UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Software Engineering Curriculum

Peter Kallaste

How do developers update dependencies in iOS libraries?

Master's Thesis (30 ECTS)

Supervisor(s): Kristiina Rahkema, MSc

Tartu 2023

How do developers update dependencies in iOS libraries?

Abstract:
Software developers use third-party libraries to help with their coding process. Although they are often tested by multiple developers, they still need to be regularly updated. For example, when a library adds a new feature or library contains a bug. It is helpful to know if there are any updating patterns for libraries, so that library developers can better understand their users. Multiple related studies have analysed the updating patterns of many environments, but none are for iOS environments.

The goal of this study is to analyse iOS libraries and find if there are any patterns for updating them. We will use the preexisting Swift Library Dependency Network (Swift LDN) dataset that contains libraries from CocoaPods, Carthage and SwiftPM package managers. We will check if and how the iOS developers update their libraries. During the analysis, we found that iOS developers usually choose the latest version of the library and later do not update that library anymore.

Keywords:
Third-party libraries, data analysis, Neo4j database, iOS

CERCS:
P170 - Computer science, numerical analysis, systems, control

Kuidas tarkvara arendajad uuendavad oma iOS teekide sõltuvusi?

Lühikokkuvõte:
Tarkvara arendajad kasutavad kolmanda osapoolte teeke, et lihtsustada oma arenduse protsessi. Kuigi need teegid on tavaliselt testitud mitmete osapoolte poolt, siis ikkagi leidub olukordi, kus seda teeki on vaja uuendada. Näiteks kui teegile lisatakse uued võimalused või teegi seest leitakse mingi viga. Kasulik on teada, kas leidub mingeid mustreid teeki uuendamise kohta, mis aitaks teekide arendajatel mõista paremini oma kasutajaid. Olemas on mitu sarnast teadustööd, mis uurivad teekide sõltuvuste uuendusi, aga see puudub iOS teekide jaoks.

Selle töö eesmärgiks on analüüsida iOS teeke ja uurida välja, kas leidub mingeid mustreid nende uuendamisete kohta. Selle jaoks, me kasutame olemasolevat Swift LDN andmekogu, mis sisaldab teeke CocoaPods, Carthage ja SwiftPM paketihalduriest. Me vaatame kas ja kuidas iOS teekide arendajad uuendavad oma teekide sõltuvusi. Selle analüüsi käigus me leidsime, et iOS arendajad tavaliselt valivad kõige viimase teegi versiooni projekti tegemisel ja hiljem enam seda ei uuenda.

Võtmesõnad:
Kolmandate osapoolte teegid, andmete analüüs, Neo4j andmebaas, iOS

CERCS:
P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

Table of Contents

1 Introduction

Software developers often use third-party libraries or packages to help them speed up the code development process. Third-party libraries are collections of resources containing different functionalities, which developers can use in their programs or in other libraries [1]. Those are often preferred over custom code because multiple people have tested them [2]. Some popular libraries are, for example, OpenCV[1], which is used for real-time computer vision tasks, and TensorFlow[2], which is for machine learning and artificial intelligence. Furthermore, large companies like Google provide hundreds of different libraries.

To make software developers' lives easier, they can use package managers to control their project dependencies. The primary function of package managers is to install necessary dependencies that are defined in user-created configuration files [3]. There are different types of package managers, but most of them can also do automatic updating, or by writing the version number, it instals that version. Most programming languages have dedicated package managers. For example, Java developers most commonly use Maven[3] or Gradle[4], and JavaScript has npm[5] and Yarn[6].

1.1 Motivation

However, even libraries must be updated regularly because of security issues or new requirements [2]. Because of that, new versions should contain improvements that the previous version did not have, and software developers can decide if they should upgrade to a newer version or stay with the old one. There can be different reasons for the developer to stay with the old version. For example, the new version requires too much effort to update, or the new version contains some bugs [4].

Many research papers have already analysed update patterns of different library dependencies. For example, Kula et al. [4] analysed Java library dependencies, and Salza et al. [3] analysed Android library dependencies. They bring insight into how libraries are updated and if they already follow any trends for updating libraries or bring out trends that developers should follow. However, there are no papers that conduct update analyses on iOS library dependencies. Therefore, our thesis goal would be to analyse iOS library dependencies and compare the result with the other papers. It would be interesting to see if there are any differences across the platform and why they are different or similar. One of the differences may be that Swift language, which is used for developing iOS projects, is not backwards compatible, unlike Java language.

[1] https://opencv.org/

[2] https://www.tensorflow.org/

[3] https://maven.apache.org/

[4] https://gradle.org/

[5] https://www.npmjs.com/

[6] https://yarnpkg.com/

In this thesis, we will use a library dependency network dataset [5] that compiled libraries from CocoaPods, Carthage and Swift PM. Those are the three package managers for iOS projects.

1.2 Research questions

In this thesis, we have formulated the following three research questions:

RQ1: To what extent do developers update their iOS library dependencies?

RQ2: What types of update patterns do developers follow when updating their iOS library dependencies?

RQ3: How do developers respond to vulnerability discovery in iOS libraries?

These research questions should overall answer the thesis question, how developers update dependencies in iOS libraries. The first research question is more general and gives information on how developers update their library dependencies. The second research question dives more in-depth and checks what patterns developers use when updating iOS library dependencies. The third research question will analyse if there are any links between vulnerability discovery and library updates. At the end of the thesis, we will compare those results with other platform results.

This will help us understand how developers update their libraries and allows us to analyse if there is a better way of doing it.

1.3 Thesis structure

This thesis consists of seven chapters. Chapter 2 gives a general overview of related work. Chapter 3 describes the starting point of this thesis. Chapter 4 goes in-depth on how the data is going to be analysed. Chapter 5 presents the results of the analysis. Chapter 6 discusses the results. Chapter 7 concludes the thesis.

2 Related work

Many package manager dependency networks have been studied throughout the years, for example, Kula et al. paper [4] about Java library dependencies. This chapter will describe some of those studies and how they are related to this thesis. One of the largest differences between this thesis and the papers below is that they analyse other programming languages, as there are no papers about iOS library dependency updates.

2.1 Update patterns in Java library dependencies

One of the papers that analysed Java libraries is “Do developers update their library dependencies?” by Kula et al. [4]. Their goal for the paper was to analyse whether developers update their library dependencies and how aware they are of migration opportunities. They analysed 4659 GitHub projects and 2736 library dependencies and conducted surveys.

Similarly to this thesis, Kula et al. paper also formulated a research question where they analysed if Java developers update their library dependencies. During their investigation, they found that most developers often do not update their dependencies, and 81.5% of studied systems have outdated dependencies. Kula et al. pointed out that most developers do not update because it requires much effort that could be used elsewhere. Additionally, they analysed why vulnerability discovery does not affect library updates. They speculated that one of the reasons could be a lack of motivation. It is interesting to compare the results with iOS library dependencies because Java language is backwards compatible, but Swift language is not, and it could affect the results.

One of the differences to Kula et al. paper is that this thesis also analyses library dependencies update patterns.

2.2 Update patterns in Android library dependencies

Research on library dependencies in mobile applications has been carried out on the Android platform [2]. One of the research papers that analyses them is “Do Developers Update Third-Party Libraries in Mobile Apps?” by Salza et al. [2]. They used 291 open-source Android applications from the F-Droid repository and analysed their evolution history.

They found that developers rarely update third-party libraries, and most of the updates are for UI libraries because of new features. For their research, they also looked through some of the communication channels where they were talking about updating libraries and found out that the most likely reasons for not updating are developers' carelessness and high cost/benefit ratio.

Salza et al. paper also contained a research question about Android updating patterns where they categorised 1126 library histories. They divided them into six update patterns, where the biggest were not updating and diligent updating patterns. This is similar to our research question about updating patterns. Salza et al. paper did not include research into developer responsiveness to vulnerability discovery.

3 Background

This section will give an overview of the package managers, Neo4j graph database and dataset that we will refer to in this thesis.

3.1 Package managers

A package manager is a tool used by software developers that is utilised for managing dependencies of software projects. Package managers can install and upgrade dependencies based on user input configuration or remove the dependencies when they are no longer needed [6]. Package managers are often capable of installing direct and transitive dependencies. Direct dependencies are packages that developers' projects use directly. They are specified in the package manager manifest file. Transitive dependencies are the libraries that other dependencies use, which are needed for them to function correctly.

In this thesis, we will be examining libraries used in Swift projects. There are three prominent package managers for Swift development. They are CocoaPods, Carthage and SwiftPM.

CocoaPods was released in September 2011 and is a package manager for XCode projects [7]. XCode is one of the applications that can be used for iOS app development. To create a library for CocoaPods, the developer needs to specify a *podspec* file containing the name, version, source and other fields later used in library details. To use a dependency from the CocoaPods platform, the developer needs to specify the *podfile* with the dependency name and version.

Carthage is a decentralised package manager meaning it does not have a central repository where developers can access Carthage dependencies [8]. Swift developer only needs to specify the source location then it downloads and builds the dependencies for the project. However, the developer still needs to manually add it to the project, giving them complete control of the project structure.

The Swift Package Manager (Swift PM) is the official package manager for Swift code. Swift PM is also a decentralised package manager [9]. Swift PM was released in December 2017, but only after Swift 5 was released in March 2019 it can build applications for iOS platforms.

Additionally, when libraries are updated to package managers, then they usually use Semantic Versioning Specification (SemVer) scheme [10]. SemVer numbering consists of three parts: major, minor and patch. For example, if we have version number 1.2.3, then 1 is the major part, 2 is the minor part, and 3 is a patch part of the version.

3.2 Neo4j graph platform

For this thesis, we will be using the Neo4j environment for analysis. To give some context, we will give a brief introduction to Neo4j and Cypher language. Neo4j is a popular graph database system that easily handles large and highly connected databases [11]. The neo4j graph database is divided into nodes, relationships, properties and labels.

Nodes represent entities in a graph and are labelled, depicting the type of an entity. For example, we will have nodes with `Library` and `App` labels. Nodes can also contain properties, which are key-value pairs that can store any sort of data. For example, a `Library` node can have name and version properties.

Relationships are connections between node pairs. They are also named and are also directional. Therefore, they have a start and end node where direction shows which way they are connected. Like nodes, relationships can also have properties with different key-value pairs.

