Using a Time Dependency Graph to find the most widely used Debian package

Teodor Dobrev
Supervisor(s): Georgios Gousios, Diomidis Spinellis
EEMCS, Delft University of Technology, The Netherlands

23-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

The main principle of Open Source development is that developers can reuse different libraries over and over again to make their lives easier. That is why this practice has gained a lot of popularity. However, libraries usually depend on other already existing pieces of code. This means that whenever some small piece of code fails, the whole application may crash due to its dependencies. Since Debian is considered as one of the largest community run distributions, it is important to have a good tool to analyse these dependencies to help in avoiding such crashes. That is why this research will focus on building a dependency graph for Debian's package manager and finding out which are the most widely used packages. What separates this research from previous related works, is the addition of a time component to the graph. This will be in the form of a timestamp of the release date for each version of a package. This allows for extensive traversal of the graph, which can also be based on time periods. By doing so, no transitive dependencies should be missed. The paper concludes with defining *most widely used* as the packages which achieve the highest PageRank score when put into the graph. The top 3 ones are ***"libc6"***, ***"libgcc1"*** and ***"multiarch-support"***.

1 Introduction

Often times engineers reuse already existing pieces of code to save time and to avoid doing work that is redundant [17]. However, the dependencies that these packages usually carry with them are often neglected. This can lead to problems since those libraries and packages themselves depend on other third-party software [1]. Those transitive dependencies may lead to failures that are very hard to detect and can go unnoticed for a long time [8]. Furthermore, Soto-Valero *et al.* have shown in reference [18] that these transitive dependencies make up for a huge part of the so called "bloated" dependencies which are not actually needed during the building or the running of the application. Prana *et al.* take a look at the effects that vulnerable dependencies have on Open-Source projects in reference [12]. They highlight that popularity of the project, activity or developer experience do not have an impact on the handling of the dependency vulnerabilities. This is already a point of concern since it shows the need of further analysis and the development of tools to limit those problems.

While according to Prana *et al.* in reference [12] the *"Denial of Service"* and *"Information Disclosure"* were the most commonly encountered problems, there are many examples of more severe vulnerabilities. One recent example of a vulnerability caused by overlooking the dependencies of the applications is the log4j vulnerability [16]. There, a vulnerability in a logging library which is very widely used, allowed malicious attackers to execute code remotely on the computers that used it. Some other famous examples are the "left-pad" incident [24] and the "SolarWinds" hack [3]. What is more, "Using vulnerable or outdated components" was ranked sixth on The Open Web Application Security Project (OWASP) top ten list of vulnerabilities for 2021[1].

As Kikas *et al.* mention in reference [7]: "dependency management practices have received little attention despite being a crucial part of almost all software projects". That is why this research is focused on creating an extensive package dependency graph to continue the work in the field. Current analysis often overlooks the evolution of the transitive dependencies over time. The main topic of this paper is introducing this time component by adding the timestamps of release dates to each version and following the transitive dependencies for the current period. This work will focus on applying this graph to the Debian Package Manager since it is considered as one of the largest community run distributions. The end goal is to see how does the introduction of time in the graph affect the Package Dependency Network and to find the most widely used package.

Therefore, to achieve this it is first needed to reformat the data about the existing package dependencies so that the release date of the package version can be taken into account. The reason for this is, that usually packages depend on the latest release version of the dependency. Looking at the dependency graph in Figure 1 provides a more intuitive description to the problem.

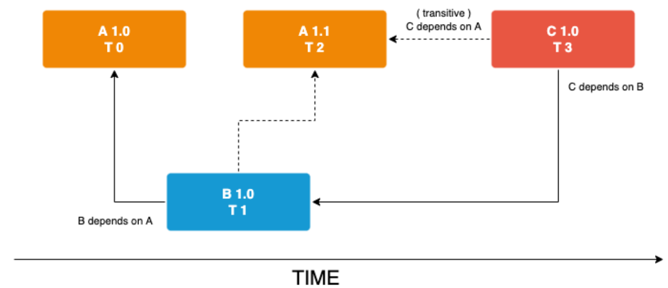

Figure 1: Representation of the dependencies of projects A,B and C in time

- Package A with version 1.0 is released at time T0.
- Package B with Version 1.0 is released at time T1 and it is depending on the latest version of Package A (currently A.1.0).
- Package A releases version 1.1 at time T3 > T2.
- Package C with version 1.0 is released at time T4 > T3 and it is depending on the latest version of Package B(currently B.1.0).

This particular scenario, shows that Package C should transitively depend on Package A's latest version. However, the latest version of A known to B which is the direct dependency

[1]https://owasp.org/Top10/

of C, is actually not the latest one. The main point of this research is to take this time effect into account and see how it will influence the already existing dependency graphs. This paper will focus on applying a time dependency graph structure for the Debian package manager and finding out ***"What are the most widely used Debian packages?"***. To answer this question, it would have to be broken down into three subquestions:

- **Research Question 1**: How to design efficiently a dependency graph data structure?
- **Research Question 2**: How does time influence a dependency graph data structure?
- **Research Question 3**: What measures of criticality could be used to evaluate this graph data structure?

