DISSERTATIONES
INFORMATICAE
UNIVERSITATIS
TARTUENSIS
45

KRISTIINA RAHKEMA

Quality Analysis of iOS Applications with Focus on Maintainability and Security Aspects

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

45

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

45

KRISTIINA RAHKEMA

Quality Analysis of iOS Applications with Focus on Maintainability and Security Aspects

UNIVERSITY OF TARTU
Press

Institute of Computer Science, Faculty of Science and Technology, University of Tartu, Estonia.

Dissertation has been accepted for the commencement of the degree of Doctor of Philosophy (PhD) in Computer Science on 12 September, 2023 by the Council of the Institute of Computer Science, University of Tartu.

Supervisor

Prof.	Dietmar Pfahl University of Tartu Estonia

Opponents

Prof. Dr.	Tom Mens University of Mons Belgium
Assoc. Prof. Dr.	Luis Miranda da Cruz Delft University of Technology The Netherlands

The public defense will take place on 23 October, 2023 at 10:15 in Narva mnt. 18-1005.

The publication of this dissertation was financed by the Institute of Computer Science, University of Tartu.

ISSN 2613-5906 (print) ISSN 2806-2345 (PDF)
ISBN 978-9916-27-354-8 (print) ISBN 978-9916-27-355-5 (PDF)

University of Tartu Press
`http://www.tyk.ee/`

To Ottar, Kairi and Noora-Liisa

ABSTRACT

Smartphones have become an inseparable component of our lives. With the popularity of smartphones continuously rising, already today more time is spent on smartphones than on desktop computers. There is an app for everything: messaging, online banking, unlocking the car. We trust these applications with our personal data and expect them to keep this data safe. Unfortunately, many of these applications are discovered to be insecure. Even when only looking at some very popular mobile applications, such as Facebook, TikTok, and WhatsApp, many vulnerabilities have been discovered over the last few years. The severity of such vulnerabilities can range from information disclosure and account high jacking to remote code execution. Due to the nature of how we use our smartphones and carry them with us everywhere, these severe vulnerabilities jeopardize our most private data. Security is one dimension of software quality. Another important, but less visible aspect is maintainability. Low code quality can lead to high maintenance costs and decrease the budget for feature development. It is therefore important to ensure that developers have sufficient tool support to build high-quality mobile applications.

Due to its more open nature, plentiful open-source Android applications exist that can be used to conduct research. Many studies have been conducted on Android applications analyzing different aspects of code quality such as maintainability and security. Similarly, useful tools have been introduced by researchers. Unfortunately, very little tool support and almost no research exist on iOS applications. Given that iOS is the second most popular mobile operating system, it is important to support developers in building quality applications both in regard to security and maintainability for iOS.

The goal of this thesis is to improve tool support for both developers and researchers and to fill some of the research gaps for maintainability and security analyses for iOS apps. First, we developed GraphifySwift, a tool that detects code smells in projects written in Swift. We then applied GraphifySwift to open-source iOS applications and analyzed the distribution and frequency of code smells. Additionally, we used GraphifySwift and PAPRIKA to compare code smells in iOS and Android applications.

We analyzed iOS applications for 34 object-oriented code smells and compared the occurrence of 19 object-oriented code smells in iOS and Android. We found that iOS applications tend to contain more code smells related to small and data classes whereas Android applications contain more code smells related to complex and large classes.

Based on the experience gained from developing and using GraphifySwift we decided to substantially increase the capability of our tool suite and developed GraphifyEvolution, an extended version of GraphifySwift. GraphfiyEvolution is an extendable tool that can analyze both snapshots and the evolution of projects. We used GraphifyEvolution for a preliminary code smell evolution analysis.

By implementing additional external analyzers it is possible to extend GraphifyEvolution with more analysis capabilities. We implemented SwiftDependencyChekcer, a tool that extracts information on third-party library dependencies from iOS apps and detects vulnerable dependencies. We implemented an external analyzer based on SwiftDependencyChecker for GraphifyEvolution and used it to build a library dependency network (LDN) dataset for third-party libraries in the Swift ecosystem encompassing libraries available through the three package managers used in iOS development: CocoaPods, Carthage, and Swift Package Manager. We used this dataset to study different aspects of the Swift LDN. We analyzed the overall evolution of the Swift LDN, the use of package managers, technical lag in library dependencies, and the spread of vulnerabilities in the LDN.

We found that the Swift LDN is growing in terms of both the number of libraries and the number of library versions. CocoaPods is the most popular package manager, Carthage has stopped growing and Swift PM is becoming more and more popular over time. The technical lag for library dependencies is growing. It is higher when more restrictive dependency requirement types are used by developers. Lastly, the percentage of libraries with vulnerable library dependencies is lower than in other ecosystems which can be explained by an overall lower number of transitive dependencies.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Acronyms

API Application Programming Interface. 53, 55, 57

CPE Common Platform Enumeration. 53, 55–57, 78

JSON JavaScript Object Notation. 42, 48, 50, 55, 56

LDN Library dependency Network. 19–21, 33–36, 51, 57, 74, 75, 79, 80, 84, 85, 89, 94, 110, 111, 114, 116, 117

NVD National Vulnerability Database. 29, 49, 51, 53, 55, 56, 78, 88, 115, 117

OWASP Open Web Application Security Project. 16, 37, 56

PM Package manager. 14, 98–100

Swift PM Swift Package Manager. 12, 13, 20, 26, 27, 75, 77–80, 85, 89, 92, 94, 95, 97–103, 110–113, 117

URL Uniform Resource Locator. 48, 53, 75, 77, 78

1. INTRODUCTION

Modern life is difficult to imagine without smartphones. The popularity of smartphones is continuously rising, and already today, more time is spent on smartphones than on desktop computers [Mee18]. We use mobile applications for almost everything, from messaging over online banking to unlocking the car. We trust these applications with our personal data and expect them to keep this data safe. Unfortunately, many of these applications turn out to be unsafe.

Even in very popular mobile applications, such as Facebook, TikTok, and WhatsApp, many vulnerabilities have been discovered over the last years [Abd20; Tea22; Wha19]. The severity of such vulnerabilities ranges from information disclosure and account high-jacking to remote code execution. Due to the nature of how we use our smartphones and carry them with us everywhere, these severe vulnerabilities jeopardize our most private data. Therefore, it is crucial to ensure developers have sufficient support and tools to build high-quality mobile applications.

Code quality has many dimensions that could be measured, such as, among others, maintainability and security. One possible way to measure the maintainability of a code base is to search for code smells. Code smells are bad practices such as too long methods or too complex classes that make code less readable, and testable and decrease code maintainability [KDG09; Olb+09].

The security of an application can be tested in different ways. An app consists mainly of three types of code: custom code, system libraries, and third-party libraries. Custom code is the code that is written for a specific application. This is the part of the code a developer has the most influence and responsibility over. Custom app code can be tested by using static and dynamic code analysis tools or even performing manual testing. Dukes et al. [DYA13] showed that different types of security testing complement each other and are needed to discover as many vulnerabilities as possible. They found that the largest number of security vulnerabilities was found through static code analysis.

Next to custom code, an app often relies on libraries that have already implemented solutions to common problems. There are two types of libraries: system libraries which are provided by the programming language or operating system and third-party libraries. Some programming languages, such as JavaScript have very small system libraries forcing developers to rely more heavily on third-party libraries. Other programming languages, such as Swift, provide extensive system libraries, allowing developers to rely less on third-party solutions. Using third-party libraries to speed up development is a common practice. The Open Web Application Security Project (OWASP), for example, strongly recommends using existing cryptography libraries instead of implementing a custom algorithm [OWA16]. Nevertheless, vulnerabilities can be found in even popular and well-tested third-party libraries. For example, in 2015 vulnerabilities were discovered in the popular networking library AFNetworking that affected 25,000 applications on the Apple App Store [Kum15].

To ensure that an application does not depend on vulnerable third-party libraries it is necessary to either check for openly reported vulnerabilities manually or rely on tools that notify developers when a vulnerability was discovered in a library dependency so that the library dependency can be upgraded to a patched version. Unfortunately, such tools do not exist for every package management system that is used to include library dependencies.

1.1. Problem Statement

Most smartphones run either Android or iOS operating systems, with iOS being the second most popular mobile operating system. In 2022, most of the global market share (72%) was held by Android, while iOS held 27% [Sta23] with the exception of some regions (e.g. the USA) where the majority of users preferred iOS [Sta22]. At the same time, overall earnings on the iOS App Store are considerably larger than on the Google Play Store [Per18]. This makes both Android and iOS very attractive targets for developers. It is, therefore, important to support developers in building quality applications with regard to both security and maintainability not only for Android but also for iOS. Currently, tool support for iOS is lacking behind Android. Swift, the official language for iOS development, is relatively new and the iOS/macOS ecosystem is seen as rather exotic.

Due to its more open nature, plentiful open-source Android applications exist that can be used to conduct research [All+16]. Additionally, closed-source Android applications are much easier to analyze than closed-source iOS applications. Many studies have been conducted on Android applications analyzing different aspects of code quality such as maintainability [GJW14; Hec+15a; Hec+15b] and security [GGN17; Enc+11; Ngu+20]. Useful tools have been introduced by researchers [Ngu+20; Hec15; Pal+17] and industry [ash23; OWA23] to help developers improve the quality of Android applications.

Unfortunately, very little tooling support and almost no research exist on iOS applications. Our goal is to improve tool support for both developers and researchers that wish to analyze code quality in iOS applications written in Swift.

1.2. Research Approach

Our research approach aims at providing tools that support the analysis of projects developed in Swift, targeting both developers and researchers. Developers need tooling that helps improve the maintainability and security of their projects, which requires analyzing a snapshot of a single project. Researchers, on the other hand, need to analyze a large set of projects at a time, possibly taking into account the evolution of the project.

Once available, we use these tools to demonstrate their usefulness by conducting a large-scale analysis of apps and third-party libraries in the iOS/macOS

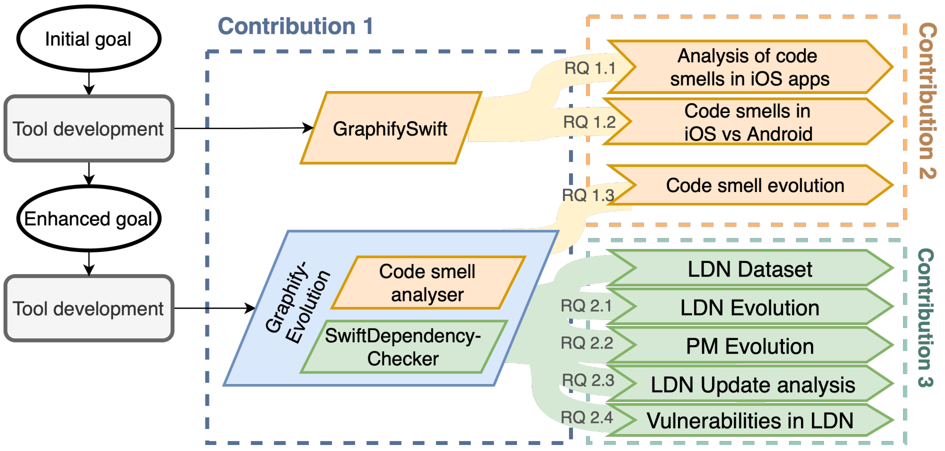


Figure 1. Research approach

ecosystem. An iOS application contains three types of code: custom code, third-party libraries, and system libraries. We apply our tools to analyze the maintainability of custom code by analyzing code smells in open-source iOS libraries. We then analyze the maintainability and security of library dependencies by conducting extensive analyses on the library dependency network in the iOS ecosystem.

An overview of the research approach is given in Figure 1.

The initial goal requiring tool development was to provide support for maintainability analysis by detecting code smells in Swift projects. We developed GraphifySwift, an analysis tool for the Swift language that analyses the project source code, enters relevant data on classes, methods, and variables to a Neo4J database, and then finds code smells defined as database queries. GraphifySwift allows for the analysis of code smells in Swift applications. We showed how GraphfifySwift can be used to analyze open-source iOS applications. We compared the prevalence and distribution of code smells in iOS applications and Android applications and answered the following research questions:

- RQ1.1: What is the distribution and frequency of code smells in iOS applications?
- RQ1.2: How do code smells in iOS applications compare to Android?

Based on the experience gained from using GraphifySwift to analyze open-source iOS applications, we revised the initial development goal. GraphifySwift is built on snapshot analysis of code smells. The revised development goal became to provide tooling that can be extended with additional analysis types and that allows the analysis of the evolution of a project in addition to snapshot analysis. These additional qualities make the tooling more valuable to researchers.

We extended GraphifySwift and developed a modular and extendable tool called GraphifyEvolution that allows analysis of both a snapshot and the evolution of projects. The tool is extendable with external analyzers that can easily be plugged into the analysis flow. This approach makes it possible to extend the tool's capabilities with other existing tools. All analysis results are stored in a neo4j database, making it possible to later detect patterns by querying the database directly. Furthermore, we developed two external analyzers for GraphifyEvolution: a code smell analyser and a library dependency checker. The second external analyzer can be used as a stand-alone tool. It is called SwiftDependencyChecker.

We showed how GraphifyEvolution and the code smell analyzer can be used to analyze the evolution of code smells in open-source iOS applications, answering the following research question:

- RQ1.3: How have code smells in iOS applications evolved over time?

After showing how GraphifyEvolution can be used to analyze the custom code of an application, we wanted to highlight its capabilities by additionally analyzing maintainability and security issues related to third-party libraries used in iOS applications. We started by using GraphifyEvolution and SwiftDependencyChecker to build a dataset of third-party library dependencies of the Swift ecosystem. We analyzed this dataset regarding different aspects. We first analyzed the Swift Library dependency Network (LDN) evolution to better understand the ecosystem and to put it into the context of other ecosystems. We analyzed how the use of package managers has evolved in the ecosystem and discussed how different properties of these package managers influence their acceptance. We then investigated which aspects affect the dependency lag in the ecosystem, specifically analyzing the relationship between version requirement types and library dependency updates. Lastly, we analyzed how far vulnerabilities spread in the Swift LDN to better understand security risks stemming from third-party libraries and the potential need for tool support. We answer the following research questions:

- RQ2.1: How has the Swift LDN evolved?
- RQ2.2: How has the package manager use evolved in the Swift LND?
- RQ2.3: Do version requirement types influence library dependency updates?
- RQ2.4: How far do vulnerabilities spread in the Swift LDN?

1.3. Contributions of the Thesis

In this thesis, we present three main contributions consisting of tool support, analysis of code smells, and analysis of the Swift library dependency network.

- **Contribution 1**: Tool support for analyzing code quality in iOS applications
- **Contribution 2**: Empirical evidence on code smells in open-source iOS applications
- **Contribution 3**: Extensive analysis of library dependency networks in the Swift ecosystem

Figure 1 illustrates the three contributions as dotted boxes. Contribution 1 is the development of tool support, Contribution 2 covers the analysis of RQ1, and

Contribution 3 covers the analysis of RQ2. The following subsections explain these contributions in more detail.

1.3.1. Tool Support for Analysing Code Quality in iOS Applications

We provide tool support for analyzing applications written in Swift. Additionally, applications written in Java and C++ are supported. GraphifyEvolution is a modular and easily extendable tool that allows a range of different kinds of analysis from a snapshot of a single project to the evolution of many projects. We show how GraphifyEvolution can be extended with external analyzers by implementing code smell analysis of iOS applications and library dependency analysis for the three package managers used in iOS development: CocoaPods, Carthage, and Swift Package Manager. The main target group for GraphifyEvolution is researchers who wish to analyze a large set of applications, but the tool can also be used by developers who wish to analyze the code quality of their projects. The output of GraphifyEvolution can be used for different purposes, for example, data analysis or visualization. Multiple prototypes have been developed by master students that visualize the code quality of projects analyzed with GraphifyEvolution.

The standalone tool SwiftDependencyChecker which is used as an external analyzer with GraphifyEvolution can be used to detect vulnerable dependencies in iOS applications. It is a lightweight tool that can be integrated into the Xcode build process and it allows developers to detect vulnerable library dependencies in their applications.

All tools[1][2][3] are open source and written in Swift.

1.3.2. Empirical Study on Code Smells in Open Source iOS Applications

We analyze 273 open-source iOS applications and report the variety and density of 34 object-oriented code smells found. Using PAPRIKA [Hec15] we also compare the distribution of code smells in 273 iOS applications and 694 Android applications. We present the first study on iOS code smells that reports data on this many code smells.