Cypher is a graph query language used by the Neo4j graph management system [12]. It allows for creating queries that can retrieve and create data for the Neo4j database. Cypher was inspired by SQL language and because of that, Cypher query syntax is very similar to SQL syntax [12]. In Cypher, nodes and relationships are represented with `(:Node)-[:RELATIONSHIP] ->(:Node)` syntax, where nodes are between parentheses and relationships are between square brackets. Angle brackets represent in which direction the relationship is pointing or oriented. This syntax system allows us to create graph patterns that help us to retrieve the correct data. The following pattern can then be inserted into the `MATCH` keyword, which matches the pattern with database data and with the `RETURN` keyword, it can be retrieved from the database. Additionally, there is the `WHERE` keyword that allows the creation of conditional clauses for filtering the data.

```
MATCH (a:App)-[:DEPENDS_ON]->(l:Library)
WHERE l.name = "shibapm/rocket"
RETURN a
```

Figure 1. Cypher language example

Figure 1 represents an example query that gets all app versions that use *shibapm/rocket* library. Cypher language allows us to create complicated queries to analyse the Swift LDN dataset.

3.3 Swift LDN dataset

This thesis aims to analyse and compare the results of update patterns for iOS library dependencies. For this reason, we need a dataset containing a dependency network of iOS libraries. Such a dataset was described in a research paper, "Dataset: dependency networks of open source libraries available through CocoaPods, Carthage and Swift PM" [5]. They collected libraries from CocoaPods, Carthage and Swift PM package managers. Additionally, they utilised the NVD database to search for known vulnerabilities in those libraries. In the research paper, they called it the Swift Library Dependency Network (Swift LDN) dataset that we will also use throughout this thesis.

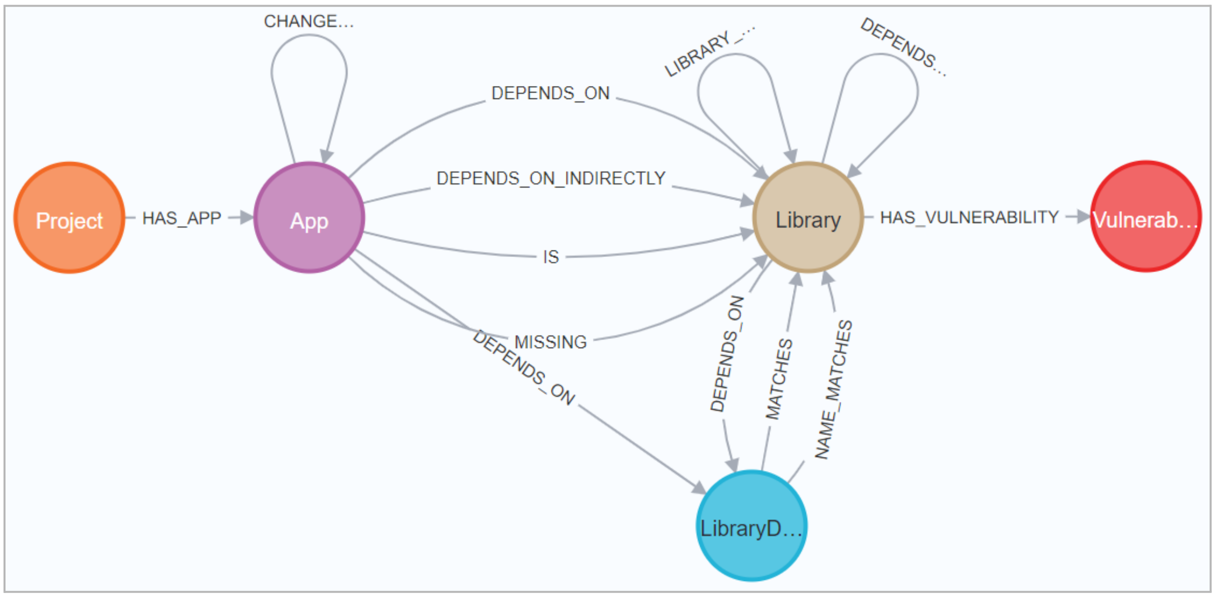

Figure 2. Structure of the Swift LDN dataset [5]

Swift LDN data is stored inside the Neo4j[7] dataset. As represented on Figure 2, in the dataset of the Swift LDN contains five different nodes with labels:

- `Project` - stores project repository information
- `App` - stores different project versions
- `Library` - stores libraries that were collected from the project resolution file
- `LibraryDependency` - stores libraries that were collected from the project manifest file
- `Vulnerability` - stores known library vulnerabilities

Table 1. Node count in Swift LDN dataset [5]

Project	**App**	**Library**	**LibraryDependency**	**Vulnerability**
75550	572131	576144	19390	159

The total count of all of these nodes is represented in Table 1. From there, we can see that the Swift LDN dataset contains application information from 75550 project repositories. Along with that, there are 159 different vulnerabilities for those libraries.

Additionally, as seen in Figure 2, there are 13 different relationships with patterns:

- `(:Project)-[:HAS_APP]->(:App)` - indicates from which project the app is from.

[7] https://neo4j.com/

- `(:App)-[:CHANGE_TO]->(:Library)` - indicates application next version change.
- `(:App)-[:DEPENDS_ON]->(:Library)`, `(:App)-[:DEPENDS_ON]->(:LibraryDefinition)`, `(:Library)-[:DEPENDS_ON]->(:Library)`, `(:Library)-[:DEPENDS_ON]->(:LibraryDefinition)`, `(:Library)-[:LIBRARY_DEPENDS_ON]->(:Library)` - indicates which are direct dependencies of a library.
- `(:App)-[:DEPENDS_ON_INDIRECLY]->(:Library)` - indicates which are transitive dependencies of a library.
- `(:App)-[:IS]->(:Library)` - indicates which application and library are the same.
- `(:App)-[:MISSING]->(:Library)` - indicates that the resolution file is missing from the project and the library version is determined based on the manifest file.
- `(:LibraryDefinition)-[:MATCHES]->(:Library)`, `(:LibraryDefinition)-[:MATCHES]->(:Library)` - indicates that LibraryDefinition matches completely or either by name with the Library node.
- `(Library)-[:HAS_VULNERABILITY]->(:Vulnerability)` - indicates which libraries contain vulnerabilities.

We will use multiple of these relationships and nodes to create an analysis for our thesis.

4 Methodology

One of this thesis's main goals is to analyse the Swift LDN dataset and find out how developers update iOS libraries. We have formulated some research questions that would help us to compare the results with other related works. Therefore, the research questions are chosen similarly to those brought out in those works.

4.1 Using the Swift LDN dataset

For the purposes of this project, Python scripts will be created to analyse the database. Python has a Neo4j driver library which easily allows us to make the connection to the database and read the graph. However, to read the Neo4j data, it first needs to establish a connection with the Neo4j database. For that, we are going to run the Neo4j server on the Debian server environment. We need to install *openjdk-11-jdk* and *neo4j* packages that are required for running the database and configure the Neo4j server to allow connections from outside the network. Then we will download the Swift LDN dataset and load it into the Neo4j server. This would now allow us to access the Swift LDN dataset from the Neo4j server and easily access the data for our Python script.

```
// Creating new relationship between two library nodes
MATCH (l:Library)<-[:IS]-(a:App)-[:CHANGED_TO]->(a2:App)-[:IS]->(l2:Library)
MERGE (l)-[:CHANGED_TO]->(l2);

// Adding indexes for faster lookup
CREATE INDEX app_timestamp_index IF NOT EXISTS FOR (a:App) ON (a.timestamp);
CREATE INDEX app_name_index IF NOT EXISTS FOR (a:App) ON (a.name);

CREATE INDEX library_name_index IF NOT EXISTS FOR (l:Library) ON (l.name);
CREATE INDEX library_version_index IF NOT EXISTS FOR (l:Library) ON (l.version);
```

Figure 3. Cypher query for improving dataset

Additionally, to help us with the analysing procedure, we added an additional relationship that can be seen in Figure 3. It will create a direct relationship between two updated library nodes. Furthermore, Figure 3 additional contains queries for indexing the database. Indexing the database will help with speeding up the lookup performance of the database. When creating an index, it creates an additional data storage for indexed property values that later the Neo4j database engine can use for finding quickly relevant nodes. The following indexes were chosen based on the queries that were created during this thesis.

4.2 RQ1: Library dependency updating

The first research question is a preliminary question that helps us understand if iOS developers generally update their library dependencies. For that, we are going to check if and how many times the library upgraded, downgraded or stayed with the same version.

First of all, we need to establish a connection with our Neo4j database. This requires us to use a Python library for Neo4j and configure it to connect to our database. Then we will need to create a cypher query for returning the upgrade/downgrade counts. Using the `CHANGE_TO` relationship, we can check where there has been a new library version. Additionally, with the `DEPENDS_ON` relationship, we will see if the library changed existing dependencies and if the change was upgrading or downgrading the library. However, if there has not been any change, then it will count as not updated. For simplicity reasons, we will create separate queries for upgrade and downgrade counts as we can easily change the `CHANGE_TO` relationship direction, and it will change from an upgrade to a downgrade or a downgrade to an upgrade.

It is also interesting to see how many dependencies there are overall, on average, for a Swift library. We will select all the latest library versions and count the number of dependencies that have been defined in the project manifest.

Created upgrade, downgrade and dependency count queries that will be used in the results are brought out in Appendix 1. Given this information, we will see how many times the developers have upgraded or downgraded their library dependencies and compare the results with other similar papers.

4.3 RQ2: Dependency updating patterns

With the second research question, we want to find out if there are any patterns or trends for updating iOS libraries. For that, we are going to see if developers update their library, then how many will use the new version in their projects. This approach is from the view of a library developer. It will be interesting to see whether other developers will adopt the library version once the iOS developer releases the new update. We will select libraries by their usage count (popular libraries) and manually analyse some of the first ones.

First, we need to get the most popular libraries from the Swift LDN dataset. We can use the `DEPENDS_ON` relationship and count the number of libraries found for each `App` node. The result will then be sorted by usage count and displayed in a graph. This allows us to select libraries that we want to analyse further. Then we need to determine what library versions we want to plot a graph. For that, we first can use the `IS` relationship on the Library node to find out the date of the library release and again use the `DEPENDS_ON` relationship to collect library usages based on the library version. From the query result, we can pick out the library version for getting version usage trends over time. Then we will use a complex query that gets all the months from the library release timestamp to our selected period, for which we choose 03/2022 as the last library release in the dataset was from 02/2022. This allows us to check how many usages the library version had every month. Next, we need to use the `DEPENDS_ON` and `CHANGED_TO` relationship and get the last library version every month to check if the dependency is there. If it is, it will be counted for the library version. This will return as with library version usages aggregated to every month.