Answering these questions would help us in analyzing all the dependencies between the Debian packages. Taking the time component into account will make sure that none of the transitive dependencies are missed. This would build upon the existing work in the field and help us in knowing which are the most used packages and most critical ones.
Structure: The Following sections will provide more details about why answering each of the questions is important and what methodology was used in the process. The paper will provide details about collecting the data and finding the optimal measures for criticality and will end with a discussion of the results and the possible future work on the topic.

2 Background

This section will provide some basic terminology about the concepts used during this research. Furthermore, it will provide some more background on the related work in the field.

2.1 Terminology

A *package* is defined as an already existing piece of code that the a developer can reuse to avoid repeating work that has already been done. It helps with modulating the code and maintaining it. Therefore these packages can sometimes be referenced as modules or libraries. When a *dependency* between two packages exists that means that in order to run the dependant piece of code, the user needs to have installed also the dependency. A *transitive dependency* then, is an indirect dependency between two packages. This terminology is combined to define the two main structures that are going to be created and used for the analysis of the Debian Package Manager, namely:

- Dependency graph(also referred to as Dependency Network) - A directed graph structure which is used to illustrate the dependencies between the different packages in the package manager. Each node represents a package and two nodes are connected with a directed edge if one of them depends on the other.
- Time Dependency graph - A dependency graph structure where a time component is also introduced. Each node now has a timestamp with its release date version and the traversal of the graph is done based on a time interval.

The main point of this research is to build upon the previous work done on the topic by introducing the time component in the dependency graph. This would lead to a better representation of all the dependencies in the graph, revealing some transitive dependencies that are lost without it.

2.2 Related Work

The reason why this paper focuses on Debian is that, as already mentioned, it is one of the most popularly used open source distributions. It has gained much popularity over the last two decades and is a topic of many research papers. Gonzalez *et al.* "take it as a proxy to analyse large open source software compilation" in reference [6]. Their research is outdated since it focuses on the versions of Debian between 2.0 and 4.0. An interesting insight is that the number of packages for Debian in this period is growing by an order of magnitude. This means that also the number of dependencies is growing quickly. This is to be expected since newer applications must adapt new functionality. It can be deduced that the number of packages between the time this research was conducted and nowadays is continuing its rapid growth(as it can be seen in Figure 2). This means that the dependency network is expanding as well. One example is the difference between the dependency trees of the package *mozilla* that Gonzalez *et al* analyse in reference [6]. In Debian 2.2 the dependencies are 13 packages. In Debian 4.0 this number rises to 72. Now in the latest Debian version there 389 dependencies for the package *firefox*[2]. This number of growing dependencies combined with the vulnerabilities of using outdated versions is the reason this paper wants to take the dependency analysis one step further and introduce the timestamps of the release dates of the package versions.

There are already several analysis on dependency management and why it is important. De Sousa *et al.* in reference [5] perform an analysis of the Package Dependencies on Debian in 2009. They look at 18 000 packages in the stable, unstable and testing branches and try to find "communities" of packages to help the Debian community by providing "criteria for creation of packaging teams". Pashchenko *et al.* in reference [11] look at the problem from a different perspective and try to understand the decisions that developers take when including dependencies in their projects.

Stringer *et al.* in reference [19] take a deeper look into the *"technical lag"* of dependencies. They define technical lag as the time that an outdated version is used in the dependencies of an application and therefore the application is vulnerable. They conclude that technical lag varies per package manager but it is always there. Usually the time period of this lag is in the range of 0.5-1.5 years. One way to fight it according to their results is to introduce the use of semantic version-

[2] The name of mozilla was changed to firefox in later versions of Debian

ing.[3] This is a convention that aims to give actual meaning to version naming. According to Stringer *et al.'s* results in reference [19], introducing the semantic versioning would reduce lagging by one third. This paper already shows that software developers do not keep their dependencies always up to date and lag behind leaving their applications vulnerable. The research carried out throughout this paper would build upon the work done by Stringer *et al.* since their work focuses mainly on the lagging of the direct dependencies. This paper will try to eliminate the problem of lagging by connecting each version to every possible dependency at the given time period. What is more the Debian package manager was not analysed in the mentioned paper.

The idea of introducing time as a component in graph structures has also been a topic of research [21],[2]. There have been different suggestions as to how to represent it. One way would be as a metadata on the edge, another would be to include it in the information contained in the Node. However, the main idea boils down to filtering out of the graph the edges and the nodes that do not satisfy a given time period.