1.3.3. Extensive Analysis of Library Dependency Networks in the Swift Ecosystem

Using GraphifyEvolution and SwiftDependencyChecker we create a LDN dataset for the Swift ecosystem encompassing libraries used through CocoaPods, Carthage, and Swift PM. Based on this dataset we conduct several studies analyzing the Swift LDN:

1. We analyze the evolution of the Swift LDN.

[1]https://github.com/kristiinara/GraphifySwift
[2]https://github.com/kristiinara/graphifyevolution
[3]https://github.com/kristiinara/swiftdependencychecker

2. We analyze the evolution of the package managers in the Swift LDN
3. We analyze the relationship between updating practices and library dependency requirements.
4. We analyze how vulnerabilities spread in the Swift LDN.

With our research, we provide the first insights into the Swift LDN. Additionally, using the Swift ecosystem as an example ecosystem, we analyze how the introduction of new package managers affects the LDN evolution.

1.4. Structure of the Thesis

The rest of the thesis is structured as follows.

In **Chapter 2** we provide background on the iOS/macOS ecosystem, on the development of iOS applications, and different types of code quality analyses.

In **Chapter 3** we summarise related work on code smells and LDN analysis and outline the research gap.

In **Chapter 4** we describe the developed tools GraphifySwift, GraphifyEvolution, and SwiftDependencyChecker covering Contribution 1. This chapter is based on work from [RP20b; RP21; RP22c]. This work was published in the proceedings of the IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft'20, MobileSoft'21, and MobilSoft'22).

In **Chapter 5** we report results on research questions RQ1.1, RQ1.2, and RQ1.3 covering Contribution 2. This chapter is based on work from [RP20b; RP20a; RP21]. This work was published in the proceedings of the IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft'20, MobileSoft'21) and in the proceedings of the 8th International Workshop on Quantitative Approaches to Software Quality (QuASoQ'20).

In **Chapter 6** we report results on research questions RQ2.1, RQ2.2, RQ2.3, RQ2.4 and describe the construction of the LDN dataset covering Contribution 3. This chapter is based on work from [RP22b; RP23a; RP22a; RP23b]. This work was published in the proceedings of the IEEE/ACM International Conference on Mobile Software Engineering and Systems (MobileSoft'23), in the proceedings of the 19th International Conference on Mining Software Repositories (MSR'22), and in the proceedings of the 23rd International Conference on Product-Focused Software Process Improvement (PROFES'22).

In **Chapter 7** we conclude the thesis and provide ideas for future research.

2. BACKGROUND

In the following, we briefly describe the macOS/iOS ecosystem. Apple has created a fairly unique ecosystem where both hardware and the operating systems running on the hardware are developed by the same company. Considering the wide selection of devices from personal computers to smartphones and smartwatches this allows the optimization of hardware and software and the interoperability between these devices. The closedness of the ecosystem makes it an interesting subject to study as it minimizes the potential effects of other ecosystems. Additionally to macOS and iOS the ecosystem also incorporates the less known operating systems iPadOS, watchOS and tvOS.

Figure 2 gives an overview of some of the main highlights of the macOS/iOS ecosystem over the years. In 2001, the first Mac OS X version (10.0) was released. The X in the name was supposed to indicate that the operating system was a Unix system. The Macintosh computers built by Apple at the time were based on PowerPC. In 2003, Apple released the first Xcode version to improve the development of Mac OS X applications. The official language to develop these applications was Objective-C, a language built on C with the purpose of making it object-oriented. In 2007 Apple released the first iPhone and its operating system iPhone OS 1. A year later the App Store was launched. The red area in Figure 2 shows the growth of the App Store as the number of iOS applications in millions [Cur23]. In 2009 Apple switched the Macintosh processors from PowerPC to Intel. The Mac App Store was launched two years later. Mac OS X was renamed as OS X in 2012 with the release of OS X 10.8. Later, in 2016, it was renamed again to macOS with the release of macOS 10.12. The iPad was launched in 2010, this coincided with the time when the number of iOS apps on the App Store started rapidly growing, reaching 1 million in 2013. In 2012, iPhones gained 64-bit support, and the support of 32-bit apps was finally dropped in 2017. Apple introduced Swift, a new and official programming language, in 2014. The first couple of years saw big changes in the language design. With the release of Swift 3.0 in 2016 the language stabilized. The use of the Swift programming language was further eased by the introduction of SwiftUI, a UI framework that was not reliant on the Objective-C heavy Cocoa Touch. In 2020 Apple released Mac computers equipped with the new Apple silicon chips, bringing the macOS architecture from Intel to ARM.

Applications for iOS are natively written in either Objective-C, Swift, or a combination of the two languages. Since Objective-C is an extension of C, using C and C++ from Objective-C code is fairly straightforward. In addition to native app development cross-platform app development and hybrid app development exist. With cross-platform app development, developers use frameworks such as Xamarin, React Native, or Flutter to build apps that can be compiled for both iOS and Android. The app code written in C# (for Xamarin), JavaScript (for React Native), and Dart (for Flutter) is compiled into native application packages. In hybrid app development, a web application is developed that is packaged inside an

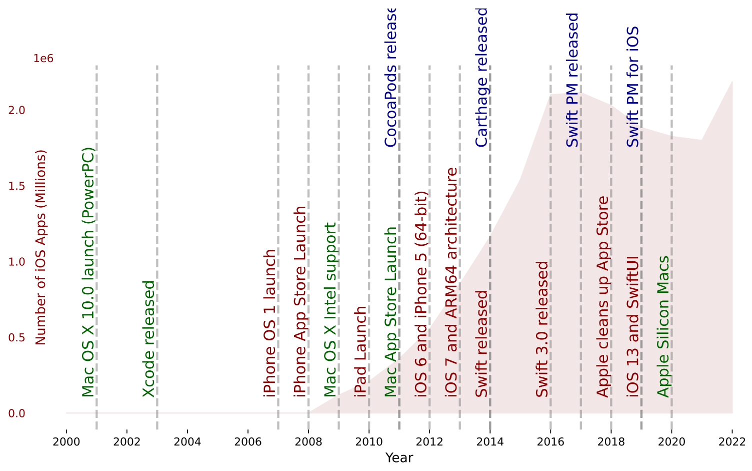


Figure 2. Timeline of the macOS/iOS ecosystem

application that displays the web application in a web view. This approach makes it rather easy to convert web pages into mobile applications. Cross-platform and hybrid app development carry many advantages in regard to targeting multiple platforms, but they may not provide the same level of performance or functionality as native apps. The remainder of this thesis handles the native development of iOS applications written in the officially recommended language Swift. Applications typically consist of three types of code:

1. Custom code that is written for a specific application.
2. System libraries that are provided by Apple.
3. Third-party libraries that are developed by other developers and can be included through package managers.

Figure 3 illustrates the division of an application into code types. In the following subsections, we describe custom code, system libraries, and third-party libraries in the iOS context.

2.1. Custom Code

Custom code in native iOS applications is mostly written in either Objective-C or Swift. New projects mostly default to Swift as it is the official recommended language for iOS development. Native iOS applications are typically written in Xcode, the official IDE for iOS and macOS development. An Xcode project contains the source files and build settings for an application. It also handles code signing.

A minimal iOS application might include the AppDelegate, SceneDelegate, and ViewController classes. The AppDelegate class handles the life cycle of the application and the SceneDelegate class handles the life cycle of different windows. The ViewController class handles a single view, defining its UI and han-

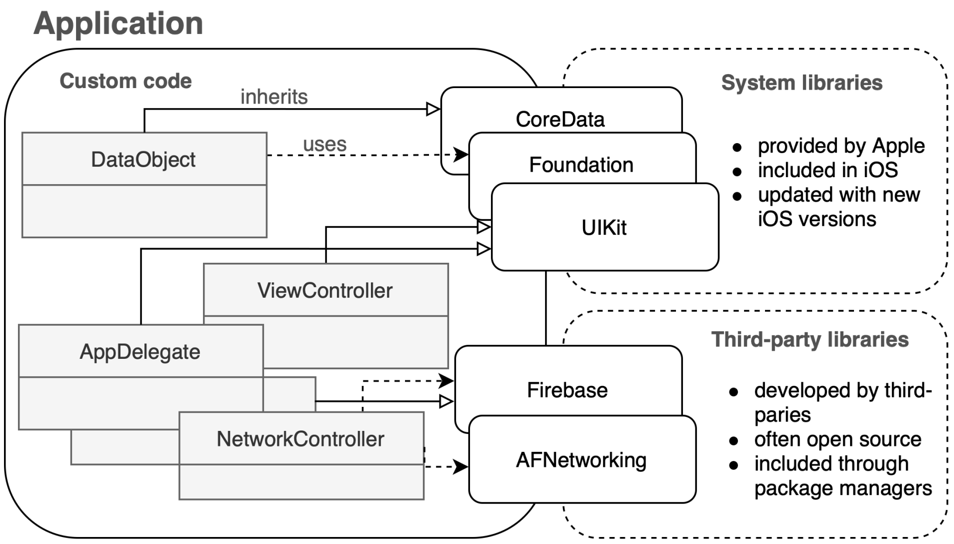


Figure 3. Application code structure

dling user events. However, most applications contain far more classes than the minimal example. The architecture for an iOS app recommended by Apple is Model-View-Controller (MVC), allowing the separation of an application's data (model), user interface (view), and control flow (controller).

2.1.1. Maintainability

The use of good architecture patterns increases the maintainability of a project. Code smells are recurring patterns in code that have been identified as bad practices [Fow18]. They can lead to technical debt and long-term maintainability problems [Fow18]. Code smells and their implications have been studied a lot for desktop applications, especially Java desktop applications [KDG09; Moh+09; Olb+09]. Code smells can lead to more change-prone [KDG09] and fault-prone [Lin+14] code increasing the maintainability effort [Olb+09]. Fowler defined 22 object-oriented code smells in his book "Refactoring: Improving the Design of Existing Code" [Fow18] and many more code smells types have been introduced ever since.

Many tools exist that detect code smells in Java code [Pai+17]. A few tools (e.g. SonarQube[1], SwiftLint[2] and Codebeat[3]) exist that can analyse Swift code. However, there are no tools that would cover all 22 object-oriented code smells introduced by Fowler[Fow18]. Some of the Fowler object-oriented code smells include Long Method, Long Parameter List, Feature Envy, and Speculative Generality. The full list of code smells with their definitions is given in A.2.

[1]`https://rules.sonarsource.com/swift/type/Code%20Smell`
[2]`https://github.com/realm/SwiftLint`
[3]`https://github.com/marketplace/codebeat`

2.1.2. Security

Another important quality aspect of custom application code is security. By testing the security of mobile applications, potential vulnerabilities can be identified and mitigated that could otherwise result in data breaches, unauthorized access, and other security risks. OWASP has provided a list of source code analysis tools that can be used to find potential security vulnerabilities [OWA23]. Multiple tools exist that can also be applied to source code written in Swift. Most of these tools rely on patterns in code that are deemed to be potentially unsafe. For example, Insider CLI marks any use of the function `withUnsafeBytes()`[4] in Swift as unsafe. Such an approach highlights potentially unsafe patterns for developers, but can often lead to false positives when the use of the potentially unsafe function is intended. Other approaches to security testing exist, such as manual security testing and dynamic testing, but they are not contingent on source code access.

2.2. System Libraries

System libraries provide access to the underlying operating system and device capabilities through system APIs. The iOS ecosystem provides a very rich set of system libraries providing developers with access to a wide range of features and functionality specific to iOS and macOS platforms. Some of the more important libraries are listed below:

- Foundation[5] provides fundamental classes and utilities for many essential data types such as collections, dates, and files.
- UIKit[6] contains the core components for building the user interface of an iOS application, including buttons, labels, and views.
- CoreData[7] provides a persistence framework for managing data.
- AVFoundation[8] provides a framework for working with audio and video media.
- Core ML[9] allows the integration of machine learning models into iOS applications.

In comparison to some other languages, for example, JavaScript, the system libraries available for Swift, especially when developing iOS or macOS applications, are more extensive, allowing developers to rely less on third-party libraries.

[4]https://github.com/insidersec/insider/blob/master/rule/ios.go

[5]`https://developer.apple.com/documentation/foundation`

[6]`https://developer.apple.com/documentation/uikit`

[7]`https://developer.apple.com/documentation/coredata`

[8]`https://developer.apple.com/documentation/avfoundation/`

[9]`https://developer.apple.com/documentation/coreml`

2.3. Third-Party Libraries

In addition to system libraries that are developed by Apple, developers can include third-party libraries in their applications. Third-party libraries are libraries that are developed by someone other than Apple or the application developer themselves. Using third-party libraries is a common practice in software engineering and allows the reuse of existing solutions to common problems. This can make the development process faster and easier.

It can, however, be tedious to maintain multiple dependencies manually. Automated solutions make this process easier, therefore, package managers have been created where the developer simply states the library name and exact version or version requirements. The package manager takes care of downloading and installing the suitable library version.

2.3.1. Package Managers

There are three package managers in the iOS ecosystem: CocoaPods, Carthage, and Swift Package Manager (Swift PM).

CocoaPods[10] was released in September 2011 and is the oldest package manager with around 88 thousand libraries. CocoaPods is a centralized package manager. Dependencies are declared in Podfile. When CocoaPods is executed it downloads and compiles libraries declared in Podfile. It generates a new Xcode Workspace that has all libraries included. This makes CocoaPods very easy to use, as there is no additional manual work needed. Information on all libraries and library versions is uploaded to the central Specs repository[11]. This means that it is possible to extract information for all packages that have ever been available through CocoaPods. Resolved dependencies are listed in Podfile.lock under "PODS:" and are given in the following format:

```
PODS:
- <libraryName> (<exactVersion>)
    - <transitiveDependency> (= <version>)
    - <transitiveDependency> (<exactVersion>)

DEPENDENCIES:
    - <libraryName> (>= <version>)
```

Carthage[12] was released in November 2014. According to Libraries.io, it includes 4.5 thousand libraries [Lib22]. This number, however, is an estimate as Carthage is a decentralized package manager and no official central repository

[10] https://cocoapods.org
[11] https://github.com/CocoaPods/Specs
[12] https://github.com/Carthage/Carthage

of libraries exists. Carthage was created as a counterweight to the more heavyweight CocoaPods. Libraries can be included through Carthage by simply adding a repository address of a library to the Cartfile. Carthage downloads and compiles these libraries but does not automatically include them in the app projects. Manual work on adding the library to the app project is still needed. This makes using Carthage slightly more complicated than CocoaPods, but it is also a lot more lightweight and developers are not forced to use a generated app project. No official list of repositories exists for Carthage. Libraries.io [Lib22] provides a list of 4498 Carthage libraries that are extracted from Cartfiles hosted on GitHub. With Carthage, dependencies are declared in a Cartfile. The dependencies are specified by giving the type of source, name of library, and the version requirement, for example:

```
github "<userName/projectName>" "<version>"
git "<repoAddress>" >= "<version>"
```

Swift Package Manager (Swift PM)[13] was released in December 2017. It is the official package manager created by Apple. Swift PM is a decentralized package manager like Carthage. Differently to the other two package managers Swift PM can also be used to create Swift packages that can be both libraries or applications. This means that Swift PM can be used for example to create a new command line application. Support for iOS applications was not added to Swift PM until 2019 [Ell20]. Since 2019 it is also possible to use Swift PM directly through Xcode (the main IDE for iOS and macOS development). Swift PM has no official centralized list of repositories. There are multiple repositories containing information on Swift PM libraries. Libraries.io contains 4,207 libraries, swiftpackageregisty[14] contains 4,348 libraries and Swiftpack.co[15] contains 12,143 Swift packages. Packages on Swiftpack.co, however, do not seem to be all libraries. With Swift PM dependencies are declared in Package.swift files. The resolution file Package.resolved is a JSON file containing information on all resolved dependencies, both direct and transitive

```
{
    "pins" : [
        {
            "identity" : "<libraryName>",
            "kind" : "remoteSourceControl",
            "location" : "<repoAddress>",
            "state" : {
```

[13]https://www.swift.org/package-manager/
[14]https://swiftpackageregistry.com
[15]https://swiftpack.co/

```
                    "revision" : "<revisionHash>",
                    "version" : "<exactVersion>"
                }
            }
        ]
    }
```

2.3.2. Maintainability

For better maintainability, library developers are encouraged to implement semantic versioning schemes[16] where each version number consists of three components: major, minor, and patch. The patch version number is supposed to change if only bug fixes are implemented with the change that does not alter how the library works. The minor version number is supposed to change if new features are added, but the library remains backward compatible. If breaking changes to the library API are introduced then the major version number should change. The version number should then be given as <major>.<minor>.<patch>. Such a versioning scheme allows developers that use these libraries to quickly assess the effort needed for upgrading the library version.

