate the GitHub URL for each of the dependencies. An example of this can be seen in Listing 1. The keyword is then matched with its associated GitHub URL from our data file. The project activity was then predicted and vulnerability data was collected using the extracted libraries and the dependency keywords. After the risk scores have been returned each of the dependencies is assigned a node in the graph according to its depth and the colour is assigned based on the risk score.

A NetworkX graph was constructed with each of the dependencies displayed as nodes on the graph and their risk scores were reflected as colours. The original project is displayed as a black square node. There is a key defining the risk levels of each of the nodes for helpful decoding of the graphs as displayed in Figure 2. The edges represent the dependency links between nodes and the depth (distance) of each of the nodes from the original project captures how dependencies rely on other dependencies. The graph shows users how they are relying on dependencies that they did not directly add to their project and how risk can be involved in importing different dependencies.

C. Project Activity Prediction

For the project activity prediction, we used data fetched from the GitHub API [31]. The focus was on the number of commits per month and time-to-fix issues as these were identified as indicators of a project's activity and thus its success [18, 21]. The user has the option to specify whether they wish to include the issue, commit data or both. All data available was used instead of specifying a specific time range as all of the projects had different time ranges available to use.

All commits/closed issues available were gathered using the GitHub API and were processed to ensure there was data for each month up until the current date. In the case of commits, this was simply a count of commits per month and we added a zero for each of the months with no data. For the issue data, the time-to-close in days was gathered for each issue and a monthly average was calculated, adding a for each of the months that had no data.

For the prediction, the date was used as an index and after calculating the *d* parameter by differencing the data until the p-value of the *Augmented Dickey–Fuller* test (ADF) test was below 0.05. To identify the p and q parameters the *Partial Autocorrelation Function (PACF)* and *Autocorrelation function (ACF)* graphs were examined but it was not feasible to implement this system of parameter selection at a large scale. The *Auto ARIMA* library was used to make the predictions as it offered a suitable method of selecting parameters when predicting across many projects [15]. The predicted value for the next month was returned for each of the GitHub libraries. If the data is constant or there is no data a -1 was returned as there is not enough data to make a prediction.

Listing 2. Example Keywords

```
JUnit4 Dependency in Maven Project:
junit:junit:jar:4.11:test

Array of Keywords in JUnit Dependency:
['junit', 'junit4', 'junit jar ']
```

D. Vulnerability Risk Prediction

For the vulnerability CVE data prediction, data extracted using the NVD API was used [23]. Dependencies were then split into an array of keywords to ensure that all of the relevant CVEs for that dependency were gathered. An example of this is in Listing 2. For each of the keywords, a request is made and any CVEs not already in the array of gathered CVEs are added. The monthly data, similar to the commits above, was gathered and the number of vulnerabilities per month was counted and a zero was inserted for missing values.

The prediction method used is similar to that used to model project activity. Some dependencies have few vulnerabilities per month or even per year. The *Auto ARIMA* library was used to make the predictions.

E. Risk Calculations

In order to calculate risk, configuration options were defined for a user to decide how many commits, days-to-close issues, and vulnerabilities per month are acceptable. Further configuration options can be added as required.

$$projectActivityScore = (x/numDaysToFixIssues) * 10$$

$$projectActivityScore = (numCommits/x) * 10$$

TABLE II
RISK SCORE LEVELS

Score	Risk Level
<0	Not Enough Data
0 - 2.5	Low Risk
2.5-5	Medium Risk
5-7.5	High Risk
>7.5	Severe Risk

$$vulnerabilityScore = (x/vulsCountPerMonth) * 10$$

$$overallScore = (projectActivityScore + vulnerabilityScore)/2$$

Above are the formulae for risk calculations. The aim was to calculate a score as defined above for project activity and a score for vulnerabilities and combine them. There is the option to calculate project activity risk based on issue fix time, commits per month or both. The default levels are set to reasonable levels which can be changed. The user sets the configuration options of *numDaysToFixIssues*, *numCommits*, and *vulCountsPerMonth* to levels they deem acceptable according to their needs. The idea is that they are then assigned a risk score: any number less than 0 indicates there is not enough data, 0-2.5 is low risk, 2.5-5 is medium risk, 5-7.5 is high risk and anything above 7.5 is a severe risk. These can be seen in table II. The overall score is then displayed in the graph - colour-coded using the key from Figure 2. The risk calculation criteria are left to the user to decide as organisations/developers have different priorities regarding an acceptable number of commits/time to close issues/vulnerabilities per month. An example of the risk score being calculated is a predicted value of 67 commits per month and 1.2 vulnerabilities per month where the acceptable levels are 70 commits per month and 2 vulnerabilities per month the risk score is: 8.22. This means the project is of severe risk level according to the set values.