One important resource that some of the mentioned works above use is Libraries.io.[4] It allows to overview the dependencies in a software project and alerts for new versions of existing packages. Furthermore, it provides useful data for some package managers and could be used to check the validity of a dependency graph without timestamps. Unfortunately, there is not much information about the Debian Package Manager on Libraries.io.

Summarizing the related work, there have been many researches on why managing package dependencies in a software project is important. Most of them focus particularly on the problem of old versions lagging behind, exposing the applications to vulnerabilities. The graph model that is going to be constructed for this paper is heavily influenced by Kikas *et al.*[7]. In addition to the graph structure that is described there, a time component is going to be added to the structure in the form of addition of a timestamp to the node containing the release date of each version.

3 Research Questions

The main focus in this section would be providing the motivation behind each of the research questions that were formulated in the beginning of the paper. It is going to show why they are important and how answering them would lead to the answer of the main research question which is ***"What is the most widely used Debian package?"***

3.1 How to design efficiently a timed dependency graph data structure?

Influenced by previous existing work [7][5],[6], this paper is also going to map the dependency data onto a graph structure. This way it would be easier to compare the results to previous findings and to put them into a broader context. What will set this work apart is the addition of the time component. However, introducing this time component, means that there would be an increase in the overall complexity of the structure and the traversing algorithms. Essentially, the idea for this Graph structure, and the traversal of it, is to be scalable and applicable to different package managers. In Figure 2 we can see the growth of the Debian distribution over the years with the amount of packages increasing drastically. Taking this into account and that other package managers such as NPM[5] and Maven[6] also should be able to use this graph, the solution proposed here should be as efficient as possible.

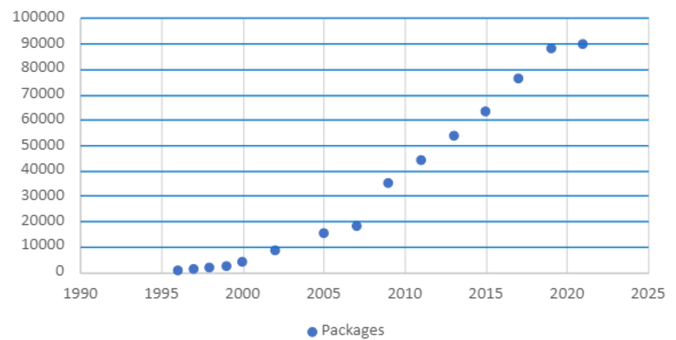

Figure 2: Growth of the number of Debian packages throughout the years

3.2 How does time influence a dependency graph data structure?

The main reason to introduce time into the dependency graph structure is to be as extensive as possible while building the dependency tree of a package. Existing papers [19], [23] show the effect that technical lag has on the major package managers. Until now, this technical lag was discussed in the context of introducing vulnerabilities. However, developers should be able to trace that lag throughout their development and to make an informed decision on which versions they want their application to run on and update their dependencies accordingly. That is why by introducing the timed component in the data structure, would help in modelling all the possible options for developers to choose their dependencies. Furthermore this would provide a basis for deeper analysis in the lagging of transitive dependencies(As defined in Figure 1).

3.3 What measures of criticality could be used to evaluate this graph data structure?

Raw data or just the Graph structure do not provide any useful insights. That is why a measure by which the packages would be ranked needs to be defined. Debian has a project

[3] https://semver.org/
[4] https://libraries.io/
[5] https://www.npmjs.com/
[6] https://maven.apache.org/

called Debian Popularity Contest[7]. The main idea is for users to install the *popularity-contest* package and to let it collect data regarding their package usage. However, since this package does not come into the default installation of the Debian distribution and has to be installed manually, the results contain the so called Selection Bias [10]. Nevertheless, the data collection is done anonymously and presents data such as installation counts, number of users who currently use the package, number of users who do not use the package but have it installed and number of users who upgraded the package recently. This presents some useful information, however poses a few questions. Taken into the context of this paper, what would the number of installations tell about a certain package? Does it make it the most widely used and does it have anything to do with its dependencies? Most importantly, how to define *"most widely used"* in a meaningful way?

Answering these questions in combination with the data that we get from our Graph structure, would provide us with the information on what actually does a widely used package mean. Furthermore, it would provide us with some interesting statistics about the most widely used packages according to our graph.

4 Method

This section will go in depth behind the process of answering the main research questions, namely *"Which is the most widely used Debian package?"*. It will start with the context of the work, describe the data acquisition process, the creation of the timed graph structure and end with the processing of the results.