CocoaPods, Carthage, and Swift PM support different types of dependency version requirements. Generally, there are eight different kinds of version requirements: `latest`, ==, ~>, >=, >, <=, < and `..<`. The version requirement types `latest`, >= and > behave similarly, by requiring the latest possible version with the only difference that `latest` does not define a minimum version. The version requirement types <=, < provide an upper bound for the version number, while `..<` provides both an upper bound, as well as, a lower bound for the version number. The version requirement ~> behaves similarly to `..<` where the version following ~> is the lower bound and the next minor or major version is the upper bound for the version number. Each package manager has slightly different ways of declaring the version requirements. Table 1 lists how the dependency requirement types were unified in this thesis. A blank entry signifies, that this version requirement type is not supported by the package manager. Entry "empty" means that this version requirement type is used if no version requirement is listed in the manifest file.

For example when using the library version requirement ~>`1.2` in CocoaPods or Carthage, bug fixes introduced in `1.2.1` and `1.2.2` would be included by the package manager automatically. However, a change of the minor version to `1.3` would require a developer to manually upgrade the version in the package manager manifest file. Similarly the library version requirement ~>`1` would allow updates up to, but excluding version `2.0`.

One of the most popular version requirement type for many package managers

[16]`https://semver.org`

Table 1. Unification of version requirement types

Requirement type	CocoaPods	Carthage	Swift PM
`latest`	empty	empty	
`==`	`=`	`==`	`exact`
∼>	∼> `x.y`	∼> `x.y`	`upToNextMajor`
∼>	∼> `x.y.y`	∼> `x.y.z`	`upToNextMinor`
`==`	`branch`	`branch`	`branch`
`==`	`tag`	`version`	`revision`
`>=`	`>=`	`>=`	`from`
`>`	`>`		
`<=`	`<=`		
`<`	`<`		
`..<`			`..<`

is the ˆ requirement type. This version requirement type does not exist in the exact same way in the Swift ecosystem, but is included by the ∼> type. With the latter it depends on how the version is given, either as `x.y` or `x.y.z`, if it is equivalent to the popular ∼> type.

2.3.3. Security

Third-party solutions are often better vetted than custom solutions. The Open Web Application Security Project (OWASP), for example, strongly recommends against the use of custom encryption algorithms [OWA16]. Nevertheless, vulnerabilities can be found in even very popular and well-tested libraries. For example, in December 2021, a security vulnerability was discovered in the widely used Log4J Java logging library. This vulnerability affected 4% of all Java applications [WR21] and made them vulnerable to remote code execution attacks.

When a vulnerability in an open source library or a popular software application is discovered the product vendor or vulnerability researchers that found the vulnerability can submit the vulnerability to a CVE Numbering Authority (CNA), who will then assign a Common Vulnerabilities and Exposures identifier (CVE) to the vulnerability. The CVE identifier is then used to uniquely identify the vulnerability. Once a CVE is assigned to the vulnerability, information about the vulnerability is added to vulnerability databases, such as the National Vulnerability Database[17] (National Vulnerability Database (NVD)). The entry for the aforementioned Log4J[18] vulnerability, for example, includes a short description and numerous links to third-party advisories, patches and exploits.

Developers can search these vulnerability databases to check if the third-party library versions they include in their applications contain publicly reported vulnerabilities. Tools, such as GitHub Dependabot[19], have been developed that allow

[17] `https://nvd.nist.gov`
[18] `https://nvd.nist.gov/vuln/detail/CVE-2021-44228`
[19] `https://github.com/dependabot`

the automatic checking of vulnerable dependencies. However, there are no tools that support all three package managers used in iOS development.

3. RELATED WORK

This chapter describes related work on code smells, library dependency network evolution, technical lag in library dependencies, and on vulnerability analysis in library dependency networks.

3.1. Code Smells

Fowler [Fow18] defined 22 object-oriented code smells and provided refactorings for these code smells. Khomh et al. [KDG09] studied the impact of code smells. They found that code smells affect classes negatively and that classes with more code smells were more prone to changes [KDG09]. Olbrich et al. [Olb+09] studied the evolution and impact of code smells based on two open-source systems. Their findings confirmed that code smells negatively affect the way how code changes. They were also able to identify different phases of evolution in code smells [Olb+09]. Linares et al. [Lin+14] made a large-scale analysis of Java Mobile applications and discovered that anti-patterns negatively impact software quality metrics such as fault-proneness [Lin+14]. Tufano et al. [Tuf+15] studied the change history of 200 open-source projects and found that most code smells are introduced when the corresponding code is created and not when it is changed. They also found that when code does become smelly through evolution then it can be characterized by specific code metrics. Contrary to common belief [Sha19] they discovered that most code smells are not introduced by newcomers but by developers with high workloads and high release pressure [Tuf+15].

Different kinds of code smells have been studied for Android, such as object-oriented, Android-specific, security-related, and energy-related code smells. Gottschalk et al. proposed an approach to detect energy-related code smells on mobile applications and validated this approach on Android and showed that it is possible to reduce energy consumption by refactoring the code [GJW14]. Cruz et al. analysed how performance based guidelines affect energy efficiency [CA17] and how energy efficiency can be increased by automatic refactoring of energy-related code smells [CA19]. Ghafari et al. [GGN17] studied security-related code smells. They discovered that most applications contain at least some security-related code smells. Reis et al. [RAC21] investigated how security patches affect the maintainability of the related code. They found that security fixed tend to increase code complexity suggesting that extra care should be taken when implementing such fixes.

Hecht [Hec15] proposed an approach to detect code smells and anti-patterns on Android systems and implemented this approach in a tool called PAPRIKA. This tool analyses the Android APK, creates a model of the code, and inserts this model into the Neo4J database. Code smells are then defined as database queries which makes it possible to query code smells on a large number of applications at the same time. He analyzed 15 popular applications for the occurrences of four object-oriented and three Android code smells. Hecht et al. [Hec+15b] tracked the software quality of 106 popular Android applications downloaded from the Google Play Store along their evolution. They calculated software quality scores for different versions of these applications and tracked their evolution. There were different evolution graphs, such as constant decline, constant rise, stability, or sudden change in either direction depending on the programming practices of the team [Hec+15b]. This shows that code quality is not necessarily linked to app size but to the programming practices of the developers. Mateus et al. [MM18] used PAPRIKA to analyze Android applications written in Java and Kotlin. They analyzed a set of 2167 open-source Android applications combining different databases of open-source Android applications and compared code smell occurrences in both languages. They concluded that applications that were initially written in Java and later introduced Kotlin were of better quality than other Android applications [MM18].

In these papers using PAPRIKA, the number of code smells studied was limited due to the number of code smells PAPRIKA is able to detect and ranged from three to four object-oriented code smells and four to six Android-specific code smells [Hec+15a][Hec+15b][MM18]. Mannan et al. [Man+16] decided to broaden this scope and studied 21 object-oriented code smells using the commercial tool InFusion. They analyzed open-source Android and Java desktop applications for these 21 code smells and compared their occurrences. Mannan et al. analyzed 500 Android and 750 Java desktop applications randomly selected from GitHub and detected that most code smells occur in both systems in a similar frequency with major differences only for a few code smells. They concluded that studying code smells on mobile platforms can be done with tools meant for desktop applications and that these results should also hold for other languages. They also found that the code smells that have been studied so far are not the same ones that occur most and that the focus should change to more relevant code smells [Man+16].

Habchi et al. [Hab+17] used PAPRIKA to detect code smells in iOS applications. To be able to use PAPRIKA they used ANTLR4 grammars to generate parsers for Swift and Objective-C code. They analyzed the AST generated by these parsers to create the applications graphs used by PAPRIKA. They then defined three iOS-specific code smells similar to how Android-specific code smells were defined by Hecht et al. [Hec+15a]. They analyzed 176 iOS applications written in Swift and 103 iOS applications written in Objective-C gathered from a collaborative list of open-source iOS applications. They analyzed these applications for three iOS-specific and four object-oriented code smells. They compared code smell occurrences on iOS and Android and concluded that Android applications were more prone to code smells [Hab+17]. They also found that although applications written in Objective-C and Swift had different thresholds for code metrics, the results in proportions of code smell occurrences were similar [Hab+17]. To the best of our knowledge, this has been the only study looking at code smells on iOS applications before conducting our analyses. Our work will build on the work of Habchi et al. [Hab+17] by analyzing 34 object-oriented code smells in iOS applications.

3.2. Library Dependency Network Evolution

Kikas et al. [Kik+17] analyzed the evolution of LDNs of three languages: JavaScript, Ruby, and Rust. They found that for each package manager, the number of libraries is growing. Similarly, the number of direct dependencies and total dependencies per project is increasing. The increase was especially concerning for JavaScript, where the average number of total dependencies grew from one per project to almost 60 between 2011 and 2016. Decan et al. [DMC17] analyzed the LDNs of three package managers npm, CRAN, and RubyGems. They found that proportionally there are more packages with dependencies in CRAN (70%) than in npm and RubyGems (60%). They also found that on average there are few direct dependencies and a much higher number of transitive dependencies. The median number of transitive dependencies for CRAN was five, for RubyGems 8, and for npm 22.

In follow-up work, Decan et al. analyzed the evolution of seven package managers LDNs [DMG19]. They defined and calculated three metrics describing the LDN evolution: the Changeability Index, the Re-usability Index, and the P-Impact Index. They used the libraries.io dataset to analyze how these package manager LDNs change over time. They found that the growth of the number of libraries and dependencies depends on the package manager. Some LDNs have grown linearly, while others have grown exponentially. For most package managers 50% of libraries were updated within two months and libraries that are referenced by other libraries are updated significantly more often than libraries, that are not referenced by other libraries. They also found that 26% to 33% of libraries were never updated. They showed that the number of transitive dependencies is significantly higher than the number of direct dependencies. For some of the package managers, the ratio between transitive and direct dependencies is growing. They also pointed out that the average dependency depth is between three and six, depending on the package manager. The libraries.io data set includes partial data about CocoaPods, Carthage, and Swift Package Manager (the three package managers used in iOS development), but according to Decan et al. this data was incomplete and therefore, these package managers were excluded from the analysis.

Bogart et al. [Bog+21] analyzed the policies and practices for 18 LDNs. Their analysis showed that ecosystems share values on stability and compatibility, but other values tend to differ. The three top values cited by developers for CocoaPods were quality, stability, and compatibility. Blanthorn et al. [BCN19] used tensor decomposition to study different communities within LDNs. They found big differences between package managers, particularly between Elm and R and the more widespread Python, Java, and JavaScript ecosystems. Korkmaz et al.

[Kor+20] found that libraries with a higher number of dependencies tend to have less impact in terms of number of dependents in the LDN. Our work will look into how the number of libraries and dependencies has evolved for the yet to be analysed Swift LDN.

3.3. Technical lag and Semantic Versioning in Library Dependency Networks

Kula et al. [Kul+18] analyzed if developers update their library dependencies and found that over 80% of projects used outdated dependencies. They plotted library usage curves and discovered that new library versions are mostly used by new dependent projects. They also found that affected developers were not likely to respond to a security advisory. The reasons cited by developers to not update libraries were perceived extra workload and added responsibility. Salza et al. [Sal+20] analyzed how developers update library dependencies in Android apps and found that library dependencies are rarely updated. Their analysis showed that mainly only libraries related to Graphical User Interfaces were updated. Developers cited compatibility with newer Android versions and bug propagation as the main reasons for upgrades. Huang et al. [Hua+19] analyzed how often library dependency updates would break Android apps and found that this was rarely the case.

Technical lag can be defined by selecting a distribution to compare to, defining a function to calculate lag for each component, and defining a function to aggregate over the lag of all components [Gon+17]. Typically technical lag for library dependencies can be defined by taking the set of library versions available at the time of release as the distribution of components, taking the time between commits of two components as the lag and aggregating lag time between different versions by taking the maximum to find the lag to the newest available library dependency version. Zerouali et al. [Zer+18] analyzed technical lag in npm library dependencies. They found that the median dependency lag time in npm libraries was 3.5 months which corresponded to a median version lag of one minor and two patch versions. They also found that major releases tend to take longer to receive a patch update. Zeroauli et al. [Zer+19] introduced a formal framework for measuring the technical lag of library dependencies. They analyzed 500.000 npm libraries and concluded that there is a need for more awareness of dependency lag and more integrated tool support for controlling the dependency lag in libraries. Decan et al. [DMC18] showed that dependency lag could be reduced by over 17% if library dependency constraints would rely on semantic versioning that enabled automatic updates of backward compatible changes. Decan et al. [DM19] analyzed semantic versioning for multiple package managers. They found that most version constraints in Cargo, npm, and Packagist are compliant with semantic versioning and that most ecosystems become more compliant over time. They also found that more permissive constraints are updated less often.

Stringer et al. [Str+20] analyzed the technical lag of dependencies in 14 package managers. They found that the majority of library dependencies with fixed version requirements were outdated and were only updated in major updates. They found that semantic versioning would remove most of the lag and recommended tooling uptake.

One way to encourage developers to update lagging dependencies is through automated pull requests. These pull requests rely on semantic versioning[1] to evaluate the probability of breaking changes that may be introduced by the new version. Mirhosseini et al. [MP17] studied if automated pull requests could encourage software developers to upgrade lagging dependencies. They found that only a third of the automated pull requests were actually merged. However, projects that used pull request notifications were more likely to upgrade their library dependency versions. Ochoa et al. [Och+22] analysed over 119,879 library updates and found that 83.4% of these upgrades complied with semantic versioning with compliance increasing over time. They also found that only 7.9% of projects are affected by breaking changes as most code with breaking changes in the library dependencies is not used by the projects. Hejderup et al. [HG22] analyzed if tests could be used to automate dependency updates. They found that tests in well-tested Java projects only cover 58% of direct and 21% of transitive dependency calls and that injected faults could be detected only for 47% of direct and 35% of indirect dependencies on average. They conclude that a combination of static and dynamic analysis should be used in future library dependency updating systems.

Suwa et al. [Suw+17] analyzed library dependency downgrades in Java libraries. They found that library dependencies with shorter release cycles were more likely to have a rollback. Equally, projects that responded to new library versions quicker were more likely to have a rollback. Cogo et al. [COH19] found that if a downgrade is performed then the version requirement type is often changed to exact.

Our work will analyse technical lag for the three package managers CocoaPods, Carthage and Swift PM and how the choice of package managers changes how developers update their dependencies.

3.4. Vulnerabilities in Library Dependency Networks

Decan et al. [DMC18] analyzed vulnerable third-party libraries with nearly 400 vulnerabilities in the npm LDN encompassing 610 thousand libraries. They found that a third of the vulnerabilities are fixed before their discovery date and half of the vulnerabilities are fixed after the discovery date but before the publication date. They found that over 40% of library versions with vulnerable dependencies could not automatically update the vulnerable dependency version due to unsuitable dependency constraints. Zerouali et al. [Zer+22] studied how long it takes

[1] `https://semver.org`

for vulnerabilities in libraries from npm and RubyGems to be fixed, how vulnerabilities spread through the LDN, and if vulnerable libraries are updated. They matched vulnerability data from Snyk to npm and RubyGems libraries and found that more than 15% of the latest library versions are directly dependent on vulnerable libraries. Additionally, dependencies to vulnerable libraries affected 42.1% of npm and 39% of RubyGems libraries. They found that one-third of vulnerable dependencies could be fixed by updating the vulnerable dependency version.

Düsing et al. [DH21] matched vulnerabilities from Snyk to libraries from the Maven, NuGet, and npm LDNs. They, then, analyzed how vulnerabilities in direct and transitive dependencies affect different LDNs. They found that only 1% of libraries in NuGet and 8% of libraries in npm are affected by vulnerable dependencies. Whereas, 29% of libraries served through Maven have dependencies on vulnerable library versions. They also studied how long it takes for libraries to update their vulnerable dependencies after vulnerability disclosure and found, that at least some libraries are probably using automated tools that follow vulnerability databases and update all vulnerable dependencies automatically.

Li et al. [Li+21] analyzed LDNs of Java projects from Maven and GitHub. They matched vulnerability data from NVD to these Java projects and found 503 vulnerabilities matching 174 Maven projects and 3326 vulnerabilities matching 840 GitHub projects. They observed libraries with vulnerable dependencies from 2019 to 2020 and found that only 5% vulnerable dependencies were fixed in this time frame.

Zimmermann et al. [Zim+19] studied security risks in the npm LDN. They found, that when installing an average npm library the user implicitly trusts 80 dependent libraries. They also studied risks related to project maintainers and found, that the average maintainers' control in the LDN has been increasing over the years. When analyzing publicly reported vulnerabilities from Snyk, they found, that up to 40% of all libraries have (direct or transitive) vulnerable dependencies. Alfadel et al. [Alf+20] analyzed the use of vulnerable npm dependencies in Node.js applications. They found, that although 67.93% of examined applications depended directly on vulnerable libraries, 94.91% of these vulnerabilities were not known at the time. For Python projects, Alfadel et al. [ACS23] found that (40.86%) of the vulnerabilities are only fixed after they were publicly released. They also found that over 50% of projects with dependencies have a dependency on a vulnerable third-party library.