Completed query can also be checked in Appendix 1.

4.4 RQ3: Vulnerability affect on updates

The third research question allows us to see if and how iOS developers react to vulnerability discovery in the library. For every library vulnerability, we would see if the usage has decreased in projects and how many implement the new version.

To implement this, we need to check when the vulnerability was published and what versions of the library were affected. Therefore, we need to use the publishedDate property from the `Vulnerability` node and `HAS_VULNERABILITY` relationship that shows what libraries were affected by that vulnerability. The resulting function for retrieving vulnerable library versions can be found in Appendix 1. This allows us to create a table with *vulnerability_id*, *library_name*, *publish_date*, *usages* and *vulnerable_versions*. From the table, we can get the vulnerable versions and their subsequent versions that need to be examined and then plot them with a usage graph function that was created for RQ2 to analyse any trend changes. We will be plotting the graphs for 3 most popular libraries with vulnerabilities, as there are not many vulnerabilities in the Swift LDN database and other libraries are with few usages and not very popular.

5 Results

In the following sections, we will describe the results of the dependency analysis. The results are based on the methods that were described before and can be viewed with all the outputs in the Jupyter Notebook[8].

5.1 RQ1: Library dependency updating

For the first research question, we want to know how many libraries have updated their dependencies. First, we are going to check how many libraries have upgraded their dependencies and then how many libraries have downgraded their dependencies.

From the Swift LDN dataset, we found that there are a total of 7024 different iOS libraries that use dependencies, but only 6602 of those libraries have released new versions.

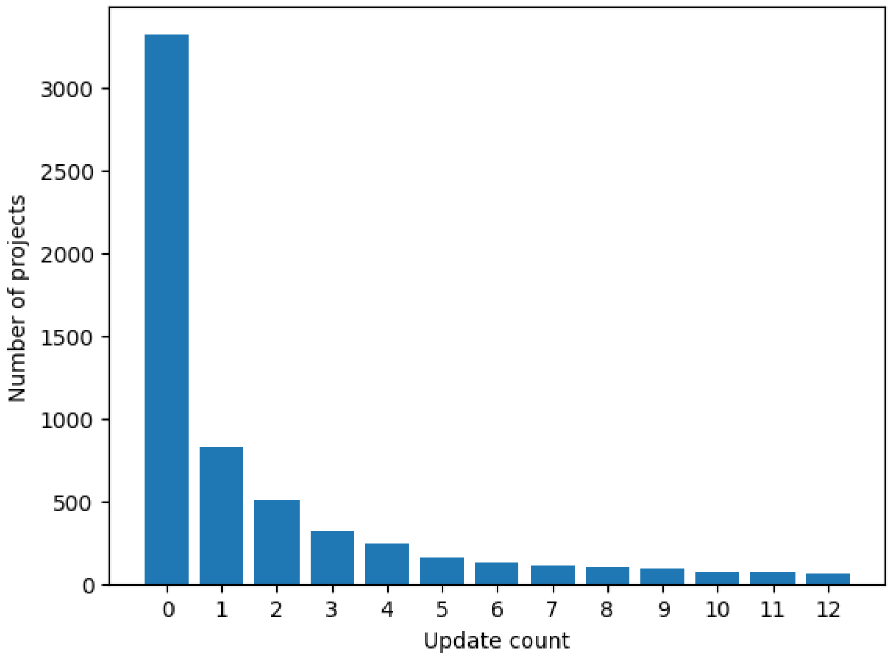

Figure 4. Update count for iOS libraries

Figure 4 shows that 3322 of the libraries do not update their dependencies, while 3280 have updated their dependencies at least once. 827 libraries updated their dependencies once, 508 libraries updated twice, 319 libraries updated three times, and so on. The table shows that the number of dependencies steadily decreases as the update count increases, with only 59 libraries updating their dependencies twelve times. Additionally, there are 591 libraries that have upgraded over 12 times and the maximum update count is 827. This information would

[8] https://github.com/PeterkalM/swift-ldn-notebook/blob/main/swift-ldn-notebook.ipynb

suggest that most iOS libraries do not update their dependencies frequently, and those that update are in the minority.

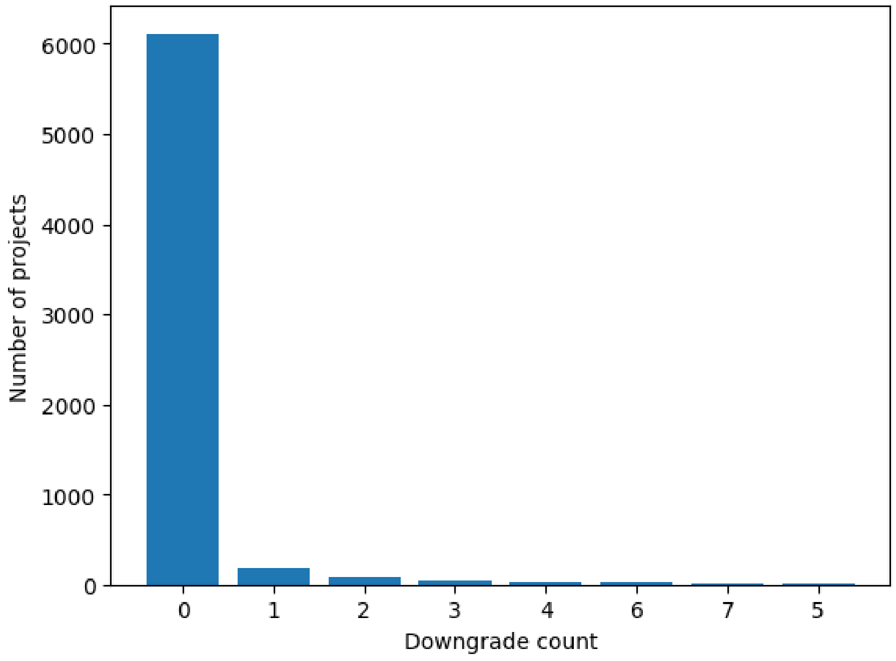

Figure 5. Downgrade count for iOS libraries

In Figure 5, we present a visual representation of the number of downgrades that have occurred in projects with dependencies. The figure shows that the majority of the projects have not downgraded libraries. Only 485 of the libraries have ever downgraded dependencies, and 6117 libraries have not. Specifically, 180 of the libraries have downgraded their dependencies once, and 74 libraries downgraded twice. Furthermore, only 281 libraries have downgraded their dependencies less than 50 times. Additionally, there are 111 libraries that have been downgraded over 5 times and the maximum count is 74 updates.

Table 2. Dependency count statistics for libraries

mean	std	min	25%	50%	75%	max
2.42	2.39	1	1	2	3	47

We also created a query to check how many dependencies are in each latest library. The resulting data is brought out in Table 2. It shows that the library has 2.41 dependencies on average, but there is also a library with 47 dependencies. This shows that most libraries have very few libraries, and the update count could be low because of that.

5.2 RQ2: Dependency updating patterns

For RQ2, we are trying to find out if there are any patterns for updating iOS dependencies. This would help us understand if and how developers update their library dependencies and if there are any correlations with iOS and other platforms.

As there are over 6000 libraries that have released new versions, we will be only investigating update patterns for more popular libraries. We will base the popularity by how many projects have used the library in any given time.

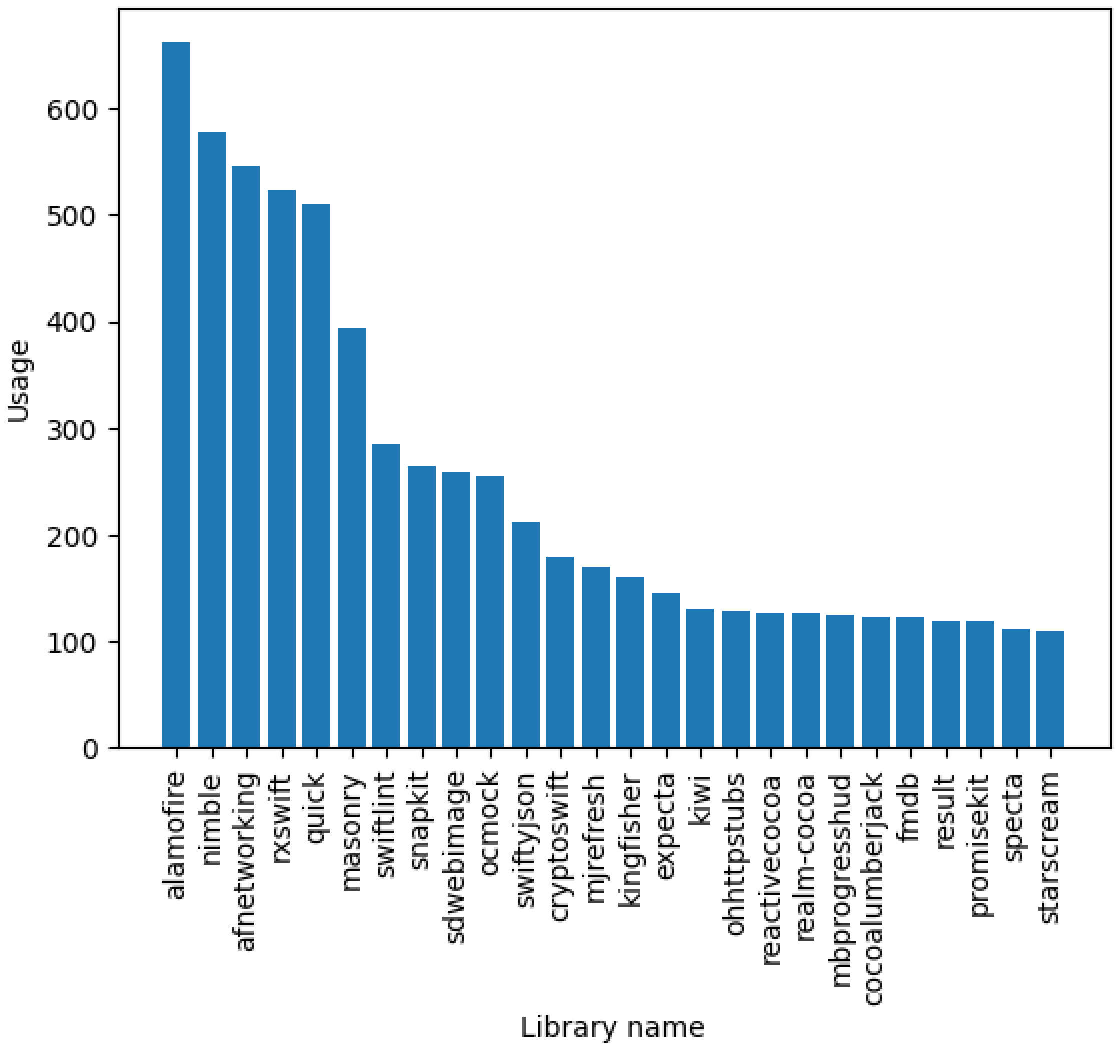

Figure 6. Most popular iOS libraries

From Figure 6, we can see that there are a total of 5 libraries that have over 500 usages. Those top-5 libraries are alamofire, nimble, afnetworking, rxswift and quick. With alamofire library having 662 usages, nimble library having 578 usages, afnetworking 545 usages, rxswift 524 usages and quick 510 usages. We found out that on average iOS developers have updated their libraries dependencies only 4 times, and library update standard deviation is 22.