VI. RESULTS

In this section, the results obtained from our tool as well as some examples of the resulting graphs are discussed. One particular project [32] affected by the Log4j vulnerability will be the focus, although a number of different types of projects were also tested to establish a baseline for the functionality of the graphing. By testing different types of projects, different dependencies were tested for risk. First, the risk dependency visual that was created is examined. Subsequently, two graphs of the predictions made are examined - one for project activity and one for vulnerability predictions. Risk graphs for different dependencies are explored.

Figure 3 shows a final dependency graph with colour-coded risk evaluations for each of the dependencies. This figure

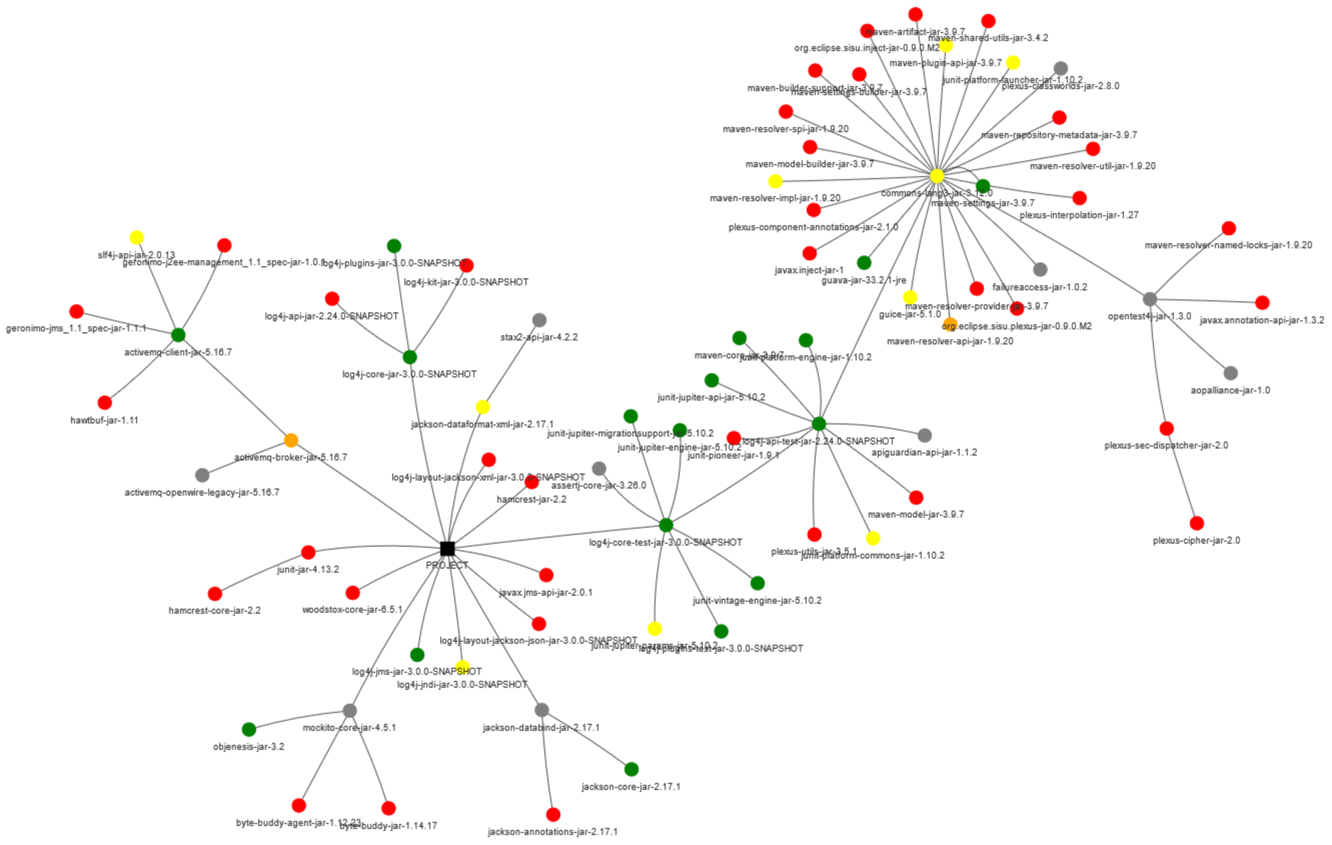
maven-artifact-jar-3.9.7
maven-shared-utils-jar-3.4.2
org.eclipse.sisu.inject-jar-0.9.0.M2
maven-plugin-api-jar-3.9.7
junit-platform-launcher-jar-1.10.2
plexus-classworlds-jar-2.8.0
maven-builder-support-jar-3.9.7
maven-settings-builder-jar-3.9.7
maven-resolver-spi-jar-1.9.20
maven-repository-metadata-jar-3.9.7
maven-model-builder-jar-3.9.7
maven-resolver-util-jar-1.9.20
maven-resolver-impl-jar-1.9.20
commons-lang3-jar-3.12.0
maven-settings-jar-3.9.7
plexus-interpolation-jar-1.27
plexus-component-annotations-jar-2.1.0
maven-resolver-named-locks-jar-1.9.20
javax.inject-jar-1
guava-jar-33.2.1-jre
failureaccess-jar-1.0.2
maven-resolver-provider-jar-3.9.7