Prana et al. [Pra+21] analyzed publicly reported vulnerabilities in Java, Python, and Ruby LDNs. They found that the most vulnerable dependencies (mean of 78.4% for Java, 97.7% for Python, and 66.4% for Ruby) were not updated to patched versions during their observation period. Vulnerable dependencies that were updated to a patched version took three to five months to update.

Massacci et al. [MP21] defined leverage as how much code is imported from third-party libraries in comparison to custom code. They found that medium-sized libraries import proportionally more code than large libraries. They also found that libraries with higher leverage are more likely to have vulnerabilities. Gkortzis et al. [GFS21] studied the distribution of vulnerabilities in custom code and included third-party libraries for 1244 open-source projects. Their analysis was based on both publicly reported vulnerabilities and potential vulnerabilities reported by static analysis tools. They found that larger projects were associated with more vulnerabilities, both in custom code and in the included third-party library dependencies. A higher number of library dependencies was strongly correlated with the number of vulnerabilities.

Alfadel et al. [Alf+21] analyzed 2,904 JavaScript projects on GitHub that are subscribed to Dependabot (an automatic update system that monitors vulnerable dependencies). They found that the majority (65%) of Dependabot pull requests related to security are accepted and often merged within a day. They also found that only 3.2% of the pull requests they examined manually broke the build. In contrast, Chinthanet et al. [Chi+21] analyzed the fixing commits of vulnerable npm libraries and found that a fixing release is rarely simply a fix for a vulnerability, but often also included other changes.

Pashchenko et al. [Pas+18] analyzed vulnerable Java library dependencies and found that in 20% of the cases, the vulnerable dependency was not deployed and therefore the vulnerability did not affect the project. They found that the majority of the vulnerable dependencies (81%) could have been fixed by upgrading to the patched version. Zapata et al. [Zap+18] analyzed the use of three vulnerable JavaScript libraries in 60 projects and found that 73.3% of projects dependent on vulnerable libraries were not vulnerable themselves. Ponta et al. [PPS18] combined static and dynamic analysis to determine if the vulnerable code in a third-party library is reachable. They implemented a tool called Vulas that makes it possible to find library versions that are not vulnerable and provides update recommendations to users. Later Ponta et al. [PPS20] release more mature version called Eclipse steady. They compare the analysis results of Eclipse steady with OWASP Dependency Check. They find that findings that are only reported by Eclipse steady are true positives while 88.8% of findings reported by OWASP Dependency Check are false positives proving the effectiveness of their code-centric analysis approach. Imtiaz et al. [ITW21] conduct an in-depth comparison of 9 source code analysis tools on a large web application composed of Maven and npm projects. They find that vulnerability reporting varies greatly between tools and that the accuracy of the underlying vulnerability database has a great impact on the output quality.

Sometimes developers decide to include library dependencies by copying the library into their project without using a package manager. This makes updating library dependencies more difficult, but allows developers to modify the library before using it. Dann et al. [Dan+21] show that the use of modified third-party libraries is common in Java projects and that conventional vulnerability scanners struggle to analyze such dependencies correctly.

Our work will analyse how vulnerabilities spread in the Swift LND. We will quantify how much public information is available on the publicly reported vulnerabilities in the Swift ecosystem to see if refinement of tools analysing vulnerable dependencies would be feasible in this ecosystem.

4. TOOLS

This chapter covers Contribution 1. So far the tool support for applications written in Swift is lacking. In this chapter, we describe the three developed tools that allow maintainability and security analysis of projects written in Swift:

- **GraphfiySwift**: analysis of code smells in snapshots of projects.
- **GraphifyEvolution**: analysis of the evolution of projects, an extendable tool that can be used for code smell analysis and beyond.
- **SwiftDependencyChecker**: detection of vulnerable library dependencies in Swift projects. The tool can be integrated into the Xcode build process.

The initial goal for tool development was to provide tool support for the maintainability analysis of applications developed in Swift. The tool GraphfiySwift was developed to facilitate code smell analyses of applications written in Swift. Section 4.1 describes the tool GraphifySwift in detail. Based on our experiences gained from developing and using GraphifySwift the enhanced goal was to build a modular code analysis tool that can be used for both maintainability and security analysis. A tool called GraphifyEvolution was developed that could be easily extended with external analyzers. The source code analysis part of GraphifySwift was used as the base of GraphifyEvolution and the code smell analysis part became an external analyzer for code smell analysis. Section 4.2 describes the tool GraphfiyEvolution in detail. Lastly, SwiftDependencyChecker was developed to allow the analysis of library dependencies including the detection of vulnerable library dependencies. Section 4.3 describes the tool SwiftDependencyChecker in detail.

4.1. GraphifySwift

Code smells are reoccurring patterns in code that have been identified as bad practices [Fow18]. They can lead to technical debt and long-term maintainability problems [Fow18]. Code smells and their implications have been studied a lot for desktop applications, especially Java desktop applications [KDG09; Moh+09; Olb+09]. Code smells can lead to more change-prone [KDG09] and fault-prone [Lin+14] code, increasing the maintainability effort [Olb+09]. Given the significance of the smartphone market, it is important to study code smells on Android and iOS, too.

Most of the research on code smells in mobile applications has been carried out on the Android platform [Hec+15a][Pal+13] [Tuf+15][Ver13][Man+16]. Mannan et al. [Man+16] analyzed 21 object-oriented code smells in open source Android applications. They also compared code smell occurrences on Android and Java desktop applications and suggested that this result should hold for other languages as well. Mateus et al. on the other hand have shown that code quality does differ for different languages used [MM18]. Hecht [Hec15] developed a tool called PAPRIKA to analyze four object-oriented and six Android-specific code smells in Android applications. This tool analyses the Android APK, generates a model, and saves it in a graph database. Code smells are then defined as queries [Hec+15a]. This approach relies on metric and rule-based code smell detection. For iOS, Habchi et al. [Hab+17] compared code smell occurrences in iOS and Android applications using PAPRIKA by translating iOS code to a suitable format. They analyzed four object-oriented code smells and three iOS-specific code smells. Their analysis showed that Android applications were more susceptible to code smells than iOS applications. To the best of our knowledge, Habchi et al. [Hab+17] were the only ones to study code smells on iOS so far, and the scope of their study was limited to a set of four object-oriented and three iOS-specific code smells, where they studied 279 open source iOS applications.

Our goal is to provide tool support that allows studying code smells in iOS applications in more depth and breadth. To get a broad overview of object-oriented code smells and their occurrences in iOS applications, we combined 21 code smells studied by Mannan et al. [Man+16], 22 code smells that were defined by Fowler [Fow18] and six out of seven code smells studied by Habchi et al. [Hab+17] resulting in 36 code smells.

GraphifySwift is a tool that allows code smell analysis of applications written in Swift. This section is based on work from [RP20]. GraphfiySwift can be used by developers to find code smells in their applications. The tool can additionally be used by researchers who wish to analyze a large set of applications at once.

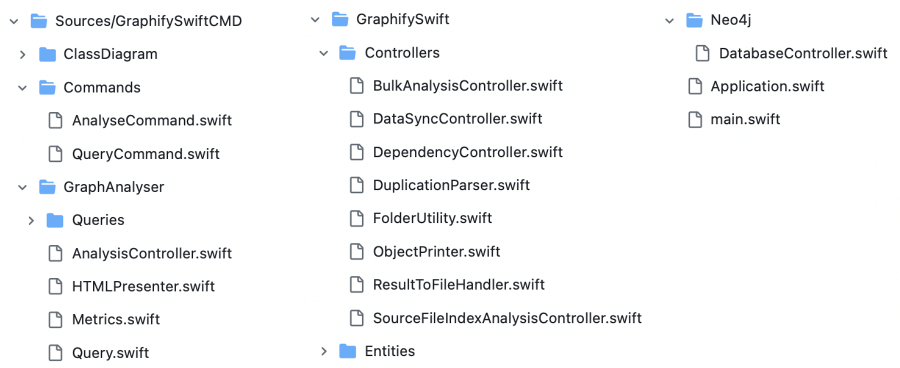
4.1.1. Architecture

GraphifySwift takes as input either a folder path to a Swift project or a list of source code repositories. It then analyses each of these projects and enters information about the project, such as classes, methods, and method calls into a Neo4J database. After the database is populated it is possible to use GraphifySwift to detect code smells by querying the database. Each code smell is defined as a cypher query.

The project structure of GraphifySwift is presented in Figure 4. The architecture of the tool is fairly simple and consists of four main entities. The `Application` class contains the overall program logic. The commands `AnalysisCommand` and `QueryCommand` define the behavior of the two types of activities the tool is capable of, namely: analyzing the source code of a project, and querying code smells from the database. The third entity GraphifySwift handles the source code analysis and the GraphAnalyser entity handles the analysis of the database.

4.1.2. Implementation

GraphifySwift analyses Swift code, generates a model of this code, and enters it into a Neo4J graph database. For indexing the source code and generating the



```
Sources/GraphifySwiftCMD
  ClassDiagram
  Commands
    AnalyseCommand.swift
    QueryCommand.swift
  GraphAnalyser
    Queries
    AnalysisController.swift
    HTMLPresenter.swift
    Metrics.swift
    Query.swift
GraphifySwift
  Controllers
    BulkAnalysisController.swift
    DataSyncController.swift
    DependencyController.swift
    DuplicationParser.swift
    FolderUtility.swift
    ObjectPrinter.swift
    ResultToFileHandler.swift
    SourceFileIndexAnalysisController.swift
  Entities
Neo4j
  DatabaseController.swift
Application.swift
main.swift
```

Figure 4. Code structure of GraphifySwift

Table 2. Relationships between database entities

Relationship name	Between entities
APP_OWNS_MODULE	App and Module
MODULE_OWNS_CLASS	Module and Class
CLASS_OWNS_METHOD	Class and Method
CLASS_OWNS_VARIABLE	Class and Variable
USES	Method and Variable
CALLS	Method and Method
IS_TYPE_OF	Variable and Class or Argument and Class
DUPLICATES	Class and Class

structure of the Swift code we used a framework called SourceKittenFramework[1] that acts as a wrapper around Apple's powerful SourceKit. Code duplication was detected using an external tool called jscpd[2]. The graph database contains the following nodes: App, Module, Class, Function, Variable, and Argument. These nodes are connected through the relationships listed in Table 2. This model differs slightly from the one used in PAPRIKA. We added the Module node and the following relationships *APP_OWNS_MODULE*, *MODULE_OWNS_CLASS*, *IS_TYPE_OF*, and *DUPLICATES*.

In GraphifySwift, code smells are defined as queries. To give an example we show the query definition for Long Method. Other query definitions can be found in Appendix A.3. A code smell query can be simply based on a metric calculated statistically as seen in this example or it can be defined through more complex relationships between classes of an application. As Neo4J is a graph database it is especially well optimized for this kind of pattern matching.

[1]`https://GitHub.com/jpsim/SourceKitten`
[2]`https://GitHub.com/kucherenko/jscpd`

Long Method is normally defined as a method, that has more lines than a given threshold. In the case of Swift code, we decided to use the number of instructions instead of the number of lines since number of instructions ignores comments and empty lines. This metric is more easily accessible as well through the used SourceKittenFramework.

```
MATCH
    (c:Class)-[r:CLASS_OWNS_METHOD] -> (m:Method)
WHERE
    m.number_of_instructions >
                        veryHighNumberOfInstructions
RETURN
    distinct(m.app_key) as app_key,
    count(distinct m) as number_of_smells
```

4.1.3. Usage

The code analysis in GraphifySwift can be run with the following command:

```
GraphifySwiftCMD analyse --appkey <applicationKey>
                                  <pathToRepository>
```

The appkey should be a unique token, for example, the app name, for each project that is entered into the database. The command can be run with the additional options:

```
--includModules -m
--noDatabase -n
--resultOutput -o
```

Here includeModules divides code not only into classes but additionally into different modules depending on the project structure. The option noDatabase allows running the analysis without entering anything into the database, which is sometimes useful for testing purposes. Lastly, the option resultOutput prints out verbose output into the console.

Additionally, it is possible to run the analysis on a large set of projects at once by running:

```
GraphifySwiftCMD analyseBulk --fileName <pathToJsonFile>
                                        <folderToSaveFilesTo>
```

The pathToJsonFile should contain data on repositories to analyze in JavaScript Object Notation (JSON) format. The JSON format was chosen to allow the use of the collaborative list of open-source iOS applications[3] as a direct input. The folderToSaveFilesTo folder is used to save downloaded project repositories.

Running code smell queries using this tool generates CSV files for each code smell and allows us to analyze the results. The CSV option was chosen to allow

[3]https://github.com/dkhamsing/open-source-ios-apps

Table 3. Comparison: our results vs results in [Hab+17]

Code smell	**Our result**	**results in [Hab+17]**
Long method	86.4%	85.6%
Complex class	53.6%	65.6%
Massive View Controller	8.8%	8%
Ignoring Low Memory Warning	73.6%	88%
BLOB Class	10.4%	37.6%
Swiss Army Knife	3.2%	11.2%

easy analysis with common data analytics tools such as pandas[4]. Another option is to choose the HTML result output which combines results for all code smells and allows browsing the analysis results easier. Results can be queried by running

```
GraphifySwiftCMD query -q all --csvFolder <pathToFolder>
```

Lastly, a prototypical feature exists that allows the generation of class diagrams:

```
GraphifySwiftCMD classDiagram <pathToRepository>
```

Code smell queries are defined using thresholds that should be calculated for each analysis set. These thresholds can be calculated using the box-plot technique. A value is seen as very high if it is higher than $Q3 + 1.5 * IQR$ where Q3 is the third percentile and IQR is the inter-quartile range[Hec+15b].

4.1.4. Evaluation

To validate our GraphifySwift tool, we replicated results from Habchi et al. [Hab+17]. We updated thresholds in our code smell queries to match those used in [Hab+17]. The thresholds used are listed on in Appendix A.1. The replication is done using the the same version of the collaborative list of iOS applications as was used by Habchi et al. [Hab+17]. To get matching results to those in the article we recalculated the percentages using their data set of applications that showed which apps were affected by which code smells (see results in Table 3).

For four code smells (Long Method, Complex Class, Massive View Controller, and Ignoring Low Memory Warning), the percentages were very similar. For two code smells (BLOB Class and Swiss Army Knife) the results differed a lot. We investigated why this was the case for Swiss Army Knife and looked at the source code of each application where the decision did not match. We discovered that Habchi et al. [Hab+17] included dependencies in the code they analyzed while we did not. Therefore, for code smells concerning view controllers, the results were very similar as most dependencies do not include view controllers. They do however often contain additional interfaces and might contain more complex classes which explain the differences between BLOB Class and Swiss Army Knife. We

[4] `https://pandas.pydata.org`

decided in our implementation to only count code smells in the application code, as opposed to also in the dependencies, as this is the code that developers have control over and whose maintainability they are concerned of.

We also compared the distribution of code smells detected by Habchi et al. [Hab+17] and detected by our tool. The differences in the distribution of code smells are presented in Figure 5. The code smell Swiss Army Knife appears more than twice often in the analysis done by Habchi et al. [Hab+17] since their analysis included a lot more interfaces than our analysis. For the same reason, proportionally fewer classes were View Controllers which resulted in a lot more instances of Ignoring Low Memory Warning in our case. Despite these differences, the results are very similar and we conclude that our implementation of code smell detection is valid.

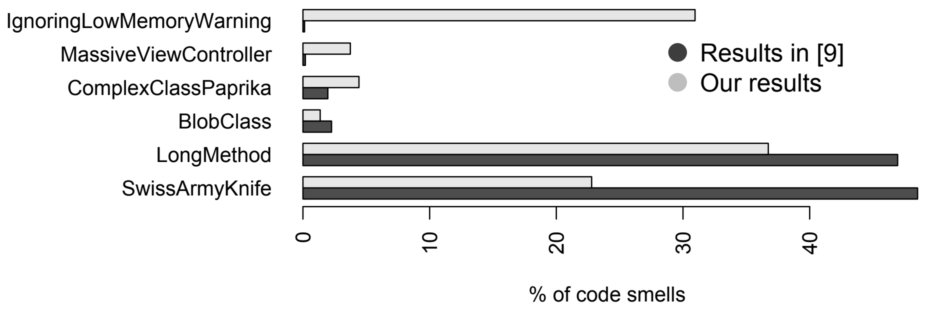


Figure 5. Comparison of the distributions of code smells using PAPRIKA [Hab+17] vs GraphifySwift (our tool)

4.1.5. Application

We used GraphifySwift to analyze code smells in open-source iOS applications. Further, we used the code smell queries from GraphifySwift in combination with PAPRIKA and compared code smells in iOS and Android. These results are described in detail in Chapter 5 and used to answer research questions 1.1 and 1.2.