Additionally, we create a version usage history graphs for those top-5 libraries that would show how many libraries have used that particular library version as dependency. This helps us understand what versions we should look at more closely.

5.2.1 Results for alamofire library

Alamofire is a popular networking library for iOS applications [13]. It features HTTP request handles, JSON parsing, and similar features.

Figure 7 represents version-specific usages for the alamofire library. There are three significant spikes for versions 4.7.3, 4.8.2 and 4.9.1. They are the last patch updates for those versions. Furthermore, the next major update, notably version 5 and so forth, has fewer usages. This could indicate that developers that use alamofire are reluctant to upgrade to the next major version or prefer to use the next minor or patch versions.

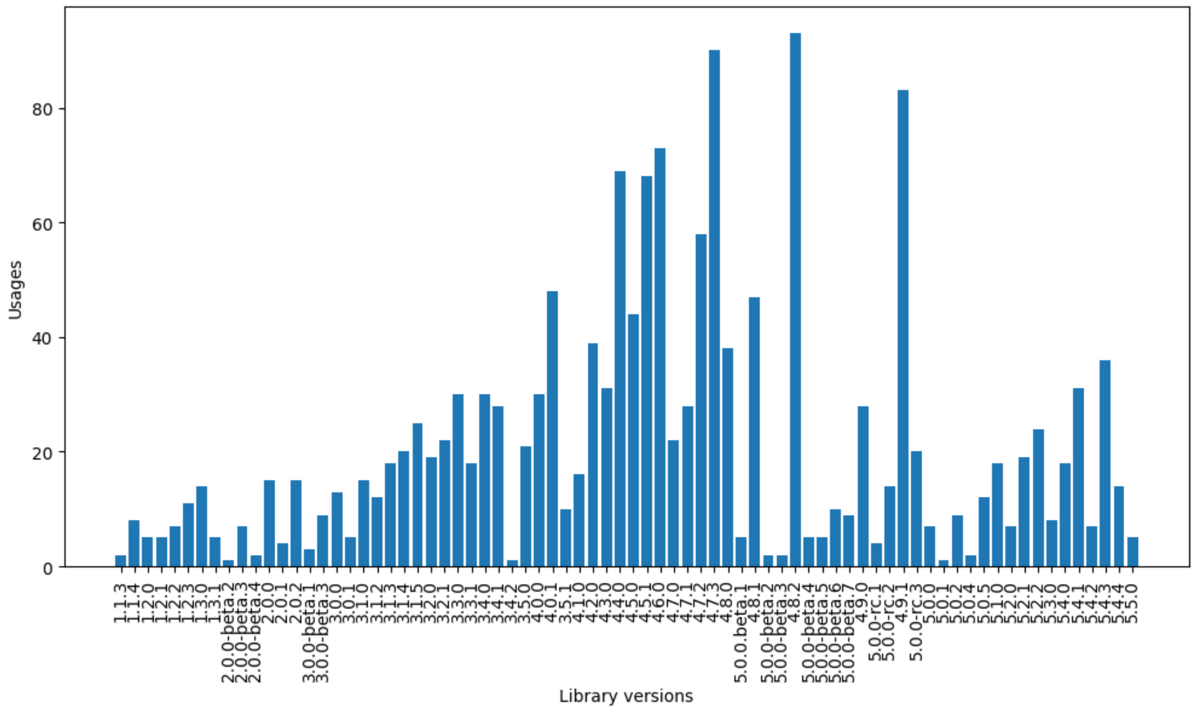

Figure 7. Version usage history for alamofire library

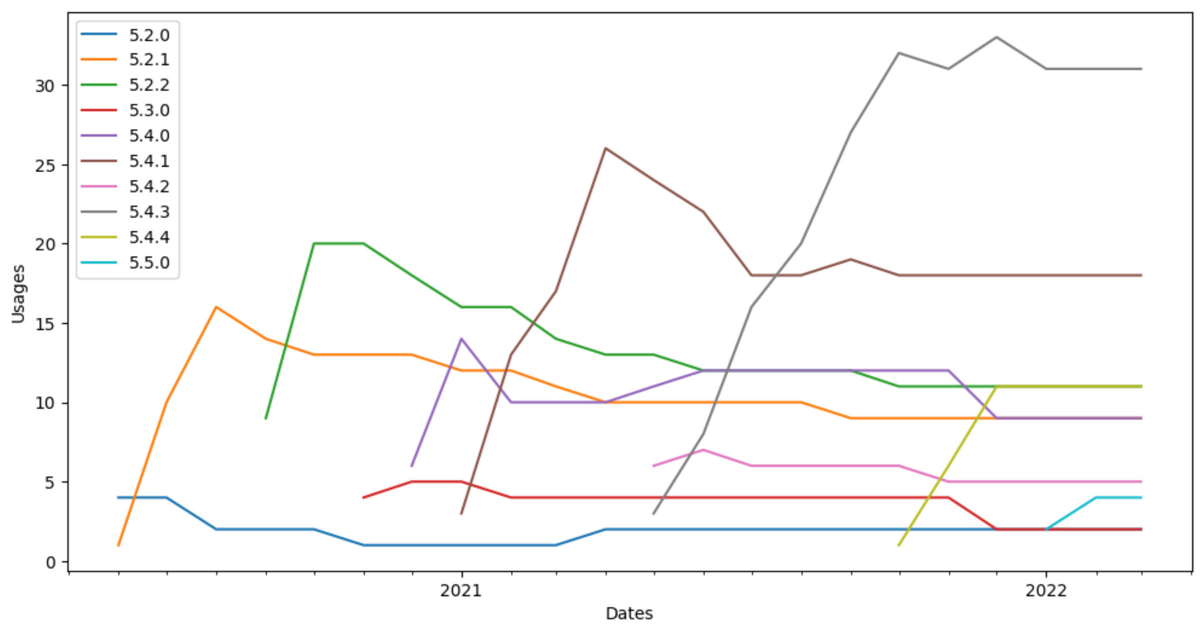

Figure 8. Version usage history over time for alamofire library

For further analysis, we selected the last ten alamofire versions and checked how many libraries use it as a dependency in each month. Figure 8 shows that versions 5.2.0 and 5.2.1 were released in the same month, and developers were usually using the library with the newer version. When the new version came out, 5.2.2, then usage of the old ones only dropped slightly. But mostly, we see a trend that if a new version comes out, then the version usage goes up significantly initially and, after a couple of months, is relatively steady. This would indicate that there are a large number of users who update straight away. However, a considerable amount stays with old versions.

5.2.2 Results for nimble library

Nimble is one of the popular testing frameworks developed for Swift applications [14]. It has multiple assert functions that allow the user to test how the data should look and match it with expected values.

Figure 9 shows the overall usage of the nimble library versions. There are multiple versions of libraries that have fewer than five total usages. Because of that, we will skip those and select versions with a higher number of total usages for further analysis. However, those smaller versions were also checked, and they were usually released within a few days apart. Therefore, when the iOS developer updated the library, then they probably just chose the version which was most recently released.