guice-jar-5.1.0
org.eclipse.sisu.plexus-jar-0.9.0.M2
opentest4j-jar-1.3.0
javax.annotation-api-jar-1.3.2
maven-resolver-api-jar-1.9.20
aopalliance-jar-1.0
maven-core-jar-3.9.7
junit-platform-engine-jar-1.10.2
junit-jupiter-api-jar-5.10.2
junit-jupiter-migrationsupport-jar-5.10.2
junit-jupiter-engine-jar-5.10.2
log4j-api-test-jar-2.24.0-SNAPSHOT
apiguardian-api-jar-1.1.2
junit-pioneer-jar-1.9.1
plexus-sec-dispatcher-jar-2.0
assertj-core-jar-3.26.0
maven-model-jar-3.9.7
plexus-cipher-jar-2.0
plexus-utils-jar-3.5.1
junit-platform-commons-jar-1.10.2
log4j-core-test-jar-3.0.0-SNAPSHOT
junit-vintage-engine-jar-5.10.2
junit-jupiter-params-jar-5.10.2
log4j-plugins-test-jar-3.0.0-SNAPSHOT
slf4j-api-jar-2.0.13
geronimo-j2ee-management_1.1_spec-jar-1.0.1
log4j-plugins-jar-3.0.0-SNAPSHOT
log4j-kit-jar-3.0.0-SNAPSHOT
log4j-api-jar-2.24.0-SNAPSHOT
geronimo-jms_1.1_spec-jar-1.1.1
stax2-api-jar-4.2.2
activemq-client-jar-5.16.7
log4j-core-jar-3.0.0-SNAPSHOT
hawtbuf-jar-1.11
jackson-dataformat-xml-jar-2.17.1
activemq-broker-jar-5.16.7
activemq-openwire-legacy-jar-5.16.7
log4j-layout-jackson-xml-jar-3.0.0-SNAPSHOT
hamcrest-jar-2.2
PROJECT
junit-jar-4.13.2
hamcrest-core-jar-2.2
woodstox-core-jar-6.5.1
javax.jms-api-jar-2.0.1
log4j-layout-jackson-json-jar-3.0.0-SNAPSHOT
log4j-jms-jar-3.0.0-SNAPSHOT
log4j-jndi-jar-3.0.0-SNAPSHOT
objenesis-jar-3.2
mockito-core-jar-4.5.1
jackson-databind-jar-2.17.1
jackson-core-jar-2.17.1
byte-buddy-agent-jar-1.14.17
byte-buddy-jar-1.14.17
jackson-annotations-jar-2.17.1