4.2. GraphifyEvolution

For analyzing the evolution of mobile applications, multiple approaches have been used. To track the software quality of Android applications, Hecht et al. [Hec+15] used PAPRIKA to find Android-specific code smells in multiple versions of the same application. Mateus et al. [MM18] used PAPRIKA to analyze the evolution of Android applications, applying the tool for each commit. They analyzed differences in Android applications written in Java and Kotlin. Tufano et al. [Tuf+17] implemented a tool called HistoryMiner that runs DECOR on each commit, but only for changed files. The output of the tool is a list of commits for each source file where that file was added, modified, or deleted and if it had a code smell. Habchi et al. [HRM19; HMR19] implemented a tool called SNIFFER that extracts commits from a git repository, detects code smells in each commit using PAPRIKA, and outputs the code smell evolution of a project.

To the best of our knowledge, no studies exist analyzing the code smell evolution of iOS applications. We extended the tool GraphifySwift and implemented a tool called GraphifyEvolution [5] that can analyze applications written in various languages, including iOS applications written in Swift. Different from previous tools that output data about the code smell evolution, we took an approach where the output of the analysis is a graph database containing structural information about all versions of the application. GraphifyEvolution detects which classes, methods, and variables are changed during a commit and records only these changes. This makes it possible to query code smells for all versions of the application at once. Additionally, we implemented the tool in a modular manner making it easy to add support for additional languages and external analysis tools that can be run for each commit. Currently, we have implemented support for Swift and preliminary support for Java, and C++. Support for external tools is added for jscpd[6] that finds code duplicates and insider[7] that detects security vulnerabilities.

GraphifyEvoltuion can analyze the evolution of applications in bulk or analyze a single application. We built the tool mainly to analyze open-source applications and frameworks written in Swift. GraphifyEvolution enters the application data into a Neo4J database. Using a graph database allows for easy querying of patterns in the database, for example code smells. This database can then be queried with GraphifyEvolution or directly through the Neo4J browser. The possibility of running custom queries from the browser makes the tool more versatile. Query results can be exported in .csv files and analyzed in the user's tool of choice as illustrated in Figure 6.

4.2.1. Architecture

The tool consists of eight main elements as described in the following sub-sections and illustrated in Figure 7. The tool is comprised of the following main entities:

1. **Main**: parsing application arguments and setup
2. **AppAnalysisController**: Language-independent source code analysis using the output of the syntax analyzer.
3. **SyntaxAnalyser** (C++, Swift, and Java): Language-specific source code analyzer that outputs language-independent code structure.
4. **LocalFileManager** (C++, Swift, and Java): Language-specific management of source code files.
5. **AppManager** (Simple, Git, and Bulk): Manager for analysis process depending on the analysis type.

[5]`https://github.com/kristiinara/GraphifyEvolution`
[6]`https://github.com/kucherenko/jscpd`
[7]`https://github.com/insidersec/insider`

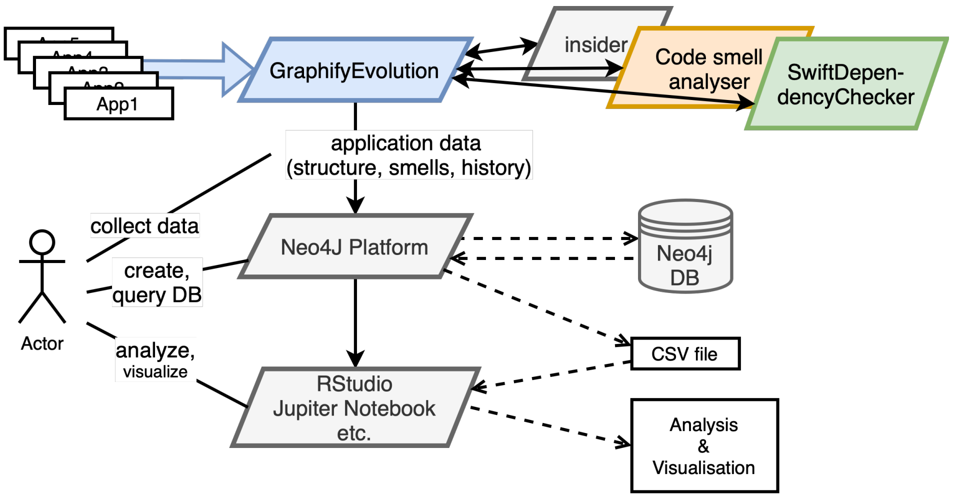
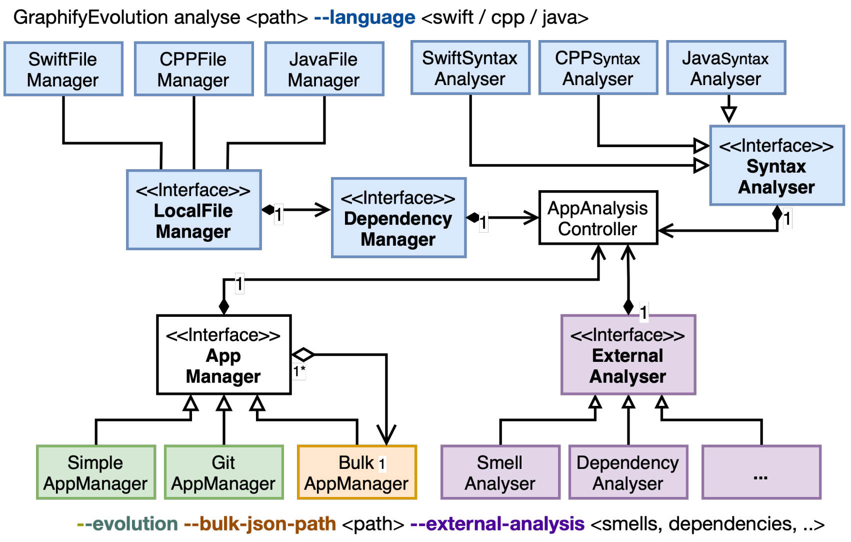


Figure 6. Using GraphifyEvolution for data analysis.



GraphifyEvolution analyse <path> --language <swift / cpp / java>
SwiftFile Manager
CPPFile Manager
JavaFile Manager
SwiftSyntax Analyser
CPPSyntax Analyser
JavaSyntax Analyser
<<Interface>> Syntax Analyser
<<Interface>> LocalFile Manager
<<Interface>> Dependency Manager
AppAnalysis Controller
<<Interface>> App Manager
<<Interface>> External Analyser
Simple AppManager
Git AppManager
Bulk AppManager
Smell Analyser
Dependency Analyser
...
--evolution --bulk-json-path <path> --external-analysis <smells, dependencies, ..>

Figure 7. Architecture and usage of GraphifyEvolution

6. **DependencyManager** (Simple, Maven, Gradle): Managers that filter out dependency-related source code that is not part of the custom code of a project.
7. **ExternalAnalyser** (Duplication, InsiderSec, Metrics, Smells, Dependency, Import, Language, Vulnerability): Implementations of different external analyzers, for example for analyzing code duplication, security vulnerabilities code smells, dependencies, import statements, and languages.
8. **Database** (Neo4J): Communication with the Neo4J database.

The application starts with `Main.swift` where the command-line arguments are parsed and the application is set up. The command-line arguments decide which implementation of the `SyntaxAnalyser`, the `LocalFileManager`, the `AppManager`, the `DependencyManager`, and the `ExternalAnalyser` are used. Figure 7 shows how the command-line arguments correspond to the tool implementation.

An important feature of GraphifyEvolution is that it is fairly easy to extend the tool by implementing additional syntax analyzers for other programming languages or additional external analyzers.

4.2.2. Implementation

The `Main` class handles input arguments and setups the application controller. Depending on input arguments the correct implementations for LocalFileManager, SyntaxAnalyser (both language-specific), DependencyManager (platform-specific), AppManager (either simple, git, or bulk), and ExternalAnalyser (if external analysis tools are included) are chosen.

The analysis of applications is done in the `AppAnalysisController` class. This analysis is language-agnostic. It relies on the `SyntaxAnalyser` to find classes in source code files. `AppAnalysisController` decides which files should be analyzed, depending on whether they have changed compared to the previous version of the application. For files that have not changed, classes from the previous application version are used. For files that are added or changed, `AppAnalysisController` calls the `SyntaxAnalyser` to find classes in these files. New classes found in these files are added to the application. For classes that have changed, a parent class is found and potential methods and variables are handled. For potential methods and variables `AppAnalysisController` decides, based on changed lines, if a method or variable is added, removed, or changed and handles them accordingly. The control flow graph of this decision process is given in Appendix B.

The class `LocalFileManager` is used to find source code files in a given project folder. Currently, there are implementations for three different languages: Swift, Java, and C++. Each of these implementations defines allowed file types and ignored folders and can include any other language-specific differences. In future versions, these managers will be able to deal with different dependency management systems, for example if an iOS app requires dependencies through CocoaPods, Carthage or Swift PM.

The class `AppManager` handles the different app versions to be analyzed. Currently, there are three different `AppManager` implementations: `SimpleAppManager`, `GitManager`, and `BulkAppManager`. `SimpleAppManager` takes a project folder of an application and creates one `AppVersion` to be analyzed. `GitManager` takes a project folder of an application and as its name suggests creates app versions for each git commit. The implementation uses git log to find all git commits and git diff to find changes for each commit. `GitManager` also tries to set the branch from which a commit originates. This is done by finding the next merge commit and trying to find the branch name from the commit message. In most cases, this works correctly, but sometimes developers change the default git message, then the branch name can be incorrect. This is currently the most reliable way to find branch names as git does not keep a record of deleted branches. `BulkManager` takes a JSON file that contains information about projects to be analyzed such as name and project folder path or repository Uniform Resource Locator (URL). `BulkManager` can be setup to use a `SimpleAppManager` or `GitManager` if the evolution of multiple applications should be analysed. Other types of managers can be added. Possible implementations can be for a manager that adds new changes to an existing application in the database.

The class `DependencyManager` handles the setup of dependencies of an application. Each type of dependency management needs its own `DependencyManager`. Currently three implementations exist: `SimpleDependencyManager`, `GradleDependencyManager`, and `MavenDependencyManager`. `SimpleDependencyManager` does not update any dependencies and `GradleDependencyManager` updates dependencies with Gradle and the `MavenDependencyManager` updates project dependencies with Maven.

The class `SyntaxAnalyser` is language-specific and handles analyses of source files. For Swift, the class `SwiftSyntaxAnalyser` uses the SourceKittenFramework to index and analyze Swift code. `CPPSyntaxAnalyser` calls a Python script that uses clang to index C++ source code. `JavaSyntaxAnalyser` calls a small Java program that uses `JavaParser` and `JavaParserTypeSolver` to index and analyse source code written in Java. In principle, `SyntaxAnalyser` can be implemented in two ways. The quickest implementation is to use an external library or tool to index the source code and pass a JSON object to the default implementation of the `parseClassFrom` function. This JSON object needs to have a specific structure[8] containing classes, variables, methods, and instructions. If converting source code into this structure is unfeasible it is possible to override methods for parsing classes, methods, variables, and instructions. In this case, a `Class` object needs to be returned. `SwiftSyntaxAnalyser`, `JavaSyntaxAnalyser`, and `CPPSyntaxAnalyser` all follow the first approach.

The class `ExternalAnalyser` makes it possible to run additional external analyses on each app version. External analyses can be either on the app or class (file) level. The findings of the analyses will then be attached to the corresponding app or class, respectively. Additionally, `ExternalAnalyser` needs to declare for which programming languages it can be used. Currently, there are 8 analyzers that implement `ExternalAnalyser`. The first analyzer is `DuplicateCodeAnalyser` and uses jscpd[9] to find code duplication. `InsiderSecurityAnalyser`, the sec-

[8] `https://github.com/kristiinara/GraphifyEvolution/blob/master/documentation/syntax_analyser.md`

[9] `https://github.com/kucherenko/jscpd`

ond analyzer, uses insider[10] to find vulnerabilities in source code. The third and fourth analyzer are used to find and save data related to code smells to the application database. The `MetricsAnalyzer` calculates class-based metrics and the `SmellsAnalyzer` queries code smells for new and changed classes. It is possible to query code smells without using these analyzers but they make it possible to extract code smell data faster later on. The fifth analyzer is `DependencyAnalyser` which finds dependencies of an application declared with CocoaPods, Carthage, or Swift Package Manager. It finds the names and versions of the library dependencies and adds them to the database. The sixth analyzer is `ImportAnalyser` which finds import statements in Swift classes and adds them to the database. The seventh analyzer is `LanguageAnalyser` which detects the language of a file and adds that information to the database. The last analyzer is `VulnerabilityAnalyser` which finds if a project to be analyzed has a publicly reported vulnerability in the NVD.

GraphifyEvolution uses the Neo4J graph database platform that has previously been used for similar purposes [Hab+17; Hec+15; RP20]. Neo4J makes it possible to easily query different patterns in code to find code smells in applications, as was demonstrated in [RP20]. For more complex analyses, Neo4J has a data science plugin that can be used to run cluster analyses. Before running GraphifyEvolution, a Neo4J database instance needs to be running (either on localhost or remotely). Queries to the database are done through the HTTP interface.

The Neo4J database stores data of nodes and relationships between nodes. There are 6 types of nodes: App, Class, Method, Variable, Argument, and External, and 7 types of relationships that describe the structure of the application.

- APP_OWNS_CLASS
- CLASS_OWNS_METHOD
- CLASS_OWNS_VARIABLE
- CALLS
- USES
- CLASS_REF
- EXTERNAL_REF

Additionally, there are two types of relationships that describe the evolution of the application. The two relationships `CLASS_CHANGED_TO` and `CHANGED_TO` connect class and application versions, respectively.

Nodes and relationships have attributes that store information about these entities. A more detailed description of the database structure can be found in Appendix B.2.

4.2.3. Usage

Code smell analysis of a single application can be run as follows:

[10]https://github.com/insidersec/insider

```
GraphifyEvolution analyse <app folder>
                         --external-analysis smells
```

Here `app folder` is the project folder path. The flag `external-analysis` lets the user choose what external analyzer to use. It can be omitted or used multiple times. After the analysis has finished, all code smells found by GraphfiyEvolution can be extracted in CSV format by running

```
GraphifyEvolution query
```

If GraphifyEvolution should run a specific query, instead of all code smell queries, it is possible to pass this as a command-line argument.

Bulk analysis of Java projects analyzing the evolution of these projects can be run as follows:

```
GraphifyEvolution analyse <apps folder>
                         --bulk_json_path <json path>
                                --language java --evolution
```

The `bulk-json-path` points to the JSON file containing information about applications to be analyzed. The `evolution` flag determines whether the evolution or a single version of the application is analyzed. Language can be set by the `language` option and can be either Swift, C++, or Java. It describes the project language. It is not yet possible to analyze projects that comprise multiple programming languages.

All options how GraphifyEvolution can be run is shown in Figure 7. Here `path` is either the project folder path or (in case of bulk analysis) folder path where analyzed applications are downloaded.

A prerequisite of running GraphifyEvolution is a running Neo4J database. The username and password for the Neo4J database need to either match the default values in the code or the user needs to pass them with command-line arguments.

4.2.4. Evaluation

For a preliminary evaluation, we analyzed the code smell evolution of 30 open-source iOS apps. We chose apps that were open source and whose repositories contained all the required code to run the application. This means we used apps that either did not use any dependency management or used Carthage and committed all the dependency files. Future versions of the tool will support other dependency management solutions, such as CocoaPods and Git submodules.

In total, we analyzed 30 applications, 3878 app versions, 7086 classes, 7965 methods, and 7794 variables. The analysis took 17 hours, this analysis included code smell analysis, duplication analysis, and vulnerability analysis with insider. Generated data can be downloaded from the tool web page [11] and browsed using the Neo4J community version. The results of the analysis are reported in Subsection 5.2.3.

[11] `https://github.com/kristiinara/GraphifyEvolution/blob/master/example_data/overview.md`

Our evaluation showed that it is feasible to analyze the evolution of multiple applications. However, we did observe that the time required to analyze the evolution of very old and large applications was extensive. To better handle applications with large git histories we added the option to only analyze git tags. This results in a less detailed history of the applications, but allows faster analysis while keeping the most important versions of the application. Additionally we added the option to start at a given git commit to allow the analysis of the most recent application versions. To better handle large applications we modified the code that handles syntax analysis by rewriting functions that relied on recursion. Our experience showed that in some cases the recursive nature of these methods led to call stack overflows.

4.2.5. Application

In addition to the preliminary evaluation, we used GraphifyEvolution to construct the LDN of the Swift ecosystem. The resulting dataset is described in detail in Section 6.1.2. We analyzed a total of 60533 libraries, 572131 library versions and detected 23419 dependencies between library versions. We used this dataset to analyze the Swift LDN. These results are described in detail in Chapter 6 and used to answer the research questions 2.1, 2.2, 2.3, and 2.4.