4.1 Context

This work will study the Debian distribution. Debian is an open-source operating system that dates back to September 1993. Since then, it has become one of the largest and most popular community run distributions. To this day there are 17 major release versions. The latest one called "Bullseye" was released on 14 August 2021 and contained 59,551 binary packages and 31,387 source packages. The Debian-Wiki[8] provides the following definitions about Binary and Source packages: "The programs inside Binary packages are ready to run on the system", "Source packages contain all the necessary files to build the desired software."[9][10]. Furthermore, there are always at least three releases in active maintenance[11] - *"stable"*, *"testing"* and *"unstable"*. The *stable* distribution is the production release of Debian. This is the distribution that is recommended to use by the Debian team since all the packages there are tested and ready to use. The *testing* branch contains packages that are in the queue to be accepted into the stable one. The *unstable* is the distribution where new packages are introduced and developed. This work will focus on analysing the ***stable*** distributions of Debian and will consider both source and binary packages. They will be referred to simply as ***packages*** in the rest of this paper. The time period that is going to be analysed will be from 2015 until 2022. This means that only the active stable versions are chosen - Debian 8, 9, 10 and 11, with Debian 8 being an archived released but still under extended support from the Debian team.

4.2 Data Acquisition

The data that was needed to conduct this research was downloaded directly from the Debian Snapshot Archive.[12] This is an archive maintained by the Debian community which contains Snapshots of the different distributions. Usually, the snapshots are taken every 5-8 hours and date back as far as 2008. The database consists of all past and current packages in Debian and all of their versions. For each snapshot there is a link pointing to the stable distribution at the time. The packages files were scraped from the Debian Snapshot Archive with the help of a web crawler.

The data is then stored in a MongoDB database[13] which helps with formatting it into a JSON file. A JSON file would be suited best to represent the data and would be easiest to map to the Graph. The file structure is represented in Figure 3. The fields that are wanted are the *Package Name*, *Package Version* and each versions would have two fields with its *Timestamp*, which are defined as the time that the package was first encountered in the Snapshot Archive, and the versions that the current package depends. This structure was chosen, so that it would be universal and it would be easier to map data from any package manager. The fields that are chosen can be found in most if not any modern package manager.

```
1  {
2   "name": "libc6",
3   "versions": {
4     "7.38.0-4": {
5       "timestamp": "2015-06-01T04:17:39",
6       "dependencies": {
7          "libc6": ">= 2.17",
8          "libcurl3": "= 7.38.0-4",
9          "zlib1g": ">= 1:1.1.4"
10       }
11     }
12     ...
13  }
14 }
```

Figure 3: JSON structure. Representation of a Pacakge

[7] https://popcon.debian.org/
[8] https://wiki.debian.org/
[9] https://wiki.debian.org/Packaging/SourcePackage
[10] https://wiki.debian.org/Packaging/BinaryPackage
[11] https://wiki.debian.org/DebianReleases
[12] https://snapshot.debian.org/
[13] https://mongodb.com/

There is a specific Debian constraint here that should be mentioned. Debian introduces a package naming convention which also allows developers to include major version updates of a package in the package name. Those updates are entered as a separate entity(i.e package) in the Debian Archive with its own intermediate versions. Figure 4 shows how this works with the gcc packages. As it can be seen there package *gcc* with version 4:6.3.0-4 is released at the same time as *gcc-6* version 6.3.0-18. To avoid failures, gcc-6 is added as a dependency in gcc. This however influences the PageRank which will be run on the Graph and is a point of discussion for future work.

Package	Version	Release Date
gcc	4:4.7.2-1	2015-01-01
gcc	4:4.9.2-2	2015-06-01
gcc	4:6.3.0-4	2017-06-18
gcc-6	6.3.0-18	2017-06-18
gcc-6	6.3.0-18+deb9u1	2018-06-01

Figure 4: Sample of GCC package versions through time

4.3 Graph Creation

The data that was collected in the previous section is now mapped onto a Graph. The Graph is created by parsing the JSON file and constructing a new node for each unique *Package Name* and *Version*. For the rest of the report I will refer to the combination of an unique Package Name and Version as ***PackageID***. For each node of the graph that represents a PackageID there would be a number of edges representing the dependencies. To reduce the amount of information that is stored in the nodes and to make the graph as efficient as possible, a map is built taking each node id an connecting it to the corresponding package information collected from the previous step.

To introduce the time component to the graph, all the possible dependencies should be mapped. That is why each unique PackageID's is taken and mapped to every possible dependency according to the dependencies field in the JSON file and their timestamps. To make the work as universal as possible and as scalable as possible, each of the Versions were translated to use Semantic Versioning. This makes it much easier to compare the versions and to follow them trough time.

One thing worth noting is that different package managers use different versioning. The Masterminds library[14] from Go is used for the general case of version parsing. However, Debian follows a different standard than the one implemented in the library. Unlike semver, which follows the MAJOR.MINOR.PATCH standard, Debian's format is **[epoch:]upstream_version[-debian_revision]**. That is why a separate parser was implemented which was inspired by the work in [15].