Fig. 3. Dependency tree results with risk levels colour coded.

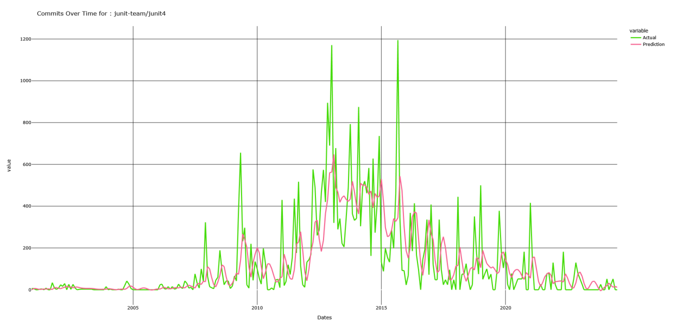

Fig. 4. Commits Per Month for JUnit4: Actual vs Fitted Values.

was based on a user configuration of two vulnerabilities per month being acceptable with a minimum of 70 commits per month as acceptable. It is clear there are a number of lower-level dependencies which are risky which the user may not be expecting. The graph is interactive so the colour of each specific dependency can be investigated.

Figure 4 shows the commits per month, fitted vs actual values, for the JUnit4 package. It is clear from this graph that the project activity in terms of commits was at its highest from around 2012 to 2017. The model is used to predict a value for the next month. For the month of August 2024, the number of commits is predicted to be 10.9951.

Figure 5 shows the Log4j vulnerabilities over time and it is clear the Log4j vulnerabilities were reported in 2021 and these are predicted reasonably well in the subsequent months. For August 2024, there are 0.07898 vulnerabilities predicted for Log4j, for example.

The tool displays graphs for every fitted model. If there is not enough data for the prediction to fit a model then there is no corresponding time series graph. The user can then investigate how accurate the fitted values are for each of the models. For time series graphs with less data over time, the models did not perform well and occasionally there were constant lines being returned rather than a fitted model which could not be returned as a value for the risk dependency graph. An example of this can be seen in Figure 6. The actual and fitted values for the vulnerability data for the Jackson databind dependency [33] is shown in this figure. The fitted values are a constant value of just below 1 which is not accurate given the actual values.

VII. Discussion

In this section, we will discuss our progress in answering the research question outlined in the introduction. Can we provide developers with an effective risk measure when choosing between multiple candidate open-source components?

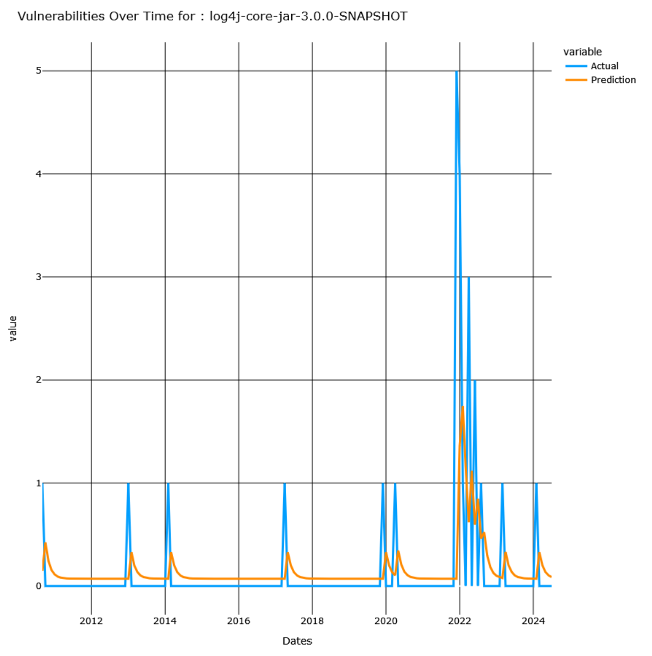

Fig. 5. Vulnerabilities Per Month for Log4j: Actual vs Fitted Values.

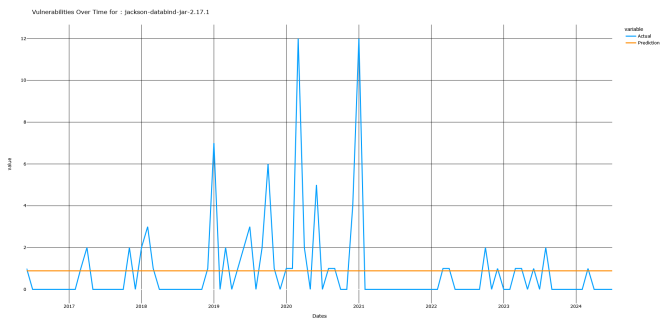

Fig. 6. Vulnerabilities Per Month for Jackson databind: Actual vs Fitted Values.

Our aim was to investigate whether features could be combined from the popular risk prediction methods to provide developers with a more complete risk measurement. The combination of risk prediction methods aided in identifying high-risk modules in dependency trees. Many of the dependencies investigated did not have any vulnerabilities reported in the NVD which is a desirable trait in any dependency. There were also, however, some dependencies no longer under development which means if a vulnerability were discovered in a non-surviving project it was unlikely to be fixed. This may not be an issue for large organisations that incorporate open-source software and configure it to meet their needs but smaller organisations, lacking the resources to rework open-source software, may prefer to incorporate software that is still actively under development. We generate a visual risk graph of the dependency tree with the calculated risk scores for each of the predictions. This graph gives an overall risk analysis for the Maven project's dependency tree and it is feasible to create a similar graph for a larger industry graph due to the zooming functionality. We believe our tool and our investigation conclude that the combination of risk prediction methods can provide developers with an effective risk measure when choosing between candidate open-source components.