4.3. SwiftDependencyChecker

In 2015, a vulnerability in the popular iOS third-party library AFNetworking was found. The vulnerability affected around a thousand iOS applications with millions of users [Kum15]. A fixed version of this library was published over six years ago and the vulnerability was publicly reported. We analyzed open-source iOS applications and found four applications that there are still actively maintained and used applications on the app store that use this old vulnerable version of the library.

Developers can search for publicly reported vulnerabilities in public vulnerability databases such as the National Vulnerability Database (NVD)[12]. Such vulnerability databases can be difficult to search and checking for vulnerabilities can be tedious. To make this task easier for developers, several tools have been developed that automatically analyze project dependencies and detect vulnerable library versions. Some of these tools include experimental support for the CocoaPods package manager but no existing tool supports all three package managers used in iOS app development.

We developed SwiftDependencyChecker [13] , a tool that checks dependencies in Swift projects by analyzing the CocoaPods, Carthage, and Swift Package Manager manifest files. It then queries the NVD database and checks if any of the used

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

[13]`https://GitHub.com/kristiinara/SwiftDependencyChecker`

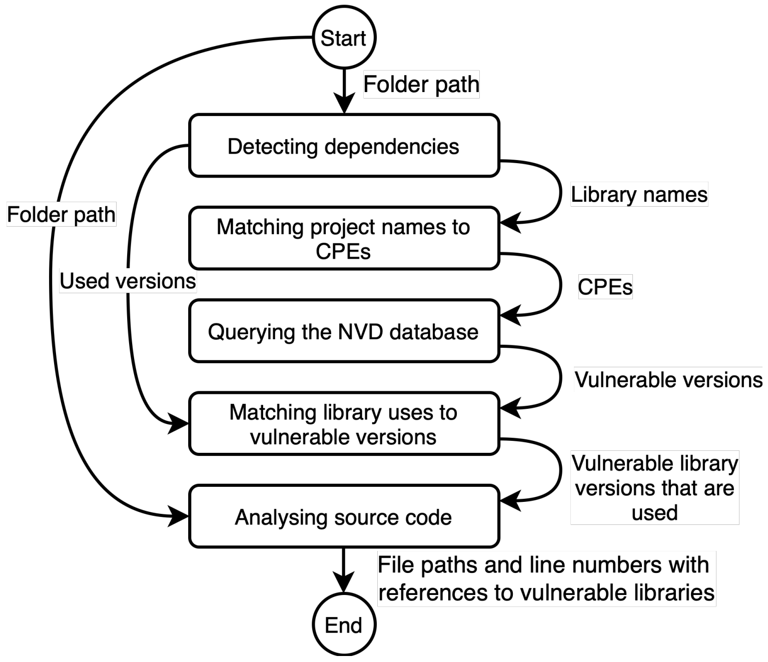

Start
Folder path
Detecting dependencies
Library names
Matching project names to CPEs
CPEs
Querying the NVD database
Vulnerable versions
Matching library uses to vulnerable versions
Vulnerable library versions that are used
Analysing source code
Folder path
Used versions
File paths and line numbers with references to vulnerable libraries
End

Figure 8. Architecture pipeline of SwiftDependencyChecker

libraries have publicly disclosed vulnerabilities. To increase the usability of the tool it can be easily integrated into the Xcode build process. Xcode then displays warnings generated by SwiftDependencyChecker in the source code editor.

4.3.1. Architecture

SwiftDependencyChecker has a simple architecture. It consists of five components that are executed in a pipeline as shown in Figure 8. Each component encapsulates one mostly self-contained analysis step that takes the output of the previous step as input. The components are:

1. Detecting dependencies: analysing the package manager manifest file to extract information on library names and version.
2. Matching project names to CPEs: matching library names to project ids (CPE - Common Project Enumeration) used by vulnerability databases.
3. Querying the NVD database: finding up-to-date vulnerability information on given CPEs from the NVD database.
4. Matching library uses to vulnerable versions: finding library uses that match vulnerable library versions found earlier.
5. Analysing source code: finding import statements of vulnerable library instances.

The implementation of these components is described in more detail in the next subsection.

4.3.2. Implementation

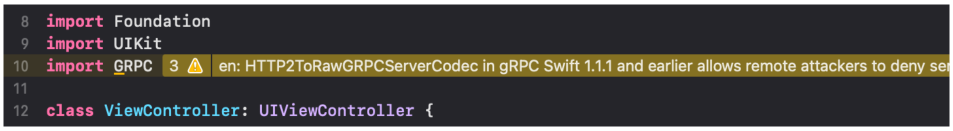
First, package manager resolution files Podfile.lock, Cartfile.resolved and Package.resolved are analyzed. SwiftDependencyChecker detects which library versions the given project depends on and stores this data in the libraries.csv file. This analysis is rerun every time and data in the libraries.csv file is currently only used for debugging purposes. How dependencies are defined differs for each of these package managers. Carthage and Swift Package Manager are both decentralized package managers and it is possible to include a library by providing its repository URL. CocoaPods on the other hand has a central repository that contains details on all libraries that can be installed through CocoaPods. To make library definitions better comparable CocoaPods library names are "translated". For this, the CocoaPods central repository is cloned and details for each library are extracted. To speed up the "translation" process all translations are stored in translations.csv and only new occurrences of libraries are "translated" by analyzing the CocoaPods repository.

Next, SwiftDependencyChecker goes through all libraries and uses the official Common Platform Enumeration (CPE) dictionary provided by NVD to match libraries to CPE values used in NVD. For each library that NVD references in the vulnerability database, there is a CPE that uniquely identifies this library. To match libraries to their corresponding CPE values SwiftDependencyChecker goes through the CPE dictionary[14] and searches for a line that contains the libraries username/projectname value (which is used as library name). If a match is found it means that the line contains a reference to the project repository. A reference to a project repository is followed by a CPE item. SwiftDependencyChecker then extracts the CPE string from the found CPE item. All CPE values are stored in the cpes.json file to speed up future runs of the tool.

SwiftDependencyChecker then queries the public NVD Application Programming Interface (API) for each CPE value. The NVD API returns all vulnerabilities connected to the given CPE value. Vulnerability data is parsed and saved in the vulnerabilities.json file. The CVE data includes values (among others) such as vulnerability description, impact, who reported the vulnerability, and affected versions. Data on affected versions is a tree with CPEs of affected versions and operators (such as "and" and "or") that define which combinations of versions are vulnerable. Currently, the operator value is ignored and all linked CPEs are considered vulnerable.

Matches between the used library versions and vulnerable library versions are determined by first matching CPEs. As a next step, SwiftDependencyChecker tries to match library versions. Library versions can be matched if versions are given as numbers separated by dots. Library matching can for example ignore "v" as a prefix and "-beta" as a suffix. Other kinds of special values are ignored as well and the library version is considered vulnerable. Library versions are compared

[14] `https://nvd.nist.gov/products/cpe`



```
import Foundation
import UIKit
import GRPC  3 ⚠  en: HTTP2ToRawGRPCServerCodec in gRPC Swift 1.1.1 and earlier allows remote attackers to deny se

class ViewController: UIViewController {
```

Figure 9. Integration of vulnerability detection into Xcode (screenshot)

by going through major, minor, and revision values of the version, first comparing the major version, then the minor version and last the revised version. If the versioning scheme of a library has changed during the lifetime of a library, such as going from version 2018.1.4 to 3.4.1, SwiftDependencyChecker may report false positive results.

After uses of vulnerable library versions are determined, SwiftDependency-Checker traverses all Swift files and all package manager resolution files to find references to vulnerable library versions. For each match, a warning is printed including the file path, line number, and warning message. If SwiftDependency-Checker is run as a build script under build phases, Xcode displays these warnings in the source code editor as shown in Figure 9.

4.3.3. Usage

SwiftDependencyChecker is a lightweight tool that can be easily installed and integrated into the Xcode build process.

SwiftDependencyChecker can be installed through homebrew with the following commands:

```
brew tap kristiinara/SwiftDependencyChecker
brew install SwiftDependencyChecker
```

SwiftDependencyChecker can either be run from the command line or it can be added as a build script under build phases in Xcode. The build script in Xcode should include the following line:

```
SwiftDependencyChecker analyse --action sourceanalysis
```

When the project is built in Xcode then the source code analysis is run by SwiftDependencyChecker. The tool outputs warnings about files that contain references to library versions with known vulnerabilities. These warnings are shown in the source code editor of Xcode as can be seen in Figure 9.

4.3.4. Evaluation: Functional Correctness

Functional correctness testing was performed on real-world apps that contained references to vulnerable third-party dependencies. To identify examples of such applications we used GraphifyEvolution [RP21] to analyze dependencies used in open-source iOS applications. We implemented a new external analyzer that was able to parse package manager resolution files for Cocoapods, Carthage, and Swift Package Manager.

We used a collaborative list of open-source iOS applications[15] that contains a total of 1318 projects. We first filtered out applications where the app metadata contained an iTunes link, which indicated that these apps were published on the app store. This was done to discard demo applications that were not meant for production use as many toy applications are published on GitHub. GraphifyEvolution was then run on the list of 374 applications of which 365 applications were successfully analyzed. All used library names and versions were entered into the Neo4J database. Of the analyzed applications 133 had at least one dependency.

We then queried the list of used library names from the database. We downloaded the CPE dictionary used by NVD and wrote a script that searched for a matching CPE value for each of these libraries. For each CPE found the script queried the NVD API to find vulnerabilities related to the given CPE value. Vulnerability data received from the NVD API was stored in a JSON file.

As a next step, we went through the received vulnerability data and manually matched vulnerable library versions to library versions used by the analyzed open-source iOS apps. We found 16 uses of third-party library versions with a publicly reported vulnerability. For each match, we then went to the repository page of the open-source library and checked if the repository is still maintained. Of these found matches we found four open source applications that were still maintained and published on the app store. We contacted the developers of these applications to inform them about the vulnerable dependencies in their projects. Then, for each of these apps we cloned the repository and ran SwiftDependencyChecker. We verified that SwiftDependencyChecker successfully identified all uses of vulnerable libraries that we were able to find manually. So far none of the developers have updated these vulnerable dependencies.

We did not check for false positives. False positives can only happen if a sub-target of a library is referenced and the published vulnerability only affects another part of the library, as the analysis ignores sub-targets.

4.3.5. Evaluation: Performance

The four applications with vulnerable third-party dependencies that we identified previously were also used when testing the performance of SwiftDependency-Checker. These applications were a good fit for performance testing because they included a high number of dependencies and analyzing them ought to be more time-consuming than smaller applications.

We analyzed how long it takes to run SwiftDependencyChecker in three scenarios:

1. analyzing a new project.
2. analyzing a project that has been analyzed before

[15]`https://github.com/dkhamsing/open-source-ios-apps/tree/eb43f2d00158579fe2d8710b8e738b887c26ac6d`

We selected these scenarios because SwiftDependencyChecker caches results between runs in order to speed up analysis. The recommended use of SwiftDependencyChecker is to include it as a build script in Xcode, meaning that long run times would not be appreciated because it slows down development.

The performance of SwiftDependencyChecker was measured on a 2020 Macbook Air with the M1 processor and 16GB of RAM running macOS 12.1. This is a typical computer that could be used for iOS development.

Running SwiftDependencyChecker on each of the four applications for the first time took 8.5 to 13 minutes. The variation stems from the number of dependencies included in a project. We assumed that a run conducted the first time after installation of the tool might take some extra time for downloading the CPE dictionary and the CocoaPods Spec repository. However, it turned out that the difference was negligible.

Rerunning the analysis of a project that had previously been analyzed took between 0.07 and 0.08 seconds. In other words, repeated runs were very fast and small enough for the recommended usage of SwiftDependencyChecker at every build during development.

Running SwiftDependencyChecker for the first time and analyzing a new project was relatively slow and might discourage developers from using the tool. As a possible solution, we are currently looking into the possibility of including JSON files with CPE and spec translation values for common libraries.

4.3.6. Evaluation: Novelty

Similar tools exist and are quite popular for other languages and platforms. In this subsection, we will mainly discuss tools that provide some support for either CocoaPods, Carthage, or Swift Package Manager and therefore overlap with our tool.

DependencyCheck[16] is a tool developed by OWASP that provides experimental support for CocoaPods and Swift Package Manager. It analyses the package manager resolution files and queries data from NVD and Sonatype. DependencyCheck can generate XML and HTML reports. There is no easy way to integrate analysis results with Xcode.

Snyk CLI[17] is developed by Snyk. The tool provides support for CocoaPods and queries the Snyk vulnerability database. The Snyk CLI did not detect all vulnerabilities found by DependencyCheck and SwiftDependencyChecker. Snyk CLI is a commercial tool that requires sign up through the Snyk web page. There is no easy way to integrate analysis results with Xcode.

FOSSA[18] provides an enterprise solution that can detect vulnerable third-party dependencies included through among others CocoaPods, Carthage, and Swift

[16]`https://jeremylong.github.io/DependencyCheck/analyzers/swift.html`
[17]`https://docs.snyk.io/features/snyk-cli/`
[18]`https://fossa.com/product/open-source-security-management`

Package Manager. Analysis for FOSSA requires an API key and analysis is done on either the cloud or in a separately set up client-server.

Veracode SCA[19] (Source Composition analysis) is a tool developed by Veracode that analyses whether dependencies used in a project match vulnerable library versions and checks where and if the vulnerable library is used in the project. Veracode SCA provides support for CocoaPods. Finding vulnerable libraries based on library names is done in the cloud, matching uses to project code is done offline. The analysis tool is commercial and there is no direct integration with Xcode.

While existing tools provide at least partial support for the three package managers targeted by SwiftDependencyChecker, none of them can be directly integrated into Xcode. For other platforms, similar tools exist. For example, Dependency-Check-Gradle[20] makes it possible to integrate dependency checking as a gradle plugin in Android projects and the Snyk Vulnerability Scanner[21] allows integrating detection of vulnerable libraries in Visual Studio.

4.3.7. Evaluation: Usefulness and Usability

We posted links to SwiftDependencyChecker to Linkedin, Reddit, and Medium and asked iOS developers to try it out. Overall, the feedback we received confirmed the usefulness of the tool. Some developers made improvement suggestions. One concern was that the tool seemed to be unresponsive when run for the first time after installation (cf. Section 4.3.5). We took the feedback into account and improved logging. In addition, we will consider how the initial run of the tool could be sped up by including translation and CPE files.

4.3.8. Application

We used SwiftDepnendencyChecker in combination with GraphifyEvolution to generate the Swift LDN dataset described in 6.1.2. We used this dataset to analyze the Swift LDN. These results are described in detail in Chapter 6 and used to answer the research questions 2.1, 2.2, 2.3, and 2.4.

[19] `veracode.com`

[20] `https://github.com/dependency-check/dependency-check-gradle`

[21] `https://marketplace.visualstudio.com/items?itemName=snyk-security.snyk-vulnerability-scanner`

5. CODE SMELL ANALYSES

Code smells are recurring patterns in code that have been identified as bad practices [Fow18]. They have been analyzed extensively, for example, in Java desktop applications [KDG09][Moh+09][Olb+09]. For mobile applications, most of the research has been done for Android with very little research done for iOS. Although Android has the largest market share, iOS is a very popular platform. Our goal is to understand code smells in iOS applications.

5.1. Method

In this section, we describe the three research questions and the corresponding research methodology.

5.1.1. Research Questions

To better understand code smells in iOS applications we first analyze the distribution and frequency of code smells in iOS applications by analyzing how many applications are affected by each code smell and by calculating the proportion of each code smell type of the number of all code smells. We then analyze how this compares to Android applications and lastly, we look into how code smells in iOS applications evolve over time. We answer the following research questions:

- RQ1.1: What is the distribution and frequency of code smells in iOS applications?
- RQ1.2: How do code smells in iOS applications compare to Android applications?
- RQ1.3: How have code smells in iOS applications evolved over time?

The answers to these research questions help bring attention to Swift developers for which code smells they should watch out for when developing iOS applications. Additionally the comparison of code smells in iOS and Android helps tool developers understand if and how code smell analysis tools should be adapted for iOS developers.

5.1.2. RQ1.1: Code Smells in iOS applications

Fowler defined 22 object-oriented code smells in his book "Refactoring: Improving the Design of Existing Code" [Fow18]. Mannan et al. [Man+16] used a commercial tool called InFusion to detect 21 object-oriented code smells in Android applications. Habchi et al. [Hab+17] used a tool called PAPRIKA, which was developed to detect code smells in Android applications, to analyze three iOS-specific and four object-oriented code smells in iOS applications. We combined these three lists of code smells into a single list of 34 object-oriented code smells and two iOS-specific code smells. The descriptions of the code smells and the four excluded code smells are presented in A.2. We created a tool called GraphifySwift that analysis Swift code, generates a model of this code, and enters it into a Neo4J graph database. This tool is described in detail in Section 4.1.