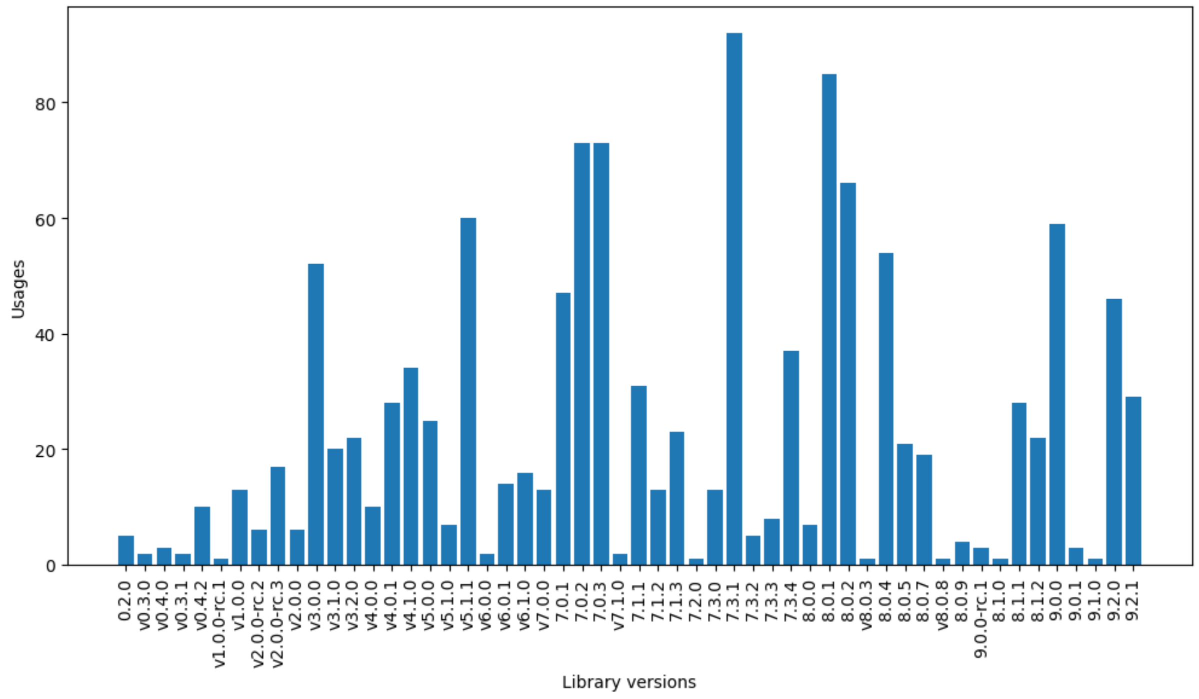

Figure 9. Version usage history for nimble library

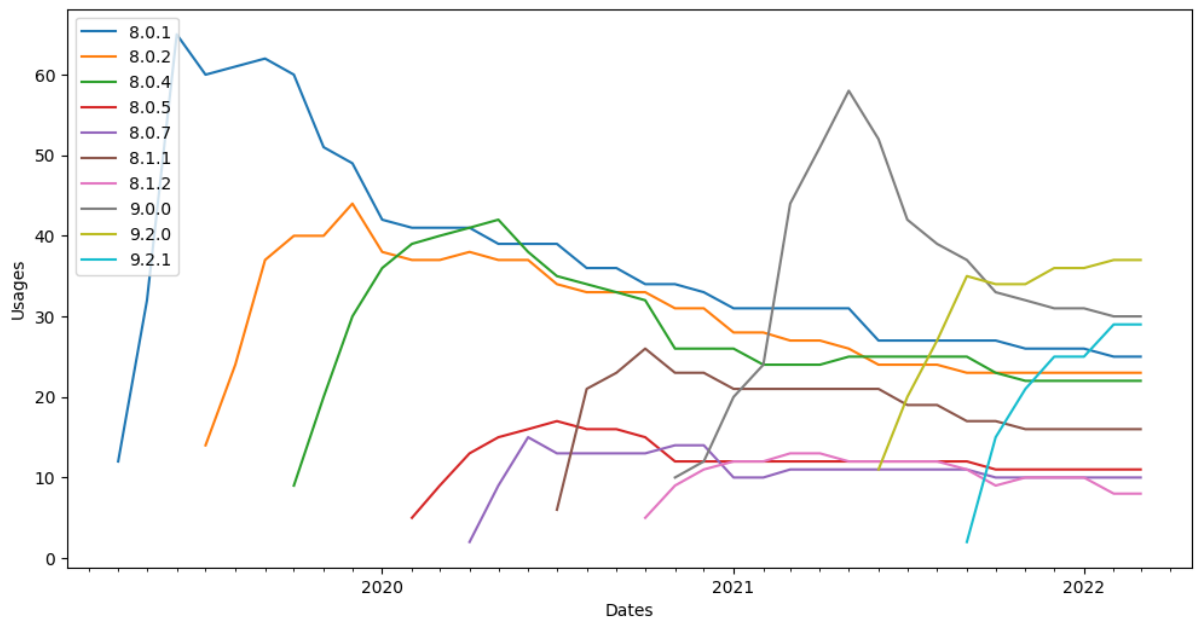

Figure 10. Version usage history over time for nimble library

Figure 10 shows the last ten popular versions of the nimble library. From there, it can be seen that after the initial version release, there are a bunch of usages, and usage will stabilise and stay the same after a couple of months. This is a similar result to the alamofire library that we analysed in the previous chapter. In Figure 10, it is seen that after the 9.0.0 version was released, it became most popular quite quickly, and after a new release of the library, then 9.0.0 usage dropped, and 9.2.0 increased.

5.2.3 Results for afnetworking library

Afnetworking is a networking library similar to alamofire [15]. It also simplifies managing HTTP requests and has other similar features. As of January 17, 2023, the afnetworking library is deprecated, and it is now recommended to use the alamofire library.

Figure 11 represents the version usage history for afnetworking library. It shows that multiple versions are quite popular, but only recent versions have much more usage. Therefore, we select the last eight libraries that we will analyse further.

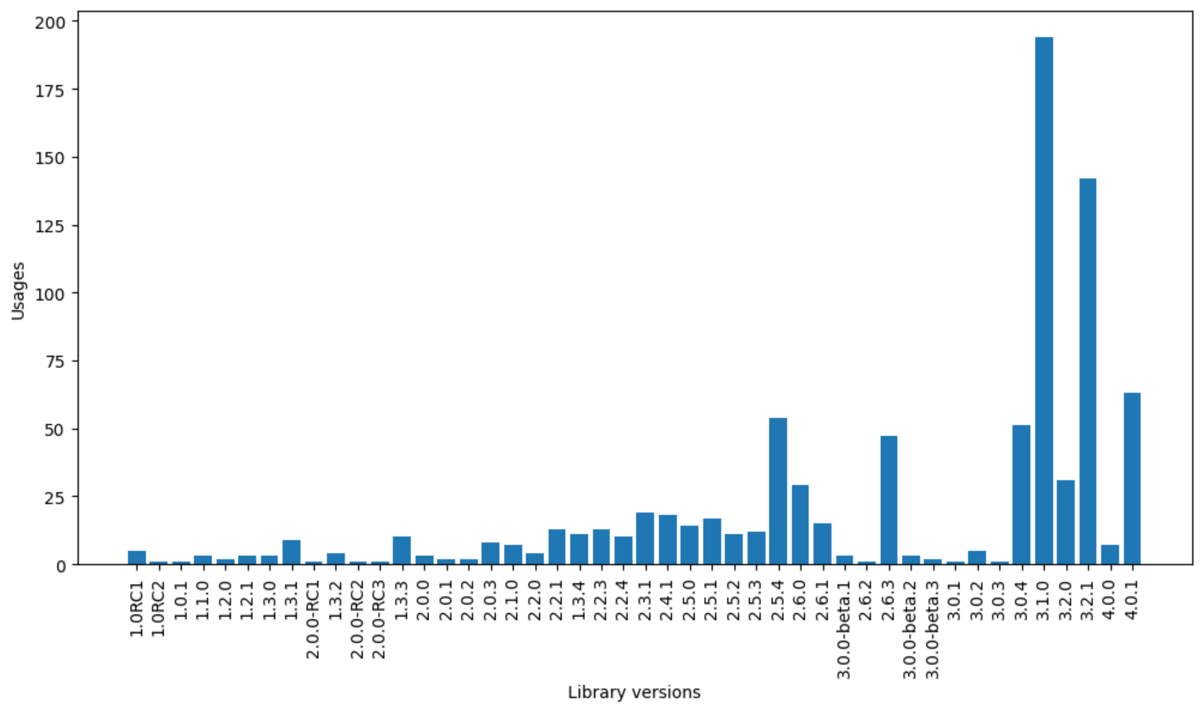

Figure 11. Version usage history for afnetworking library

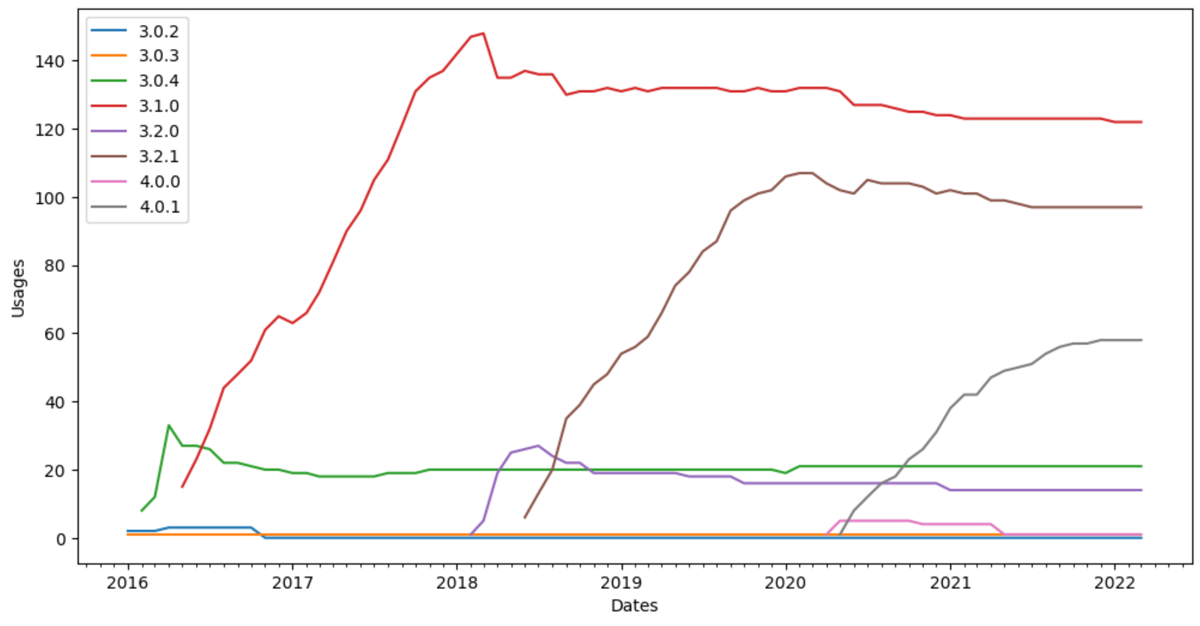

Figure 12. Version usage history over time for afnetworking library

Figure 12 shows the last eight popular versions of the afnetworking library. It shows that developers usually use the last available version of the afnetworking library. When the new library version is released, the old library version usage stays the same and new version usage grows. Afnetworking library also made very few releases over the six years, which could indicate that library is stable enough, and developers do not want to update to a new version.

5.3 RQ3: Vulnerability affect on updates

For the third research question, we wanted to find out if discovery of a vulnerability in a library would affect the usual update pattern for that library.