VIII. Performance Evaluation

To evaluate the algorithm, a number of Maven projects were selected with varying dependencies. A sample Log4j maven project was used.

The project activity predictions generally performed significantly better than the vulnerability CVE data - due to a lack of data in the NVD databases (which can in some cases mean no vulnerabilities - a positive). Numerous dependencies were also grouped in the same projects on GitHub which meant that there was only the need to calculate their associated project activity risk once. One minor issue encountered in testing was that artifacts are not currently linked to their hosted GitHub repository. A mapping file was generated to solve this problem allowing us access to the appropriate GitHub repository for each dependency artifact to obtain the data for analysis. For the evaluation of the models in this project, two commonly used metrics, mean absolute error (MAE), and root mean squared error (RMSE) were applied.

$$MAE = (1/n)\sum i = n|actual - forecast|$$

$$RMSE = \sqrt((\sum(actual - forecast)^2)/samplesize)$$

These are the most commonly used evaluation metrics when evaluating Time Series predictions such as ARIMA. MAE is simply the mean of absolute errors where the absolute error is the difference between actual and predicted values. The error in MAE scales linearly so an error of ten is ten times worse than an error of one. RMSE measures the standard deviation of the residuals and their dispersion. It is the commonly used metric to evaluate the difference between actual and forecasted values. The RMSE measurement can be heavily influenced by any outlier predictions [34] as the errors are squared. The squared errors mean that an error of ten is ten times worse than an error of one. We investigated the use of mean absolute percentage error (MAPE) but we found this was not suitable as some of the actual values were 0 - causing division by zero errors. MAPE is a measure of the average of the percentage error and will produce extreme values if the actual value is quite small and in the case of the actual value being a 0 the result is always Infinite. Through the use of these evaluation metrics, an understanding can be gained of the performance of our time series models.

The results we obtained for the two figures, fig 4 and fig 5, can be seen in Table III. The results for the commit prediction model can be compared with the standard deviation of the actual values which is 152.79. This shows that the MAE and RMSE results are within a reasonable error margin. The

TABLE III
EVALUATION OF PREDICTIONS

Prediction	MAE	RMSE
Commits per month	123.18	213.96
Vulnerabilities per month	0.145	0.476

RMSE result is larger than the MAE result as there are large fluctuations in the commits per month data but the fitted model performs well. The results for the vulnerability predictions can be compared with the standard deviation of the actual values which is 0.432. Both the MAE and the RMSE values for the vulnerability prediction model are within a reasonable error margin when compared to the standard deviation. Considering the Log4j vulnerability caused a large spike in vulnerabilities being reported for this package, the vulnerability prediction performs quite well and takes these spikes into account when predicting the next values. Overall both predictions performed well and thus the returned predicted value for the next month can be considered a reasonably accurate predicted value.

A. Threats to Validity

In this section, we identify possible threats to the validity of the results we obtained and how they were addressed.

1) Dataset: The dataset used was quite limited. Several small Maven projects were used to test the initial design. This was to ensure that the analysis was tractable and test the code rapidly and efficiently generate the graphs. The analysis takes significant time to run for larger projects - although the data gathering takes up the largest proportion of time. To combat the lack of available datasets and the manual work gathering GitHub URLs a sample Log4j Maven project was analysed as this had known vulnerabilities and thus had a spike in fixes and project activity. Another dataset issue was that there was a scarcity of vulnerability data per month in the same way that there was for the project activity. As discussed above, the resulting vulnerability predictions were not always the most accurate.

2) AutoARIMA: The use of manual ARIMA prediction was investigated but this meant PACF and ACF plots for each of the predictions would need to be investigated which was unfeasible for some of the larger dependency trees. It was found to be reasonably simple to calculate the differencing parameter calculated but very difficult to calculate the others. The use of the AutoARIMA predictions means the predictions are not as accurate as they could be for every dataset but as this project requires significant API usage and data processing for each of the dependencies (which can be a large number) it was preferable to implement an automatic system rather than manually analysing each dataset. The prediction made assumes that there is no seasonality in the datasets as this was reported in the literature to be of little significance [25].

IX. CONCLUSIONS & FURTHER WORK

The growing reliance on open-source software [35] has introduced several challenges regarding the security and integrity of software supply chains. Tools to explore the risk in software supply chains like the one we proposed, designed and implemented as described in this paper have proven to be useful in identifying high-risk open-source dependencies. As the reliance on open-source software grows, so does the need for tools such as ours. We set out to answer one research question and our investigation concluded that the combination of the two measures of risk, both project activity and vulnerabilities over time gives a valuable overview of the risk in using a project. We have depicted risk visually in an interactive dependency graph.