[14] https://pkg.go.dev/github.com/Masterminds/semver/v3

The graph is implemented using Go[15]. This programming language was chosen due to its simplicity and efficiency [22],[20] and the fact that it should be able to handle the huge amount of data that we want to map. To make the work easier, the graph was implemented with the help of a Go libray named GoNum[16] which provides a simple framework for a basic directed graph. For the sake of reproducibility, a simple Command Line Interface was implemented to run the code and navigate the options.

To asses the timed graph performance(i.e to answer the second sub-question), a comparison is done between the Debian Package Manager and the results of querying the graph. For a sample of random packages, the command *apt-rdepends* is executed and the results are compared to the resolving of the transitive dependencies using the graph structure.

4.4 Measures of package criticality

To answer the third and last sub-question, it is need to understand how to interpret the data that can be obtained from the graph. The aim is to find the most widely used package but what does "widely used" mean? The idea is to find the packages that have the most dependants and then to rank the importance of these dependencies. This would make future developers more careful when incorporating a package in their application that is "widely" used. Furthermore, it can actually raise awareness towards the critical packages, and persuade the Debian community to monitor them more closely. To find the most important node in the Graph structure, a PageRank algorithm is run. This is a popular metric when it comes to evaluating graph structures. It aims to introduce a metric of importance of a Node and its Edges. The main idea is counting the edges pointing to a node, and the edges pointing out from the node.

Running this algorithm would be done by making use of the existing implementation in the GoNum library. For further reference on the algorithm, one can check the work done by London, Nemeth *et al.* in reference [9]. They present an interesting use of PageRank for ranking scientific articles.

The PageRank version that is ran on the graph first filters out the nodes outside of the given time period, then the results can be aggregated for better overview of the most used packages. To get a grasp on the PageRank per versions of a package one can run the PageRank algorithm without grouping by package name. This provides insights to which is the most popular version of a package at a given time.

5 Results and Validation

5.1 Results

The analysis of the Timed Dependency Graph that has been implemented, will begin by looking at some basic graph statistics which are usually used when evaluating graphs.

[15] https://go.dev/

[16] https://www.gonum.org/

A quick overview can be done by looking at the number of Nodes and Edges. For our Graph $G = (V, E)$ it can be seen that the $V = 244289$ and $E = 6318031$. Here the Vertices represent each Package with its unique version and an Edge represents the dependencies between two packages. Thus, on average there are around 25 dependencies per version. Most of them are concentrated in the main central cluster(as seen in Figure 5). The original input was 86 410 packages. It can be noted that the number of nodes with possible dependencies already increase by almost 3 times. To get a better feeling of this network a Graph visualisation software was used named Graphia[17]. This was one of the few tools that were able to visualize a graph of this size. It can be seen in Figure 5 that there is one huge cluster of dependencies in the middle which is hard to visualize from a closer point of view. There are some other small clusters around the main one. There is a significant part of nodes which are currently not of interest to us, since they have no dependencies.

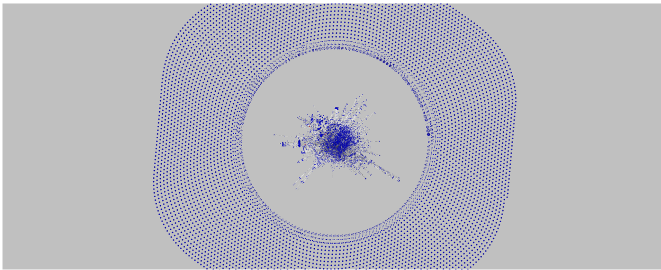

Figure 5: General visualization of the timed graph

To validate the precision of the timed graph structure, one can take a look into the Debian Package Manager. Executing the *"apt-rdepends"* command recursively constructs the dependency tree of a given package. The results of that command can be compared to the the resolving of the graph for 10 arbitrary packages. To formalize this the following method which was formulated in the work of a peer in reference [13] is used:

Let:

- A be the set of transitive dependencies resolved by apt-rdepends
- B be the set of transitive dependencies resolved using the implemented algorithm
- E be the number of dependencies that have a correct name but incorrect version

We calculate the accuracy of the algorithm by this formula:

$$Acc = \begin{cases} 1 - \frac{|A| - |(B \cap A)| + 0.5 * E}{|A|}, & \text{if } A \neq 0 \\ 1, & \text{otherwise} \end{cases} \quad (1)$$

According to this formula, the resolved dependencies using the graph structure, are always at least as accurate as the dependencies from the package manager since on average of 10 random packages, the result is always 1. This is considered when the dataset is not missing any information. This experiment was ran 5 times.

[17] https://graphia.app/

To be able to answer the third Research Question and thus to make a conclusion on which is the most widely used package one can take a look at the results from running the Page Rank algorithm. The data that that the algorithm is ran on is collected between 2015 and 2022.