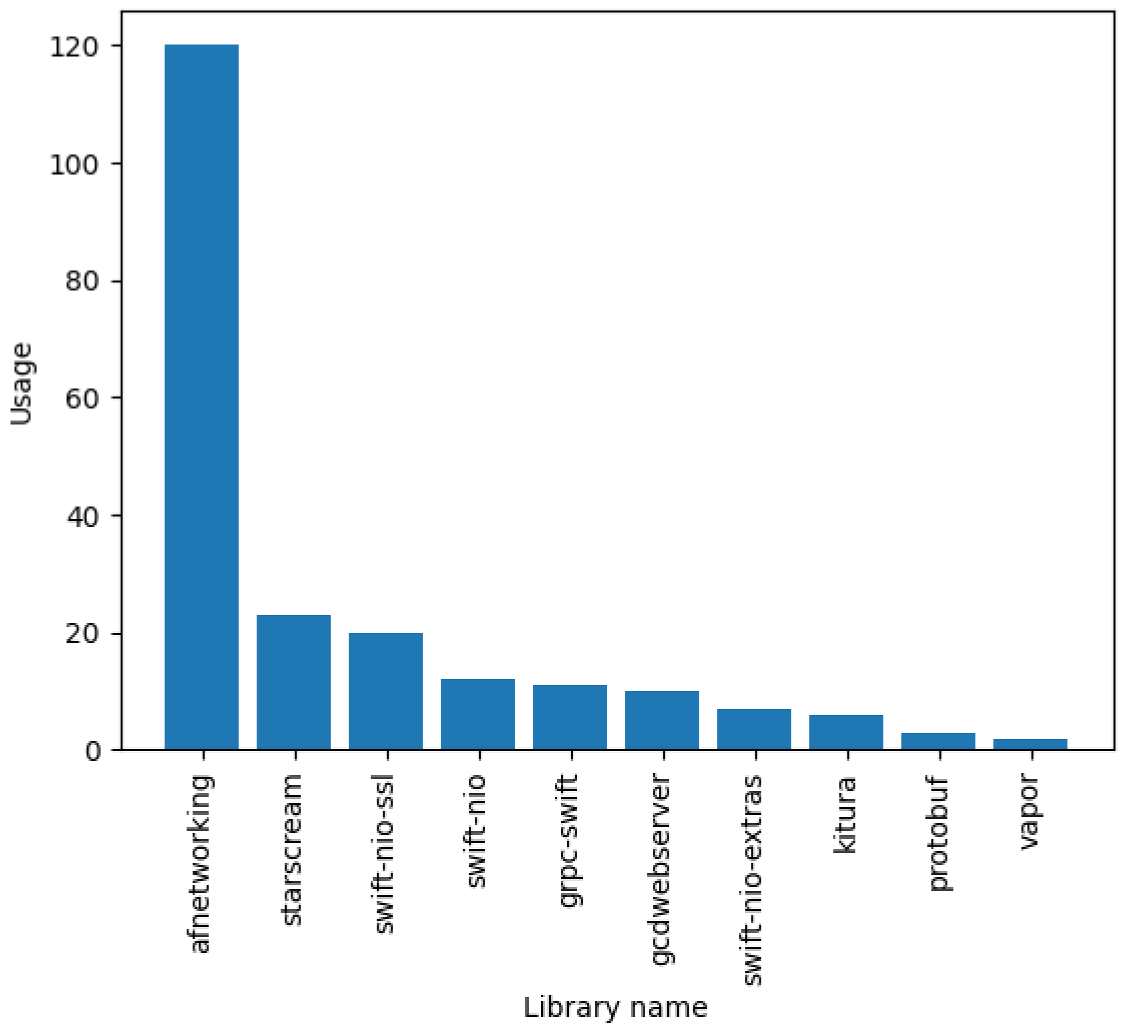

Figure 13. Libraries with vulnerabilities by popularity

We found out that there are a total of 147 vulnerabilities in 39 different libraries for Swift LDN. In Figure 13, we see that the most popular library where vulnerability was found is afnetworking. Afnetworking library usage was 120 while other libraries had usages of 23 and 20 which are considerably lower.

5.3.1 Vulnerability in afnetworking library

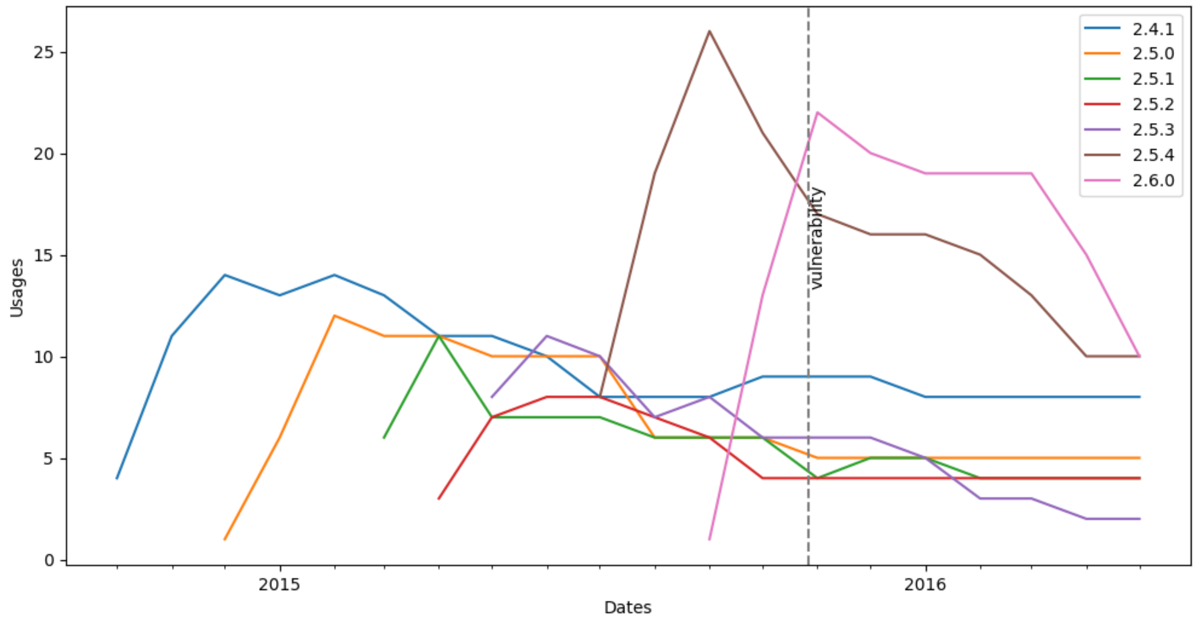

Figure 14. Afnetworking library usages with vulnerability

Found vulnerability entry code was CVE-2015-3996[9] and it affected 2.5.2 version and below. Based on Figure 11, we selected three versions after and before the library version where the vulnerability was found and made a graph about its version usages. In Figure 14, it can be seen that after discovery of the vulnerability, which was published on 27/10/2015, there were not any major changes to affected version usages in the graph. When the vulnerability was published then, there was already a new version with a fixed version. New users were already using the new version, and there were no significant changes in old versions.

5.3.2 Vulnerability in starscream library

The next two vulnerabilities (CVE-2017-5887[10] and CVE-2017-7192[11]) we will analyse where found in the *starscream* library. They were published on the same date, 06/04/2017 and affected the same library versions, version 2.0.3 and below. Therefore, based on Figure 15, we selected versions that were next to the vulnerability and plotted them in Figure 16. It shows that the affected versions were unpopular (fewer than five usages) and had no significant changes. Initially, there was one additional usage for version 2.0.3 as there were no new versions, but later one of the libraries updated to a new 2.1.0 version.

[9] https://nvd.nist.gov/vuln/detail/CVE-2015-3996
[10] https://nvd.nist.gov/vuln/detail/CVE-2017-5887
[11] https://nvd.nist.gov/vuln/detail/CVE-2017-7192