For the purpose of this paper, we focused on examining Maven dependency trees which are primarily used for Java projects. Further work to examine other package manager dependency trees such as *PyPI* for Python, *NPM* for JavaScript, and *Conan* for C is clearly warranted. This may give a different perspective as different languages may be more prone to vulnerabilities or have projects which are less likely to survive. Java is known for having many third-party dependencies with vulnerabilities but it is also well known that C is more prone to vulnerabilities in source code so it may be useful to explore this question across multiple languages.

Another avenue of future work would be to extend the supported source code repository beyond GitHub and support others such as Jira or GitLab via an extended API connector for each type of repository. This would allow more extensive data to be gathered and used for evaluating projects. An automatic tool to look up matching artifacts to repositories like this would be useful in other contexts such as finding other details about project activity. This would make the project activity prediction for this project easier and increase its scalability when analysing larger projects.

In general, the performance of our tool was reasonably accurate in terms of the prediction of both vulnerabilities per month and project activity. We compromised on the accuracy of our predictions in order to ensure that we could obtain reasonably accurate predictions for many dependencies without having to analyse each individually. As a result, the tool can scale up when making predictions for any type of project. We kept the tool configurable so users could set the level of risk that they deem appropriate. There remains scope to investigate other project activity metrics such as the number of remaining contributors which was an important factor in the literature for the survival of an open-source project [19]. The vulnerability aspect of the project can also be configured so that users can choose their own risk levels to suit their context. The aim of this project was to highlight to users high-risk modules in their Maven dependency trees so that they can opt for more secure modules when comparing open-source software. We have demonstrated that a tool like ours could aid developers in choosing modules which are less risky and in discovering high-risk unknown dependencies. We have shown our approach provides a solid foundation for future research in the area of securing the software supply chain.

References

[1] S. Zajdel, D. E. Costa, and H. Mili, "Open source software: an approach to controlling usage and risk in application ecosystems," in *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A*, ser. SPLC '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 154–163, event-place: Graz, Austria. [Online]. Available: https://doi.org/10.1145/3546932.3547000

[2] V. N. Subramanian, "An empirical study of the first contributions of developers to open source projects on GitHub," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 116–118, event-place: Seoul, South Korea. [Online]. Available: https://doi.org/10.1145/3377812.3382165

[3] K. Singi, J. C. B. R P, S. Podder, and A. P. Burden, "Trusted Software Supply Chain," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2019, pp. 1212–1213, journal Abbreviation: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).

[4] A. M. Mir, M. Keshani, and S. Proksch, "On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2023, pp. 201–211, journal Abbreviation: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).

[5] W. Tang, Z. Xu, C. Liu, J. Wu, S. Yang, Y. Li, P. Luo, and Y. Liu, "Towards Understanding Third-party Library Dependency in C/C++ Ecosystem," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023, event-place: Rochester, MI, USA. [Online]. Available: https://doi.org/10.1145/3551349.3560432

[6] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," *CoRR*, vol. abs/2005.09535, 2020, arXiv: 2005.09535. [Online]. Available: https://arxiv.org/abs/2005.09535

[7] M. Z. Malik and S. Z. A. Bukhari, "Protection Mechanism against Software Supply Chain Attacks through Blockchain," in *2023 International Conference on Communication Technologies (ComTech)*, Mar. 2023, pp. 73–78, journal Abbreviation: 2023 International Conference on Communication Technologies (ComTech).

[8] X. Wang, "On the Feasibility of Detecting Software Supply Chain Attacks," in *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)*, Dec. 2021, pp. 458–463, journal Abbreviation: MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM).

[9] I. Abunadi and M. Alenezi, "Towards Cross Project Vulnerability Prediction in Open Source Web Applications," in *Proceedings of the The International Conference on Engineering & MIS 2015*, ser. ICEMIS '15. New York, NY, USA: Association for Computing Machinery, 2015, event-place: Istanbul, Turkey. [Online]. Available: https://doi.org/10.1145/2832987.2833051

[10] T. Xia, W. Fu, R. Shu, R. Agrawal, and T. Menzies, "Predicting health indicators for open source projects (using hyperparameter optimization)," *Empirical Softw. Engg.*, vol. 27, no. 6, Nov. 2022, place: USA Publisher: Kluwer Academic Publishers. [Online]. Available: https://doi.org/10.1007/s10664-022-10171-0

[11] S. Wu, C. Wang, J. Zeng, and C. Wu, "Vulnerability Time Series Prediction Based on Multivariable LSTM," in *2020 IEEE 14th International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, Nov. 2020, pp. 185–190, journal Abbreviation: 2020 IEEE 14th International Conference on Anti-counterfeiting, Security, and Identification (ASID).