Firstly, the PageRank was ran for the whole time period. In this experiment, are shown only the unique results, so the packages are aggregated by names. PageRank is non-deterministic algorithm so the results shown in Figure 1 are an average of 10 runs of the algorithm. There may be small deviations when reproducing the experiment but the general trends are always the same.

Package	PageRank
libc6	0.2811
libgcc1	0.1906
multiarch-support	0.0387
libgcc-s1	0.0210
libcrypt1	0.0206
libjs-jquery	0.0132
gcc-6-base	0.0087
perl	0.0072
dpkg	0.0070
libuima-core-java	0.0069

Table 1: Top 10 Debian packages ranked by PageRank for 2015-2022
(Average of 10 runs)

One thing that is noticeable right away is that 6 out of the top 10 ranked packages are library packages. The number one package is *libc6*. By going to the Debian package manager this can be found as the description of that package: *"Contains the standard libraries that are* ***used by nearly all programs on the system****."*. The fact that this is mentioned as **"used by nearly all programs on the system"** serves as validation for the graph and the results provided by the PageRank algorithm. The margin between the PageRank score for *libc6* and the other packages is significantly larger than the difference between the rest of the packages. The package includes basic functionality so it is reasonable that it is so widely used. The *multiarch-support* package ensures compatibility for different architectures so it is also reasonable for it to be ranked so high. It is also worth noting that the package *libgcc1* provides GCC[18] support. It also depends on certain versions of *libc6*, while *libc6* also depends on it. That is why these two packages are co-related as it can be seen in Figure 6 which shows their development through time. What is more their correlation may be due to the fact that virtual packages are not handled well enough in this version of the work.

[18] https://gcc.gnu.org/

This second experiment runs the same PageRank algorithm on the same dataset, however this time each version is considered as a separate entity in the graph so the resulting packages are not unique. The results in Table 2 can be used to analyze the most popular versions of the most popular packages.

Package	Version	PageRank	Release Date
libgcc1	1:8.3.0-6	0.042894	2020-01-01
libgcc1	1:6.3.0-18+deb9u1	0.042891	2018-06-01
libgcc1	1:6.3.0-18	0.042891	2017-06-18
libgcc1	1:4.9.2-10	0.042885	2015-06-01
libgcc1	1:4.7.2-5	0.042885	2015-01-01
libc6	2.31-13+deb11u3	0.026273	2022-06-01
libc6	2.31-13+deb11u2	0.026273	2022-01-01
libc6	2.28-10	0.025670	2020-01-01
libgcc-s1	10.2.1-6	0.024291	2022-01-01
libcrypt1	1:4.4.18-4	0.023923	2022-01-01

Table 2: PageRank results, without grouping the versions.
(Average of 10 runs)

These results should be interpreted carefully since the number of versions influences the overall PageRank scores. That is why *libgcc1* which is ranked second in Table 1 here is ranked first. The timestamps confirm that the graph is actually keeping all the possible dependencies of a package, rather than just a constraint in the form of "<=" or ">=" for the latest version's dependencies. It is also interesting to notice that, although *libgcc-s1* is a new package with only one version, it has similar PageRank score to *libgcc1*.

In Figure 6 is taken a look at the evolution of the top ranked packages throughout the years. Here, the time component has the most influence. The PageRank is ran on each year and filters the packages that are not included in the time period. To be able to get more meaningful results, the data builds upon the results from each previous year i.e the algorithm is first ran on the period between 2015 and 2016, then on 2015 until 2017 and so on, until the whole data set is traversed. This is done in order to avoid losing dependencies on packages which have not released a new version during the last year.

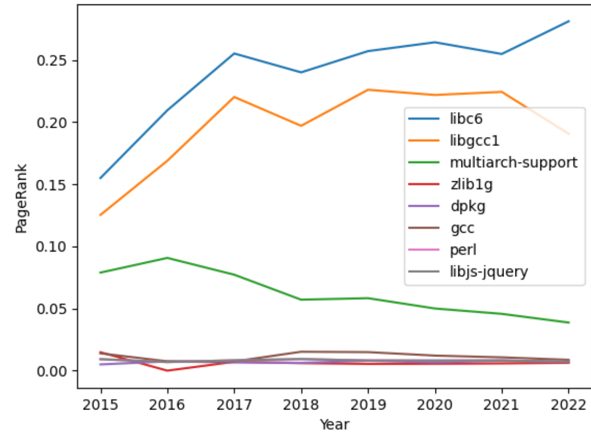

Figure 6: Most popular packages throughout the years 2015-2022, based on PageRank
(Average of 10 runs)

Several remarks should be made about Figure6. Firstly, as mentioned before *libc6* and *libgcc1* are co-related. They tend to change with the same rate with the period between 2021 and 2022 being the only exception. This is due to the fact of the introduction of another version of *libgcc1* called *libgcc1-s1*. While a lot of the projects seem to keep their dependencies on the old version, some migrate to the new one and this results in a drop of the PageRank of *libgcc1*. Similar remarks can be said about the *gcc* packages. The versions there range from 4.x to 10.x which results in a relatively low PageRank score when the versions are looked at individually.