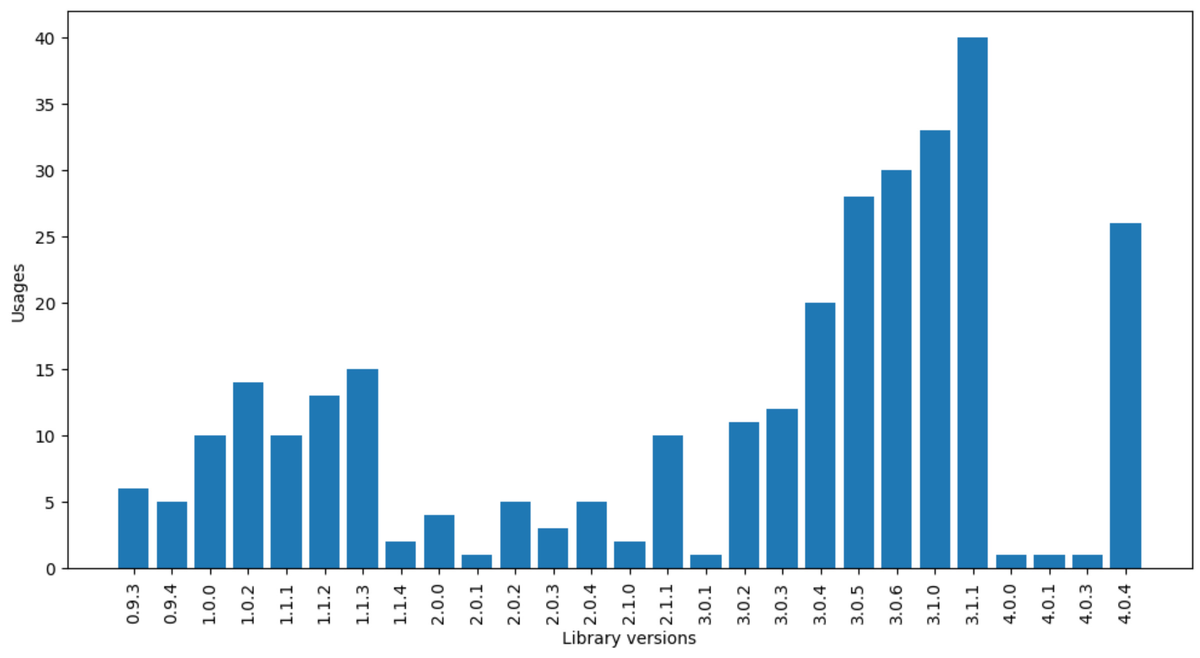

Figure 15. Version usage history for starscream library

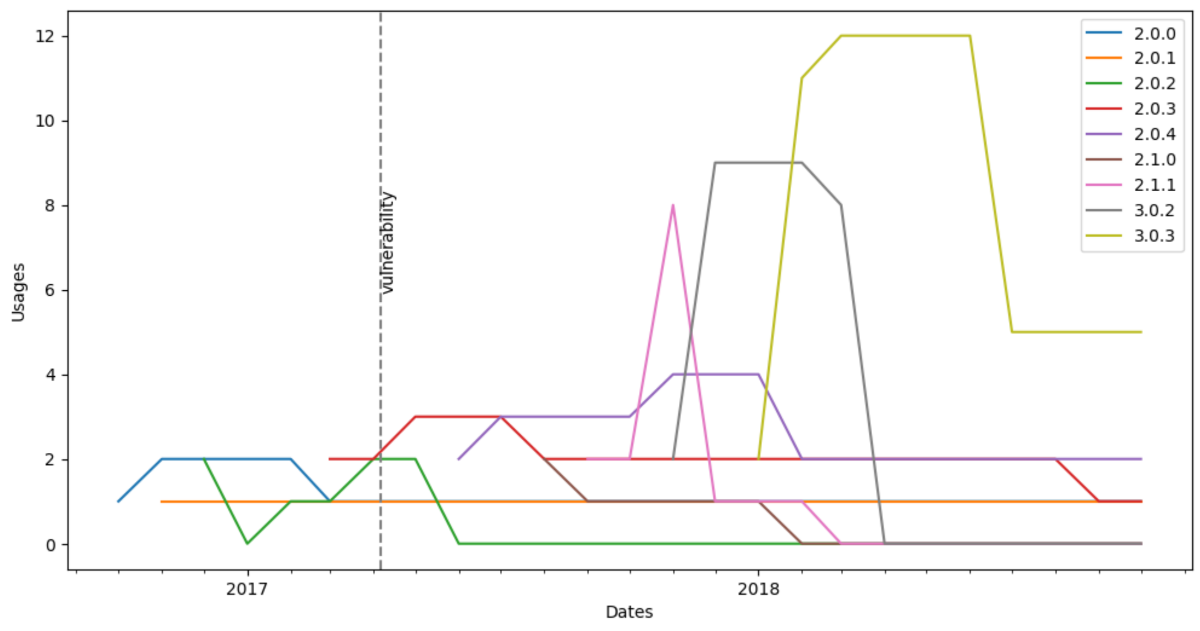

Figure 16. Starscream library usages with vulnerability

5.3.3 Vulnerability in swift-nio-ssl library

The next vulnerability by popularity was *swift-nio-ssl* library. The vulnerability entry code was CVE-2019-8849[12] and was published on 18/12/2019. The vulnerability affected version 2.4.0 and below. Based on Figure 17, we saw there were a few versions of this library and we displayed all of the versions in Figure 18. It shows that during the publishing date, there was already a fixed version available, and there were not any trends changes. When analysing what the libraries were, we found that they were all forks of *grpc/grpc-swift* library, and when it updated the version, then all other forks also updated their versions.

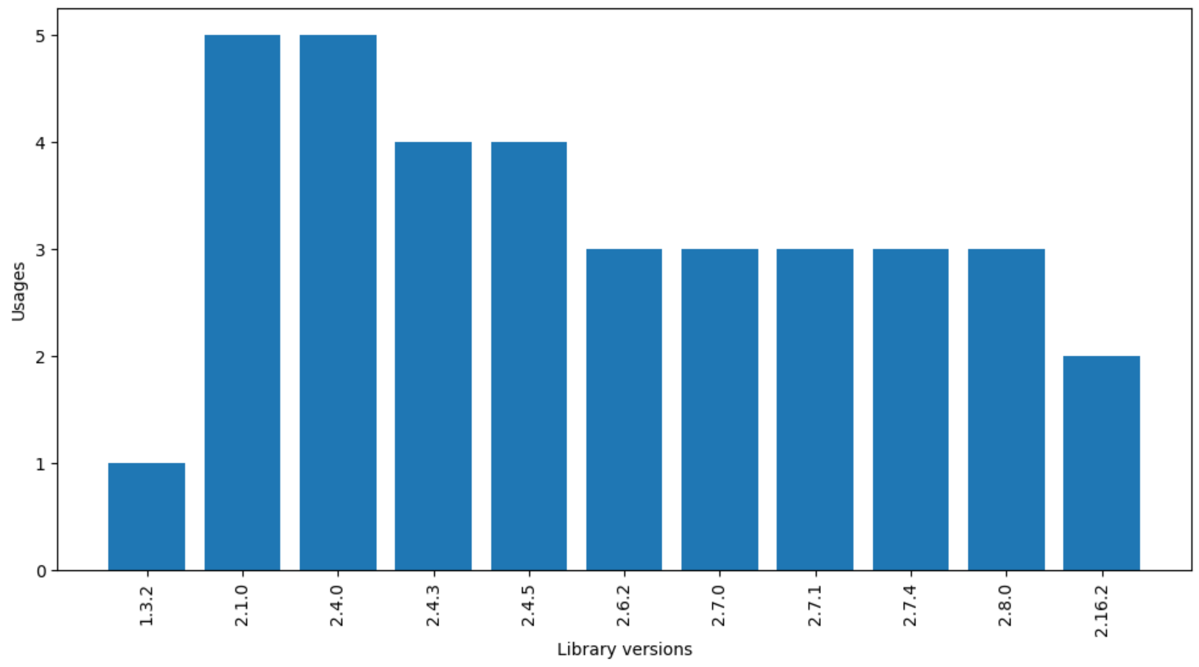

Figure 17. Version usage history for swift-nio-ssl library

[12] https://nvd.nist.gov/vuln/detail/CVE-2019-8849

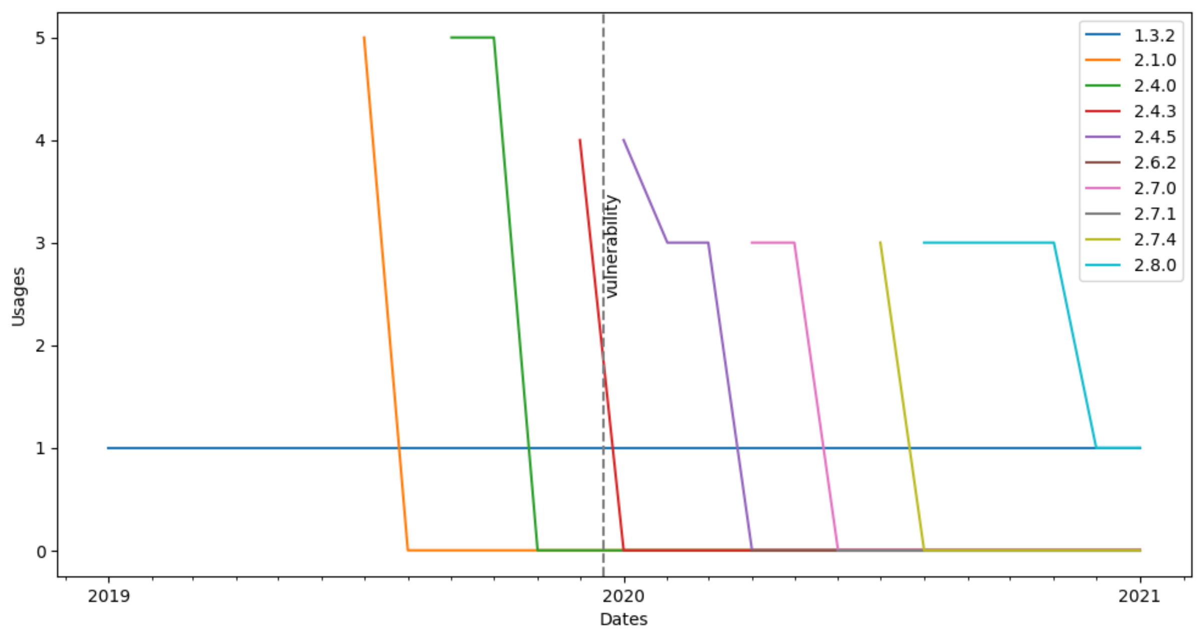

Figure 18. Swift-nio-ssl library usages with vulnerability

6 Discussion

In the following sections, we will discuss the results and compare our findings with the related papers.

6.1 Discussing results

When we brought out RQ1 results in the previous chapter, we found that most software developers do not update their library dependencies. We found that there were around 50% of libraries that have not updated their libraries at all, with only 12% of libraries updating their libraries only once. This would indicate that most developers would use the latest library version that is available until there is no need for updating it. Then we described the results of library downgrades, where we also found that only 0.07% of libraries have ever been downgraded. This is quite normal because developers usually want to have new features and only downgrade when they are not satisfied with the new version. Additionally, we found that most libraries have, on average, about three dependencies, with a minimum count of 1 and a maximum count of 47. As we only looked at libraries that have dependencies, then the minimum count is as expected.

For RQ2, we analysed three popular libraries for Swift applications: alamofire, nimble and afnetworking. There was a similar trend across those three libraries. Which was that when the library version was brought out then, new libraries would start using that new version, and libraries with older versions would stay with the older version. This was brought out by the fact that usage count would stay the same or only slightly decrease when a new version was introduced. One of the reasons could be that as iOS projects are not backwards compatible, then it is always beneficial to use the latest version to support the newer version of iOS environments. Another interesting fact was that there were some instances of cases where developers would create a fork for that new library instead of updating the library. This would allow the developers to make slight changes to existing library code. However, this could also impact the library usage graphs as not all library usages would not be displayed there.

For RQ3, we selected only the three most popular libraries with any vulnerabilities. This would get us one larger library, afnetworking, and two smaller ones: starscream and swift-nio-ssl. We found that there were not any changes to the normal library usage trend after the vulnerability was found. For afnetworking and starscream cases, this could be because a new version was already available, and developers who would usually update were already using the new version. This resulted in no significant vulnerable library version drop. For swift-nio-ssl library, all usages were from the same library fork, and that library diligently updated to the newer version.

6.1 Comparing results with related papers

One related paper was Salza et al. paper “Do Developers Update Third-Party Libraries in Mobile Apps?” which analysed library updates for the Android environment. They found that Android developers rarely update their project dependencies, with 33% of libraries with outdated versions. Additionally, when Android developers update the library, then it would usually be upgraded and there would be only a few cases where the library was downgraded. This sufficiently correlates with our findings. As in RQ1, we found that iOS developers do not update their libraries very often and most version changes are upgrades. Interestingly, they also brought out that Android applications, on average, contain 3.8 dependencies but with a maximum amount of 37 dependencies. This is similar to our dataset because we found that our average dependency count is 2.42 with a maximum dependency count of 47.