[12] "Open Source Software Security | CISA." [Online]. Available: https://www.cisa.gov/opensource

[13] "Maven – Introduction." [Online]. Available: https://maven.apache.org/what-is-maven.html

[14] A. A. Ariyo, A. O. Adewumi, and C. K. Ayo, "Stock Price Prediction Using the ARIMA Model," in *2014 UKSim-AMSS 16th International Conference on Computer Modelling and Simulation*, Mar. 2014, pp. 106–112, journal Abbreviation: 2014 UKSim-AMSS 16th International Conference on Computer Modelling and Simulation.

[15] "pmdarima: Python's forecast::auto.arima equivalent." [Online]. Available: http://alkaline-ml.com/pmdarima

[16] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, May 2022, pp. 672–684, journal Abbreviation: 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE).

[17] W. Hu, Y. Wang, X. Liu, J. Sun, Q. Gao, and Y. Huang, "Open Source Software Vulnerability Propagation Analysis Algorithm based on Knowledge Graph," in *2019 IEEE International Conference on Smart Cloud (SmartCloud)*, Dec. 2019, pp. 121–127, journal Abbreviation: 2019 IEEE International Conference on Smart Cloud (SmartCloud).

[18] R. Sen, S. S. Singh, and S. Borle, "Open source software success: Measures and analysis," *Decision Support Systems*, vol. 52, no. 2, pp. 364–372, Jan. 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016792361100159X

[19] L. Bao, X. Xia, D. Lo, and G. C. Murphy, "A Large Scale Study of Long-Time Contributor Prediction for GitHub Projects," *IEEE Transactions on Software Engineering*,

vol. 47, no. 6, pp. 1277–1298, Jun. 2021.

[20] A. Decan, E. Constantinou, T. Mens, and H. Rocha, "GAP: Forecasting Commit Activity in git Projects," *Journal of Systems and Software*, vol. 165, p. 110573, Mar. 2020.

[21] K. Chahal and M. Saini, "Fuzzy Analysis and Prediction of Commit Activity in Open Source Software Projects," *IET Software*, vol. 10, Jun. 2016.

[22] K. Gencer and B. Fatih, "Time Series Forecast Modeling of Vulnerabilities in the Android Operating System Using ARIMA and Deep Learning Methods," *Sustainable Computing: Informatics and Systems*, vol. 30, p. 100515, Jan. 2021.

[23] "Vulnerability APIs." [Online]. Available: https://nvd.nist.gov/developers/vulnerabilities

[24] N. R. Pokhrel, H. Rodrigo, and C. Tsokos, "Cybersecurity: Time Series Predictive Modeling of Vulnerabilities of Desktop Operating System Using Linear and Non-Linear Approach," *Journal of Information Security*, vol. 08, pp. 362–382, Jan. 2017.

[25] Y. Roumani, J. K. Nwankpa, and Y. F. Roumani, "Time series modeling of vulnerabilities," *Computers & Security*, vol. 51, pp. 32–40, Jun. 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404815000358

[26] E. Leverett, M. Rhode, and A. Wedgbury, "Vulnerability Forecasting: Theory and Practice," *Digital Threats: Research and Practice*, vol. 3, Nov. 2021.

[27] H. Gupta, A. Chaudhary, and A. Kumar, "Identification and Analysis of Log4j Vulnerability," in *2022 11th International Conference on System Modeling & Advancement in Research Trends (SMART)*, Dec. 2022, pp. 1580–1583, journal Abbreviation: 2022 11th International Conference on System Modeling & Advancement in Research Trends (SMART).

[28] F. Maulana, H. Fajri, M. F. Safitra, and M. Lubis, "Unmasking log4j's Vulnerability: Protecting Systems against Exploitation through Ethical Hacking and Cyberlaw Perspectives," in *2023 9th International Conference on Computer and Communication Engineering (ICCCE)*, Aug. 2023, pp. 311–316, journal Abbreviation: 2023 9th International Conference on Computer and Communication Engineering (ICCCE).

[29] S. Kyatam, A. Alhayajneh, and T. Hayajneh, "Heartbleed attacks implementation and vulnerability," in *2017 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, May 2017, pp. 1–6, journal Abbreviation: 2017 IEEE Long Island Systems, Applications and Technology Conference (LISAT).