To **summarize** the results of this paper, a time dependent graph structure has been created by constructing a new node for each version of a package and including the time it was first encountered in the Debian Archive as its timestamp. It has been shown that the a time dependency graph structure is accurate when compared to the ground truth provided by the Debian Package Manager. The term "most widely used" was defined as packages with the highest PageRank result in the graph structure that was built. The results of that algorithm show the most widely used packages in Debian are ***"libc6"***, ***"libgcc1"*** and ***"multiarch-support"***.

5.2 Validation and Reproducibility

The data that has been analysed here is downloaded directly from the Debian Archive. This means that it is verified and easily accessible. The timed graph that is constructed for this paper relies on extending the basic GoNum library code for simple graphs. This means that the setup of this research could easily be reproduced. All the code that was used to acquire the data and to construct the graph can be found in the repository of this research. The project is open-source and available on GitHub[19] where a Read Me file is included

[19] https://github.com/teodordob/SoftwareThatMatters

with step by step instructions on how to run the code. Additionally, the data file is uploaded to Zenodo[20] and can be downloaded and referenced [4].

To Validate the results of the PageRank algorithm, one can look into how does the package manager define the packages that are shown as most widely used. As discussed in the results section, the highest ranked package according to our graph, is also described by "used by nearly all programs in the system" and it is co-related with the second one.

6 Responsible Research

This section includes a quick remark about the ethics that were taken into account when conducting this research. As Quinn and Malgieri discuss in reference [14] it is hard to define what sensitive data is. The data that was used during this research is publicly available on the Debian Snapshot Archive. The information about each package can be accessed easily and does not contain any fields that can be considered as private. Following the instructions in the Reproducibility subsection, the results from this research can be reproduced on any machine. What is more, no data regarding humans was collected during this research. Thus, it can be argued that the research conducted as part of this paper is as ethically neutral as possible.

However, the results can still be interpreted in a somewhat ethical context. From an ethical point of view one can say that this paper can influence the future development and use of the *"most widely used Debian packages"*. Users may choose to avoid using these packages due to the vulnerability issues that they may present. On the other hand, this paper may actually increase the awareness of future users, and they may start maintaining their dependencies more carefully.

7 Discussion

To gain a better understanding of the contributions of this research, one can take a look at the results in the context of the Debian Popularity Contest. There, can be seen what is the number of installations of the most widely used packages, the number of users currently using the package and the number of users who have recently upgraded the package. In 3 aggregation of the data with the one from the Popularity contest is shown.

These results are gathered on 18/06/2022. One can clearly see by the ranks of number of installations that it is difficult to find a relation between the most installed packages and the most widely used. It can be observed that on average for every recent update there have been around 8 installations of a package. This could mean that only one in every eight packages is up to date. One thing to notice here is that the Debian Popularity Contest represents a subset of users that can be considered as "conscious". This is because they have voluntarily installed the *popularity contest* package. That can be

[20]https://zenodo.org/

Package	PageRank	Installations(Rank)	In Use	Upgraded
libc6	0.2811	205553(40)	192337	12377
libgcc1	0.1906	133156(389)	16036	70481
multiarch-support	0.0387	75228(1135)	-	75228
libgcc-s1	0.0210	96693(739)	86614	4
libcrypt1	0.0206	96587(742)	81386	10876
libjs-jquery	0.0132	107404(584)	22997	2149
gcc-6-base	0.0087	49719(1649)	-	-
perl	0.0072	203887(94)	128005	23389
dpkg	0.0071	205594(14)	190136	14261

Table 3: Package metrics from Debian Popularity Contest with Number of Installations Rank

interpreted as that they are interested in knowing these statistics. That is why they can be considered as users with above average level of knowledge in this field. An everyday user of Debian would most probably update his software less frequently then the people participating in the Popularity Contest Project.

8 Conclusions and Future Work

The main research questions that this paper aimed to answer was *"Which is the most widely used Debian Package?"* The motivation behind this questions was that it is advised not to use packages with many dependencies. This is due to the fact that whenever a vulnerability occurs somewhere along the dependency tree, the whole system may be affected. Previous work in the field, did not take into account the fact that the time at which a given package was released can also influence its dependency tree. Thus in this paper was provided the concept of a new dependency graph structure that takes into account also the time when a package is released. Furthermore the graph was analyzed to find out which is the package with the most dependencies. This was done by splitting the main question into three sub-tasks: Constructing the Dependency Graph Structure, Introducing the time component and, lastly, analysing how to define *"most widely used"* and what measures to use.