They also looked for update patterns in updating Android libraries and found that 63% of libraries are not updated after the first application release, with only 13% of libraries having diligent updates. This is similar to our finding with most of the developers staying with old versions and only a small portion of libraries constantly using the newest dependency version.

We also brought out a Kula et al. paper about “Do developers update their library dependencies?”. in the related section. It analyses how libraries from the maven repository are updated, and it additionally focuses more on vulnerability discovery effects. From there, it was seen that 85.5% of studied systems have outdated libraries. This indicates that developers often do not update libraries and is the same as our findings.

Additionally, they found that developers often do not respond to vulnerability discoveries. This discovery is similar to our RQ3 result, as we saw no significant changes to update trends. In the research paper, it is brought out that most developers do not even know that the library has a vulnerability.

6.2 Threats of validity

The Swift LDN dataset is newly created and has yet to have this kind of analysis done before. When we analysed the results of vulnerability impacts on update patterns, there were not many vulnerabilities found in Swift libraries, and the most vulnerabilities found were from small libraries. Afnetworking library was the only large library where vulnerability was found and it may not represent situations with other large libraries.

Furthermore, we selected only the three most popular libraries and analysed only them. There were similar graphs for two other libraries, but the results were the same. However, if we would select another set of libraries, then the results could be different. To mitigate this, we could have analysed more libraries, but due to time constraints, we were not able to do that as it would increase the workload.

7 Conclusion

The goal of this thesis was to analyse the Swift LDN dataset. We formulated three research questions to help our research and tried to find answers for them.

For the first research question, we wanted to know if developers update their iOS library dependencies. That resulted in us creating two graphs that showed how many libraries have upgraded or downgraded the dependencies. We found that over 50% of the libraries have yet to update their dependencies after the first release, and there were only a few cases where the library was downgraded to a lower version.

After that, we wanted to know if we see any patterns for updating libraries. We analysed the three most popular libraries from the Swift LDN dataset and created version trend graphs for each. That allowed us to see how the latest library version usages change over time. We saw that most of the libraries stay with the same dependency versions, and only a small amount of libraries update their dependencies diligently. This was similar to other research papers that we compared with our results.

Finally, Swift LDN also contained some information about known library vulnerabilities. We found out what vulnerable libraries were most popular and looked up the affected versions. Following that, we got those and subsequent versions and plotted a usage graph for them. The results showed that there were not any significant changes to affected version usage. From the related research paper, we found the same results.

Future work on this thesis could include more in-depth analysis on the second research question. This could mean selecting more libraries to analyse or looking for other patterns inside library version updates. Additionally, libraries could be categorised into different types and checked if some types are more prone to update than others

References

[1] Library (computing). (Sep. 9, 2022). *In Wikipedia*. https://en.wikipedia.org/wiki/Library_(computing) (accessed Oct. 17, 2022)

[2] P. Salza, F. Palomba, D. Di Nucci, C. D'Uva, A. De Lucia and F. Ferrucci, "Do Developers Update Third-Party Libraries in Mobile Apps?," *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, 2018, pp. 255-265, doi: 10.1145/3196321.3196341.

[3] D. Spinellis, "Package Management Systems," in IEEE Software, vol. 29, no. 2, pp. 84-86, March-April 2012, doi: 10.1109/MS.2012.38.

[4] R.G. Kula, D.M. German, A. Ouni, T. Ishio and K. Inoue, "Do developers update their library dependencies?," *Empirical Software Engineering 23*, 2018, pp. 384–417, doi: 10.1007/s10664-017-9521-5.

[5] V. Bauer, L. Heinemann and F. Deissenboeck, "A structured approach to assess third-party library usage," *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 483-492, doi: 10.1109/ICSM.2012.6405311.
D. Pfahl, K. Rahkema, "Dataset: dependency networks of open source libraries available through CocoaPods, Carthage and Swift PM,", Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 393–397, doi: 10.1145/3524842.3528016.

[6] What is a package manager? *Debian manual*. https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html (accessed May 8, 2023)

[7] CocoaPods project repository. *GitHub*. https://github.com/Alamofire/Alamofire (accessed May 9, 2023)

[8] Carthage project repository. *GitHub*. https://github.com/Carthage/Carthage (accessed May 9, 2023)

[9] Swift Package Manager documentation. *Swift*. https://www.swift.org/package-manager/ (accessed May 9, 2023)

[10] Semantic Versioning. https://semver.org/ (accessed May 8, 2023)

[11] What is a graph database?. *Neo4j documentation.* https://neo4j.com/docs/getting-started/get-started-with-neo4j/graph-database/ (accessed May 9, 2023)

[12] Introduction to Cypher. *Neo4j documentation.* https://neo4j.com/docs/getting-started/cypher-intro/ (accessed May 9, 2023)

[13] Alamofire project repository. *GitHub*. https://github.com/Alamofire/Alamofire (accessed May 8, 2023)

[14] Nimble project repository. *GitHub*. https://github.com/Quick/Nimble (accessed May 8, 2023)

[15] Afnetworking project repository. *GitHub.* https://github.com/AFNetworking/AFNetworking (accessed May 9, 2023)

Appendix

I. Neo4j graph queries

Cypher query for optimising query performance:

```
// Creating new relationship between two library nodes
MATCH (l:Library)<-[:IS]-(a:App)-[:CHANGED_TO]->(a2:App)-[:IS]->(l2:Library)
MERGE (l)-[:CHANGED_TO]->(l2);

// Adding indexes for faster lookup
CREATE INDEX app_timestamp_index IF NOT EXISTS FOR (a:App) ON (a.timestamp);
CREATE INDEX app_name_index IF NOT EXISTS FOR (a:App) ON (a.name);

CREATE INDEX library_name_index IF NOT EXISTS FOR (l:Library) ON (l.name);
CREATE INDEX library_version_index IF NOT EXISTS FOR (l:Library) ON (l.version);
```

Cypher query for getting label counts:

```
MATCH (n)
RETURN DISTINCT labels(n) as labels, count(*) as count
ORDER BY labels
```

Cypher query for getting library upgrade counts:

```
MATCH (l:Library)<-[r:DEPENDS_ON]-(a:App)-[:CHANGED_TO]->(a2:App)
OPTIONAL MATCH (a2)-[:DEPENDS_ON]->(l2:Library)
WHERE (l)-[:CHANGED_TO*]->(l2) AND r.from_manifest IS NULL

WITH a.name AS name, count(l2) AS update_count
RETURN name, update_count
ORDER BY update_count
```

Cypher query for getting library downgrade counts:

```
MATCH (l:Library)<-[r:DEPENDS_ON]-(a:App)-[:CHANGED_TO]->(a2:App)
OPTIONAL MATCH (a2)-[:DEPENDS_ON]->(l2:Library)
WHERE (l)<-[:CHANGED_TO*]-(l2) AND r.from_manifest IS NULL

WITH a.name AS name, count(l2) AS downgrade_count
RETURN name, downgrade_count
ORDER BY downgrade_count
```

Cypher query for getting latest dependency count for each library:

```
MATCH (a:App)
WITH a.name as name, a
ORDER BY a.version_number
WITH name, last(collect(a)) as latest

MATCH (latest)-[:DEPENDS_ON]->(l:Library)
WITH name, count(l) as libraries
RETURN name, libraries
```

Cypher query for gettings most popular libraries (popularity by usages count):

```
MATCH (a:App)-[r:DEPENDS_ON]->(l:Library)
WITH l.name AS name, count(DISTINCT a.name) AS usages
RETURN name, usages
ORDER BY usages DESC
```

Cypher query for gettings poplar libraries that also have vulnerabilities:

```
MATCH (:Project)-[:HAS_APP]->(:App)-[:IS]->(l:Library)-
[:HAS_VULNERABILITY]->(v)
OPTIONAL MATCH (a:App)-[:DEPENDS_ON]->(l)
WITH a, l, v
ORDER BY l.version
WITH l.name AS name, count(DISTINCT a.name) AS usages,
count(DISTINCT v) AS vul_count, collect(DISTINCT l.version) AS
vul_versions
RETURN name, usages, vul_count, vul_versions
ORDER BY usages DESC
```

Cypher query for library version usages count:

```
MATCH (a:App)-[r:DEPENDS_ON]->(l:Library)<-[:IS]-(a2:App)
WHERE r.from_manifest IS NULL
WITH l.name AS name, l.version AS version, a2.version_number AS
version_number, a2.time AS time, size(collect(DISTINCT a.name)) AS
usages
RETURN name, version, version_number, usages, time
ORDER BY name, version_number
```

Cypher query for version usage trends over time:

```
MATCH (a:App)-[:IS]->(l:Library {name: $library, version:
$version})
WITH l, date.truncate('month',
datetime({epochSeconds:toInteger(a.timestamp)})) AS start_date,
datetime($end_date) AS end_date
WITH l, [month in range(0, duration.between(start_date,
end_date).months) | start_date + duration({months: month})] AS
dates
UNWIND dates AS date
WITH l, dateTime({year:date.year, month:date.month}) AS time

MATCH (l)<-[r:DEPENDS_ON]-(a:App)
WHERE toInteger(a.timestamp) < time.epochSeconds AND
r.from_manifest IS NULL
OPTIONAL MATCH (a)-[:CHANGED_TO]->(a2:App)
WHERE toInteger(a2.timestamp) < time.epochSeconds
WITH date(time) AS date, a.name AS name, collect(a2 IS NOT null
AND NOT exists((a2)-[:DEPENDS_ON]->(l))) AS uses_different_library
RETURN date, count(
    CASE
    WHEN true IN uses_different_library THEN null
    ELSE name
    END
) AS count
```

Cypher query for vulnerable library versions:

```
MATCH (:Project)-[:HAS_APP]->(:App)-[:IS]->(l:Library)-
[:HAS_VULNERABILITY]->(v)
OPTIONAL MATCH (a:App)-[:DEPENDS_ON]->(l)
WITH a, l, v
ORDER BY l.version
WITH v.id AS id, l.name AS name, v.publishedDate AS date,
count(DISTINCT a.name) AS usages, collect(DISTINCT l.version) AS
vul_versions
RETURN id, name, date, usages, vul_versions
ORDER BY usages DESC, name, date
```

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public