[30] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, "The Matter of Heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 475–488, event-place: Vancouver, BC, Canada. [Online]. Available: https://doi.org/10.1145/2663716.2663755

[31] "GitHub REST API documentation." [Online]. Available: https://docs.github.com/_next/data/mjcX-Kg16OuchVhOk_10Y/en/free-pro-team@latest/rest.json?apiVersion=2022-11-28&versionId=free-pro-team%40latest&productId=rest

[32] "logging-log4j-samples/log4j-server at main · apache/logging-log4j-samples." [Online]. Available: https://github.com/apache/logging-log4j-samples/tree/main/log4j-server

[33] "FasterXML/jackson-databind," Jul. 2024, original-date: 2011-12-23T07:17:41Z. [Online]. Available: https://github.com/FasterXML/jackson-databind

[34] F. Abdulhafidh Dael, U. Yavuz, and A. A. Almohammedi, "Performance Evaluation of Time Series Forecasting Methods in The Stock Market: A Comparative Study," in *2022 International Conference on Decision Aid Sciences and Applications (DASA)*, Mar. 2022, pp. 1510–1514, journal Abbreviation: 2022 International Conference on Decision Aid Sciences and Applications (DASA).

[35] "CISA Open Source Software Security Roadmap | CISA," Sep. 2023. [Online]. Available: https://www.cisa.gov/resources-tools/resources/cisa-open-source-software-security-roadmap

Appendix A
Configuration Options JSON

This listing shows an example of how the configuration options can be set.

Listing 3. Configuration Options

```
{
  "num_vuls": "2",
  "num_days_to_fix": "46",
  "num_commits": "70",
  "issues_or_commits": "commits",
  "token": "",
  "nvd_key": ""
}
```

Appendix B
Dependencies Comparison

This figure 7 shows the tool being used to compare two similar dependencies and it is clear to see that Mockito is the safer option when compared to JUnit4 according to the parameters set.

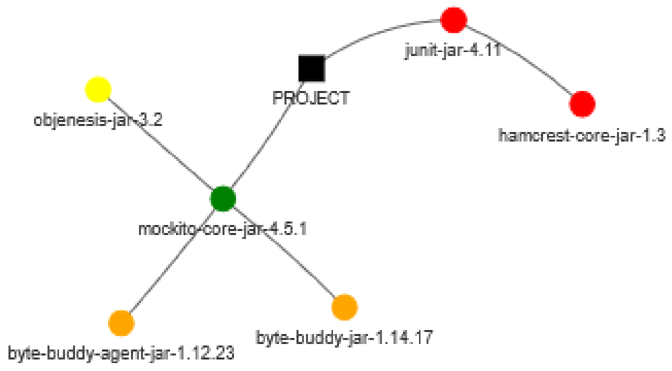

Fig. 7. Comparison of Unit Testing Dependencies: Mockito vs JUnit4.

Appendix C
Dependency Tree Example

The following listing shows a Maven dependency example used to test this project.

Listing 4. Example Maven dependencies

```
[INFO] Scanning for projects...
[INFO]
[INFO] --------------------------< org.youtube:form >--------------------------
[INFO] Building form 1.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- dependency:3.6.0:tree (default-cli) @ form ---
[INFO] org.youtube:form:jar:1.0-SNAPSHOT
[INFO] +- org.openjfx:javafx-controls:jar:16:compile
[INFO] |  +- org.openjfx:javafx-controls:jar:win:16:compile
[INFO] |  \- org.openjfx:javafx-graphics:jar:16:compile
[INFO] |     +- org.openjfx:javafx-graphics:jar:win:16:compile
[INFO] |     \- org.openjfx:javafx-base:jar:16:compile
[INFO] |        \- org.openjfx:javafx-base:jar:win:16:compile
[INFO] +- org.openjfx:javafx-fxml:jar:16:compile
```

```
[INFO] |  \- org.openjfx:javafx-fxml:jar:win:16:compile
[INFO] +- org.eclipse.persistence:eclipselink:jar:2.7.7:compile
[INFO] |  +- org.eclipse.persistence:jakarta.persistence:jar:2.2.3:compile
[INFO] |  \- org.eclipse.persistence:commonj.sdo:jar:2.1.1:compile
[INFO] +- org.eclipse.persistence:javax.persistence:jar:2.0.0:compile
[INFO] \- org.postgresql:postgresql:jar:9.3-1102-jdbc41:compile
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.360 s
[INFO] Finished at: 2024-05-19T20:49:23+01:00
[INFO] ------------------------------------------------------------------------
```