In the end the construction of the graph was done by representing each package as a node and each dependency as a directed edge. To take the time factor into account, a timestamp with the release date of each package was included to each node. The graph can then be filtered according to a time period and traversed. The data that was used was collected from Debian 8,9,10 and 11. The introduction of individual nodes per each unique version resulted in a graph which was almost triple the size of the one which included only the regular packages. In this paper *"most widely used"* was defined as the package that had the most important dependencies. To evaluate this, a PageRank algorithm was run on the timed graph. This resulted in showing that the top 3 most widely used packages for the time period of 2015-2022 were "libc6", "libgcc-s1", "multiarch-support".

Future work on this topic may include optimisations on creating the graph structure. Currently it handles the dataset but operations like filtering according to the timestamps take too much time. What is more, there are package managers which include much bigger datasets which may be hard to handle. An idea would be to switch the programming language to one that can better allocate memory use.

More importantly, Debian contains virtual packages which should be studied more in depth. A virtual package, is defined when a package can combine the functionalities of several others. Virtual packages are defined only logically. If there are dependencies that are mapped to a set of packages that are contained in a virtual package, one can ignore them and just point the dependency to the virtual one. Currently, virtual packages are ignored and the dependencies are looked at individually.

Lastly, the benefit of the precision of the timed graph should be analysed further. Currently, this paper does not mention how in particularly the achieved results can be used to improve the Debian ecosystem. In addition to this, to enhance the PageRank results, a new formula based on the priority *Tag* field in the Packages file can be taken into account.

References

[1] Marcelo Cataldo, Audris Mockus, Jeffrey Roberts, and James Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Trans. Software Eng.*, 35:864–878, 11 2009.

[2] Farah Chanchary and Anil Maheshwari. Time windowed data structures for graphs. *Journal of Graph Algorithms and Applications*, 23(2):191–226, 2019.

[3] Pratim Datta. Hannibal at the gates : Cyberwarfare & the solarwinds sunburst hack. *Journal of Information Technology Teaching Cases*, page 204388692199312, 03 2021.

[4] Teodor Dobrev. Debian data for bachelorsthesis "software that matters".

[5] Orahcio Felício de Sousa, Marcio Argollo, and Thadeu Penna. Analysis of the package dependency on debian gnu/linux. 01 2009.

[6] Jesus Gonzalez-Barahona, Gregorio Robles, Martin Michlmayr, Juan Amor, and Daniel Germán. Macro-level software evolution: A case study of a large software compilation. *Empirical Software Engineering - ESE*, 14:262–285, 06 2009.

[7] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proceedings of the 14th Working Conference on Mining Software Repositories*, MSR '17, pages 102–112. IEEE press, May 2017.

[8] Jannik Laval. Package dependencies analysis and remediation in object-oriented systems. Université des Sciences et Technologie de Lille, 2011.

[9] András London, Tamas Nemeth, András Pluhár, and Tibor Csendes. A local pagerank algorithm for evaluating the importance of scientific articles. *Annales Mathematicae et Informaticae*, 44:131–140, 01 2015.

[10] Aronson JK Nunan D, Bankhead C. Selection bias., 2017.

[11] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications (to appear in acm ccs 2020), 08 2020.

[12] Gede Prana, Abhishek Sharma, Lwin Khin Shar, Darius Foo, Andrew Santosa, Asankhaya Sharma, and David Lo. Out of sight, out of mind? how vulnerable dependencies affect open-source projects. *Empirical Software Engineering*, 26, 07 2021.

[13] Andrei Purcaru. Analyzing the effect of introducing time as a component in python dependency graphs, 2022.

[14] Paul Quinn and Gianclaudio Malgieri. The difficulty of defining sensitive data – the concept of sensitive data

in the eu data protection framework. *SSRN Electronic Journal*, 01 2020.

[15] Romlok. Romlok/python-debian: Python modules to work with debian-related data formats.

[16] Sasindu Shehan. Log4j vulnerability/log4shell vulnerability (cve-2021- 44228). 05 2022.

[17] S. Shiva and Lubna Shala. Software reuse: Research and practice. pages 603–609, 04 2007.

[18] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering*, 26, 05 2021.

[19] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. Technical lag of dependencies in major package managers. 10 2020.

[20] Naohiro Togashi and Vitaly Klyuev. Concurrency in go and java: Performance analysis. pages 213–216, 04 2014.

[21] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. Time-dependent graphs: Definitions, applications, and algorithms. *Data Science and Engineering*, 4:1–15, 12 2019.

[22] Erik Westrup and Fredrik Pettersson. *Using the Go Programming Language in Practice*. PhD thesis, 06 2014.

[23] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesus Gonzalez-Barahona. An empirical analysis of technical lag in npm package dependencies. 04 2018.

[24] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Smallworld with high risks: A study of security threats in the npm ecosystem. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 995–1010, USA, 2019. USENIX Association.