A shortcoming of GraphifySwift in comparison to PAPRIKA is that we are not able to analyze compiled applications. With Android, it is possible to decompile an APK, while iOS applications are encrypted which makes them more difficult to reverse engineer. Due to these limitations, we limited our scope of application to open-source applications written in Swift. Habchi et al. [Hab+17] used a collaborative list of open-source iOS applications from GitHub[1]. The collaborative list of open source applications is assembled by open source contributors and contains information for each application added such as application name, repository address, the language used and app store link if applicable. We downloaded open-source iOS applications written in Swift from this list of applications.

Since this list is collaborative in nature it has been updated, old applications have been removed and new applications added. For our analysis, we used the 2019-11-13 version of the list of open-source iOS applications[2]. We downloaded iOS applications written in Swift from the collaborative list. We downloaded 454 open-source iOS applications from GitHub. For some applications in the collaborative list, the source code was no longer available or we were not able to access the needed dependencies. We were able to successfully analyze 273 iOS applications written in Swift.

In the following, we refer to two different lists of applications:

- **all_new_apps** refers to all 273 successfully analyzed applications from the collaborative list of open source applications
- **appstore_new_apps** refers to 68 applications from all_new_apps that are available on the app store

We ran our analyses on these lists of applications. First, our tool analyzed these applications and entered the corresponding data into the graph database. Next, we calculated thresholds for code smells such as a very high number of instructions using the box-plot technique. According to the box-plot technique thresholds are calculated as $Q3 + 1.5 * IQR$ where $Q3$ is the third percentile and IQR is the interquartile range. Established thresholds based on the all_new_apps list are listed in Appendix A.1. Using these thresholds we were able to update and execute code smell queries. These code smell queries are listed in Appendix A.3.

We then executed code smell queries to find code smells in these iOS applications. Using the list of code smells we plot the distribution and frequency of code smells in iOS applications.

[1]`https://GitHub.com/dkhamsing/open-source-ios-apps/tree/4d62674195711c580166d9e7859f82e30695dcf2`

[2]`https://GitHub.com/dkhamsing/open-source-ios-apps` accessed: 2019-11-13

5.1.3. RQ1.2: Code Smells in iOS vs Android

For the analysis of Android apps, we took the list of apps provided by Habchi et al. [Hab+17]. Since the list only included app package names, we queried AllFreeAPK API[3] to find and download these apps. We decided to search AllFreeAPK instead of GitHub, as PAPRIKA uses APKs for analysis, and this way we were able to skip the step of compiling these apps. Later during the analysis we needed to discard some of the very big apps (apps with more than 100 classes) due to performance issues. In total, we included 694 open-source Android apps in our analysis.

For these Android apps, we used PAPRIKA to populate the Neo4J database. We then took the queries defined for GraphifySwift (listed in Appendix A.2) to find code smells. Since GraphifySwift was built to analyze iOS apps we had to adapt the code smell queries so that they could be used on the database produced by PAPRIKA. We made the following changes to the code smell queries:

We removed references to Module nodes, i.e., the relationship

```
(app)-APP_OWNS_MODULE->(module)-
        MODULE_OWNS_CLASS->(class)
```

was substituted by the relationship

```
(app)-APP_OWNS_CLASS->(class)
```

We removed references to argument type or substituted them with argument name. Argument names are not accessible in Java bytecode and therefore the argument name provided by PAPRIKA is actually the argument type. Finally, we added the relationship

```
(variable|argument)-IS_OF_TYPE
                ->(class)
```

by finding classes whose names matched the argument name or variable type.

After these modifications of the database and queries, 19 of the 34 GraphifySwift code smell queries could be used on the Android app database produced by PAPRIKA. The adapted code smell queries are available on FigShare[4]. The code smell queries that had to be excluded contained metrics or attributes that were not provided by PAPRIKA. We excluded for example queries referring to code duplication, maximum nesting depth, number of switch statements, and number of comments. For the analysis of Android apps, we calculated new thresholds based on the apps that we analyzed using the box-plot technique. The list of iOS and Android thresholds is included in Appendix A.1.

Using the results from the code smell queries, we checked whether any of the 19 identified code smells occurred in at least one app on each platform. Then, we calculated the densities of code smells for Android apps and compared these to

[3]`https://m.allfreeapk.com/api/`

[4]`https://figshare.com/articles/conference_contribution/GraphifySwift_queries_adapted_for_PAPARIKA_for_Android_code_smell_analysis/13102994`

iOS. Code smell density was calculated by counting the number of code smells (total and per code smell type) and dividing by the number of app instructions. The number of app instructions here is the number of class instructions found by the SourceKittenFramework for each class added together. We use instructions instead of number of lines of code as the number of instructions ignores white-space and comments. Additionally the number of instructions is readily available through the code parsing library used.

Lastly, we calculated the relative frequencies of code smells per code smell type on each platform. We counted the code smells of a type in all apps and divided by the total code smell count. We did this per platform. To calculate the code smell distributions on the app and class levels per platform, we counted how many apps (and classes) contain at least one code smell of a certain type and then divided it by the total number of apps (and classes).

5.1.4. RQ1.3: Code Smell Evolution

For a preliminary evaluation, we analyzed the code smell evolution of 30 open-source iOS apps. We chose apps that were open source and whose repositories contained all the required code to run the application. This means we used apps that either did not use any dependency management or used Carthage and committed all the dependency files. Future versions of the tool will support other dependency management solutions, such as CocoaPods and Git submodules.

We used GraphfiyEvolution to analyze the evolution of these applications. As external analyzers, we used code smell analysis and security analysis with indidersec. For a preliminary analysis, we then queried the LongMethod code smell in all versions of these applications. Additionally, we analyzed when potential vulnerabilities were introduced.

5.2. Results

We present our results on Code Smell analysis in the following section.

5.2.1. RQ1.1: Code Smells in iOS applications

We analyzed 273 open-source iOS applications written in Swift using our code smell detection tool which is able to detect 34 object-oriented and two iOS-specific code smells. When looking at applications that have at least one occurrence of a given code smell, iOS applications are most often affected by Lazy Class, Long Method, Message Chain, Ignoring Low Memory Warning, and Data Class as can be seen in Figure 10.

The distribution of code smells can be seen in Figure 11. The most common code smell is Internal Duplication, followed by Lazy Class, Long Method, Message Chain, and Primitive Obsession. This distribution was established by counting occurrences of a given code smell through all applications.

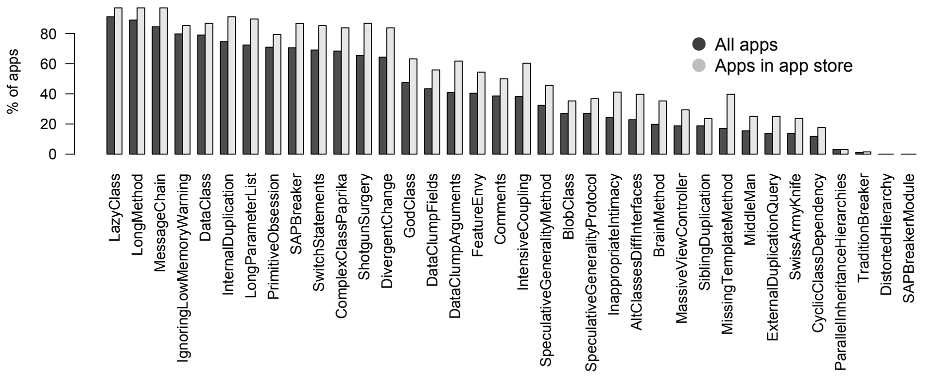


Figure 10. Percentage of apps with at least one code smell of a given type

We also compared the distribution of code smells in applications that can be found on the App Store (applications in list appstore_new_apps) to all open-source applications analyzed. As shown in Figure 11 the distribution of code smells is very similar, the biggest difference (by two percentage points) seems to be Internal Duplication. It could be that applications in the App Store are slightly better maintained and less copy-pasting is allowed.

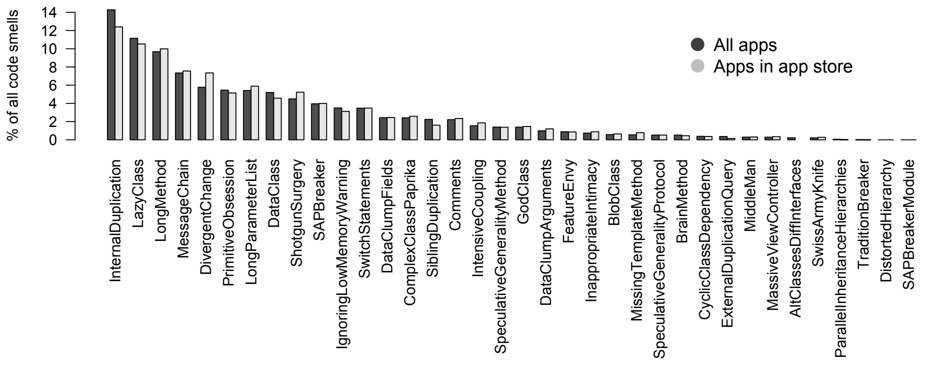


Figure 11. Comparison of the distribution of code smells in applications on the App Store

5.2.2. RQ1.2: Code Smells in iOS vs Android

We analyzed 694 open-source Android apps using PAPRIKA and modified code smell queries from GraphifySwift to answer our research question. We analyzed the apps with regards to 19 code smells BlobClass, ComplexClass, CyclicClass-Dependency, DataClass, DataClumpFields, DistortedHierarchy, DivergentChange, InappropriateIntimacy, LazyClass, LongMethod, LongParameterList, MiddleMan, ParallelInheritanceHierarchies, PrimitiveObsession, SAPBreaker, ShotgunSurgery, SpeculativeGeneralityProtocol, SwissArmyKnife and TraditionBreaker. When comparing the occurrence of code smells on each platform, we found that 18 of the 19 identified code smells occurred in apps on both platforms, i.e., Android

	mean	std	min	25%	50%	75%	max
BlobClass	0.9	3.4	0.0	0.0	0.0	1.0	48.0
ComplexClass-Paprika	3.7	11.1	0.0	0.0	1.0	3.0	149.0
DataClass	8.0	18.8	0.0	1.0	3.0	7.0	228.0
DataClumpFields	3.7	13.1	0.0	0.0	0.0	2.0	159.0
DistortedHierarchy	0.0	0.0	0.0	0.0	0.0	0.0	0.0
DivergentChange	8.9	29.4	0.0	0.0	2.0	7.0	375.0
Inappropriate-Intimacy	1.1	3.9	0.0	0.0	0.0	0.0	38.0
LazyClass	17.2	38.8	0.0	2.0	7.0	17.0	510.0
LongMethod	14.9	36.4	0.0	2.0	5.0	12.0	377.0
LongParameterList	8.4	25.5	0.0	0.0	3.0	7.0	351.0
MiddleMan	0.5	1.8	0.0	0.0	0.0	0.0	19.0
ParallelInheritance-Hierarchies	0.1	0.8	0.0	0.0	0.0	0.0	12.0
PrimitiveObsession	8.4	17.9	0.0	0.0	2.0	8.0	130.0
SAPBreaker	6.1	13.6	0.0	0.0	2.0	6.0	147.0
ShotgunSurgery	6.9	19.7	0.0	0.0	2.0	5.0	241.0
SpeculativeGenera-lityProtocol	0.8	2.8	0.0	0.0	0.0	1.0	35.0
SwissArmyKnife	0.3	1.5	0.0	0.0	0.0	0.0	18.0
TraditionBreaker	0.0	0.1	0.0	0.0	0.0	0.0	1.0

Table 4. Description of code smell data for iOS applications

and iOS. Code smell DistortedHierarchy never occurred in iOS apps.

Tables 4 and 5 show the mean, standard deviation, percentiles and extremes for each code smell type for iOS and Android applications, respectively. We used the Mann–Whitney U test with a Bonferroni corrected significance threshold of 0.0027 to test if the distribution of each code smell differs between iOS and Android. We saw that the difference is significant for code smells ComplexClassPaprika ($p < 0.001$), DataClass ($p < 0.001$), DistortedHierarchy ($p < 0.001$), LongMethod ($p < 0.001$), LongParameterList ($p < 0.001$), MiddleMan ($p < 0.001$), PrimitiveObsession ($p < 0.001$), ShotgunSurgery ($p < 0.001$), SpeculativeGeneralityProtocol ($p < 0.001$), and SwissArmyKnife ($p < 0.001$). The difference was not significant for code smells BlobClass ($p = 0.013$), DataClumpFields ($p = 0.668$), DivergentChange ($p = 0.044$), InappropriateIntimacy ($p = 0.004$), LazyClass ($p = 0.392$), ParallelInheritanceHierarchies ($p = 0.618$), SAPBreaker ($p = 0.802$), and TraditionBreaker ($p = 0.565$).

The results of our code smell density analysis are shown in Figure 12. Accumulated over all code smells it turned out that the apps on the iOS platform had a density of 41.7 smells/kilo-instructions while the apps on Android only had

	mean	std	min	25%	50%	75%	max
BlobClass	0.9	2.1	0.0	0.0	0.0	1.0	27.0
ComplexClass-Paprika	1.9	3.6	0.0	0.0	1.0	2.0	37.0
DataClass	0.1	0.8	0.0	0.0	0.0	0.0	12.0
DataClumpFields	2.7	10.1	0.0	0.0	0.0	2.0	184.0
DistortedHierarchy	1.1	3.1	0.0	0.0	0.0	0.0	42.0
DivergentChange	5.4	16.1	0.0	0.0	1.0	5.0	327.0
Inappropriate-Intimacy	1.6	5.5	0.0	0.0	0.0	1.0	111.0
LazyClass	9.7	10.3	0.0	3.0	6.0	13.0	81.0
LongMethod	25.4	49.2	0.0	3.0	11.0	30.0	782.0
LongParameterList	16.3	29.9	0.0	2.0	7.0	21.0	448.0
MiddleMan	0.0	0.1	0.0	0.0	0.0	0.0	1.0
ParallelInheritance-Hierarchies	0.1	0.6	0.0	0.0	0.0	0.0	10.0
PrimitiveObsession	4.0	15.0	0.0	0.0	1.0	4.0	329.0
SAPBreaker	5.7	8.7	0.0	0.0	2.0	7.0	51.0
ShotgunSurgery	14.0	26.8	0.0	1.0	5.0	17.0	410.0
SpeculativeGenera-lityProtocol	0.2	0.8	0.0	0.0	0.0	0.0	14.0
SwissArmyKnife	0.1	0.4	0.0	0.0	0.0	0.0	4.0
TraditionBreaker	0.0	0.3	0.0	0.0	0.0	0.0	4.0

Table 5. Description of code smell data for Android applications

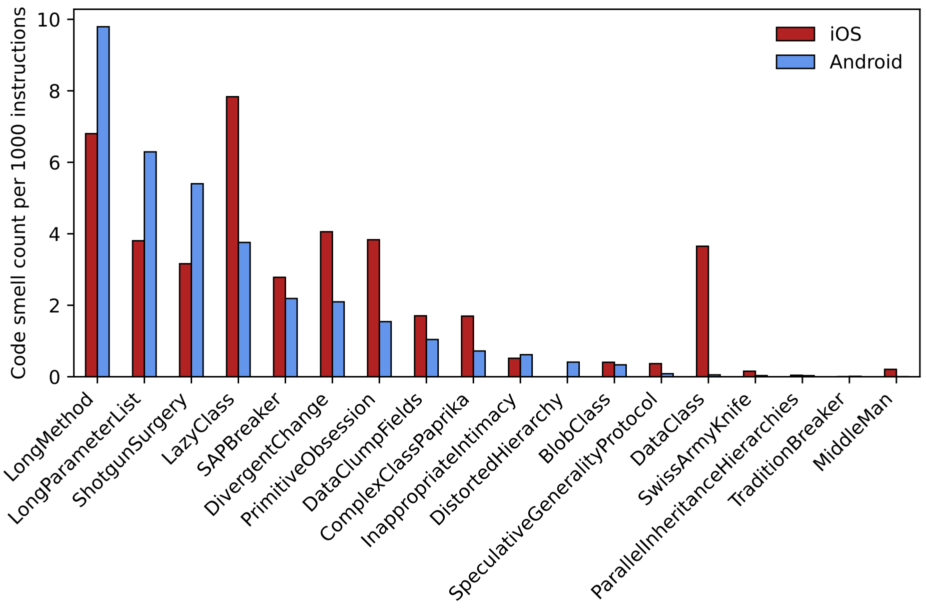


Figure 12. Comparison of code smell densities between Android (blue) and iOS (red) apps

a density of 34.4 smells/kilo-instructions. Moreover, it can be seen from Figure 12 that the code smell densities differ between iOS and Android. Code smells in iOS applications have a smaller smaller variance (Gini index 0.55) compared to code smells in Android applications (Gini index 0.67). Code smells LazyClass, DivergentChange, PrimitiveObsession, and DataClass had a particularly high density in iOS apps (with 7.8, 4.1, 3.8, and 3.7 per 1000 instructions, respectively). On the other hand, code smells LongMethod, LongParameterList and Shotgun-Surgery were clearly more frequent in Android apps (with 9.8, 6.3, and 5.4 per 1000 instructions, respectively).

Figure 13 shows the relative frequency of code smell occurrences over all apps on the Android platform (blue bars) and the iOS platform (red bars). The results confirm what we had seen when we compared code smell densities: the proportions of code smells differ between platforms. In addition, we see that code smells are more evenly distributed in iOS apps (Gini index 0.55) as compared to Android apps (Gini index 0.67).

Then we analyzed how large the share of smelly apps on each platform is and how large the share of smelly classes is on each platform. We did these analyses for each code smell type separately. Figures 16 and 17 show the percentages of apps and classes, respectively, containing code smells of a certain type.

We found that the percentages of smelly apps are relatively similar between platforms (with Gini index 0.38 for iOS and 0.43 fro Android). The biggest differences occur for code smell DataClass (79% of iOS apps have at least one af-

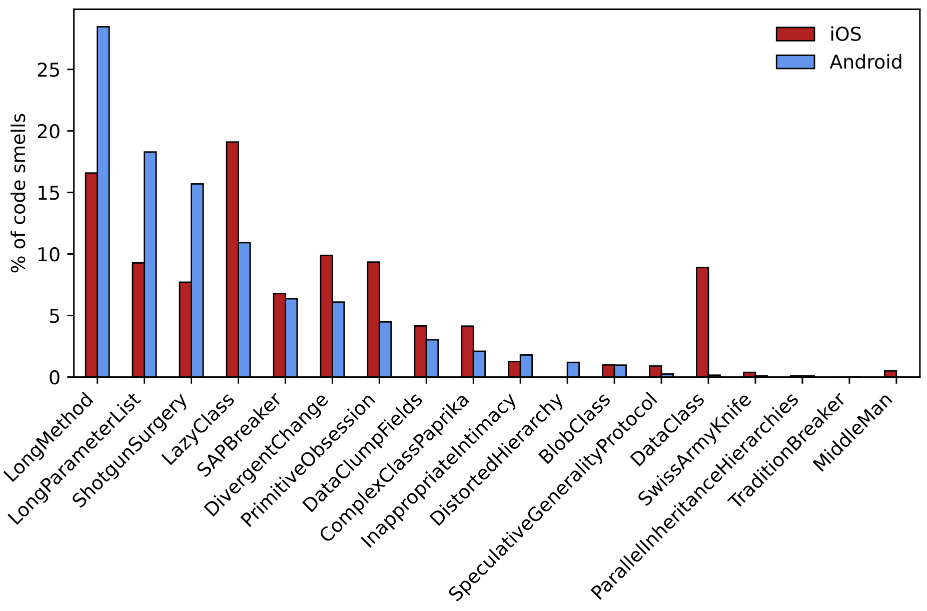


Figure 13. Code smell proportions on Android (blue) and iOS (red)

fected class while only 7% of Android apps are affected), MiddleMan (15% of iOS apps are affected but only 1% of Android apps), and DistortedHierarchy (25 % of Android apps are affected but none of the iOS apps is). The distribution of number of code smells per application is shown in Figure 14 and in Figure 15. For some code smells like LongMethod and LongParameterList the number of code smells per application is more evenly distributed in Android applications than in iOS applications. This means that a higher percentage of iOS applications has a lower number of smells of the the given type. For code smells like LazyClass and SAPBreaker this is opposite and there is a higher concentration of Android applications with a small number of these smells.

In addition, we analyzed the occurrence of the method-based code smells LongMethod and LongParameterList separately. We found that in iOS apps 9% of methods are considered LongMethod while this is the case for 14% of the methods in Android apps. In iOS apps 5% of the methods have a LongParameterList while this is the case for 9% of methods in Android apps.

5.2.3. RQ1.3: Code Smell Evolution

In total, we analyzed 30 applications, 3878 app versions, 7086 classes, 7965 methods, and 7794 variables. The analysis took 17 hours, this analysis included code smell analysis, duplication analysis, and vulnerability analysis with insider. Data generated can be downloaded from the tool web page [5] and browsed using the

[5] `https://github.com/kristiinara/GraphifyEvolution/blob/master/example_data/overview.md`

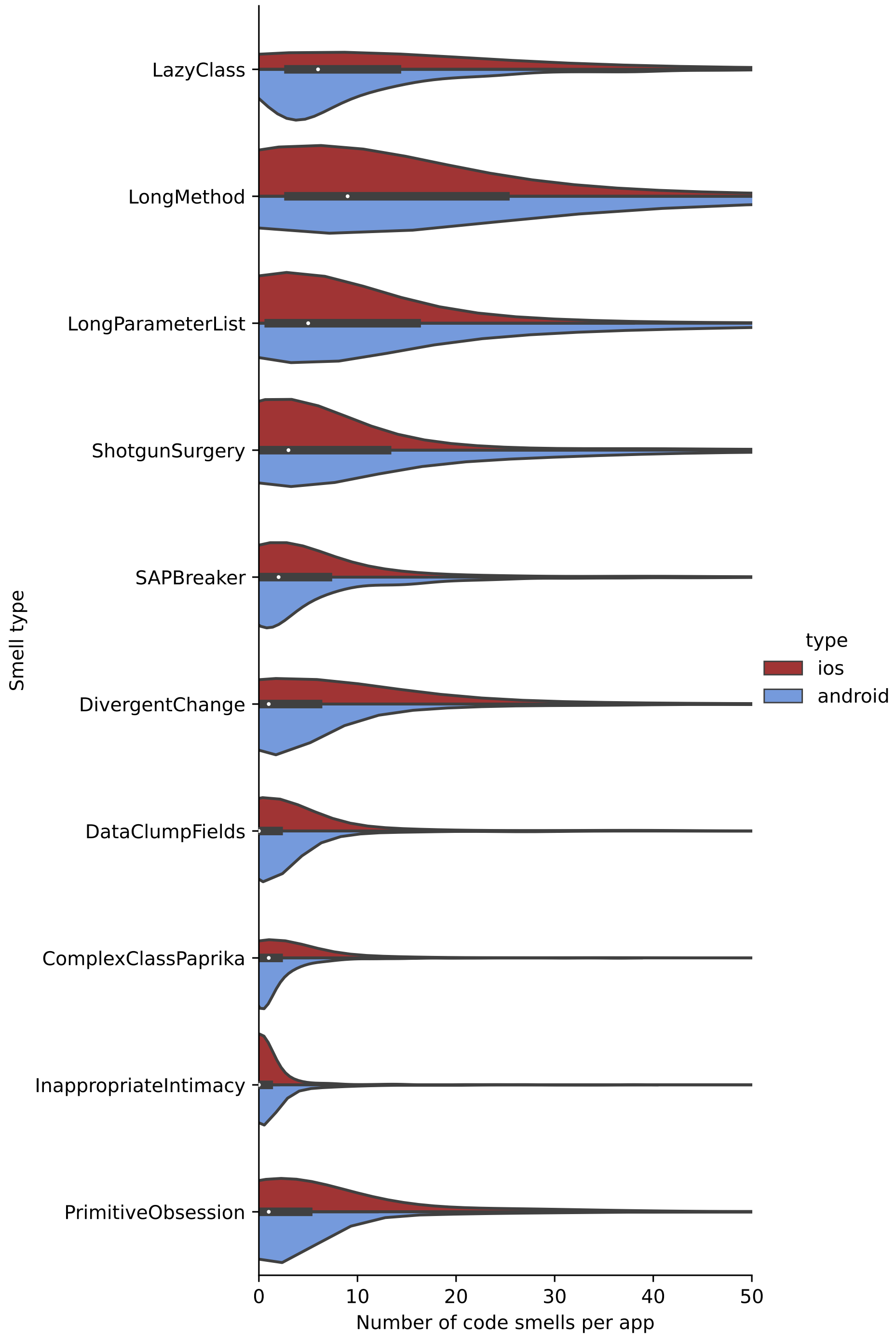


Figure 14. Distribution of number of code smells in Android (blue) and iOS (red) applications

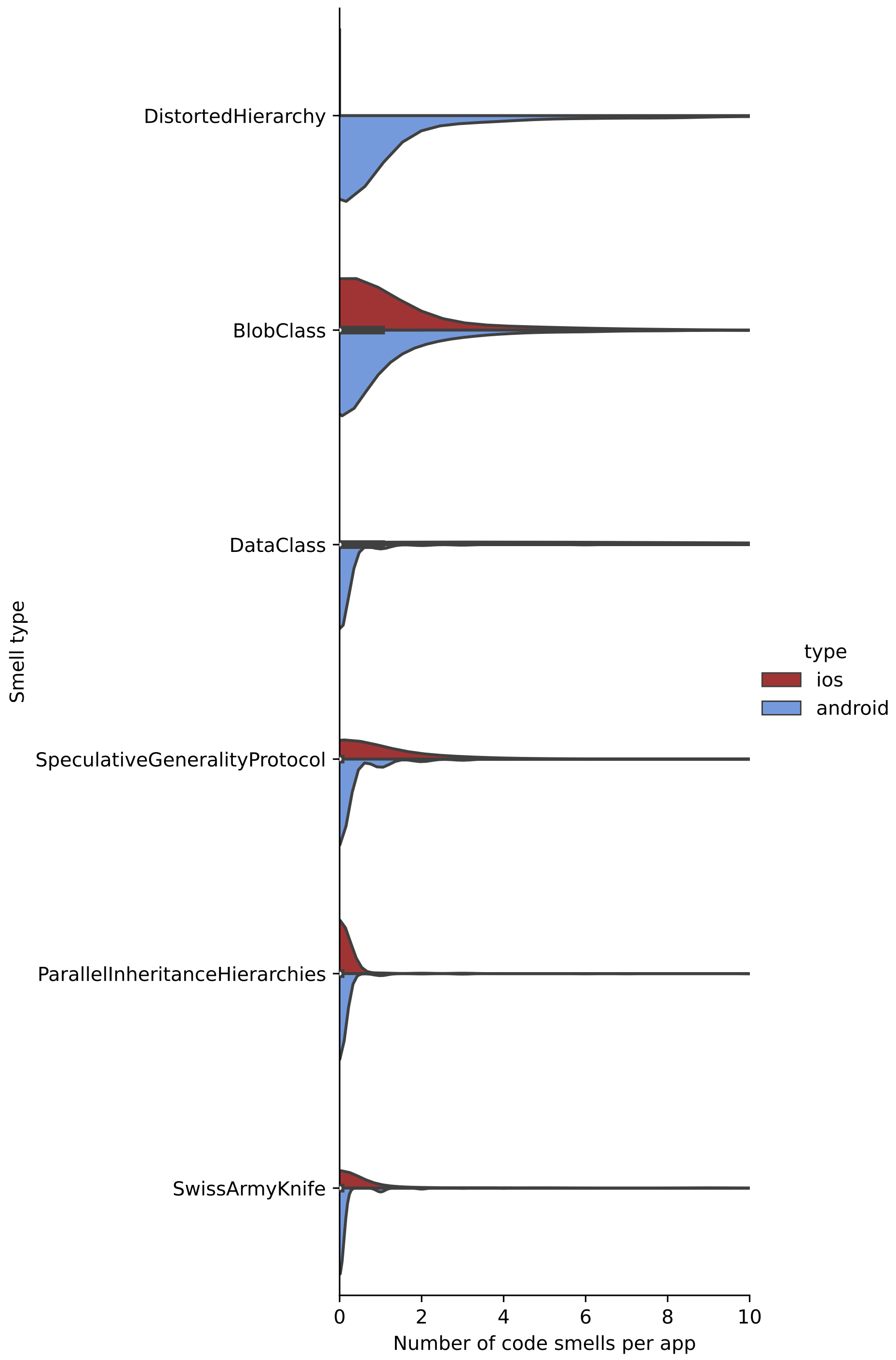


Figure 15. Distribution of number of code smells in Android (blue) and iOS (red) applications (continued)

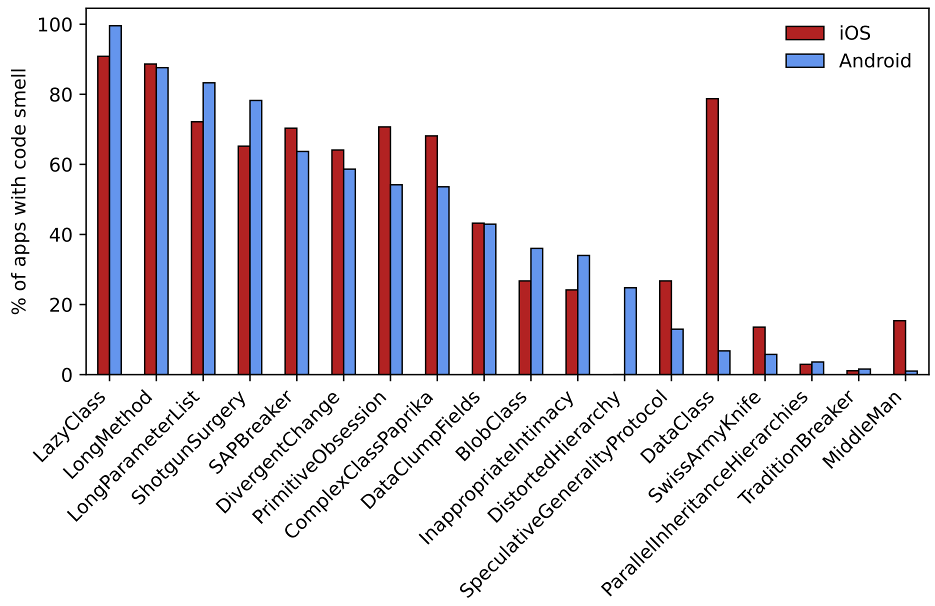


Figure 16. Comparison of code smell frequencies on app level between Android (blue) and iOS (red)

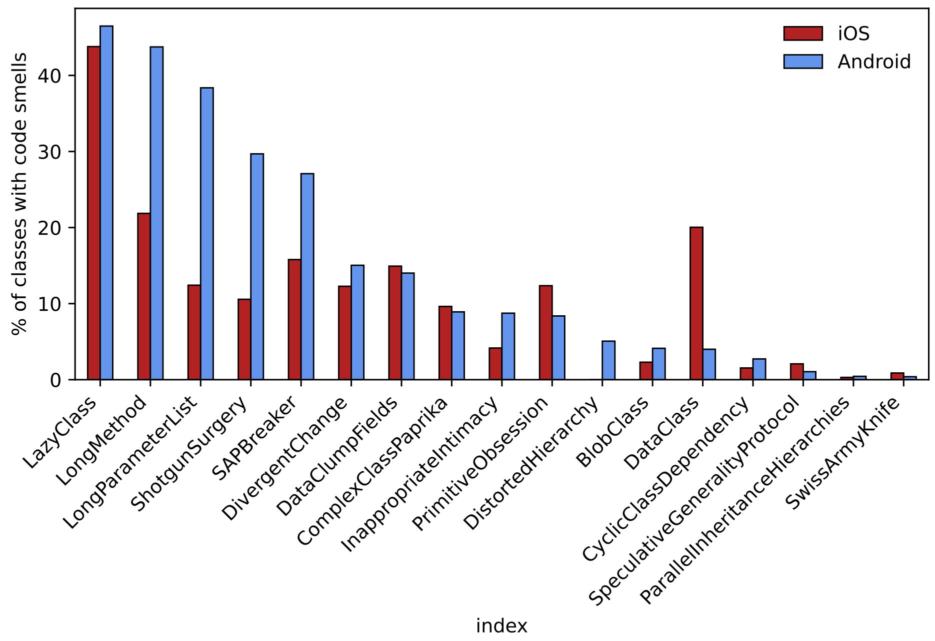


Figure 17. Comparison of code smell frequencies on the class level between Android (blue) and iOS (red)

Neo4j community version.

In the following we describe three examples of different kinds of information we can extract from this Neo4j database using cypher queries.

Firstly, as an example, we choose the Tweetometer app which has 373 analyzed app versions and 19 long methods. We query all app versions with the name "Tweetometer" and count long method instances that are connected to these app versions. We then save the results as a .csv file and plot the results using R. Then we observe how the number of long methods grew in the first half of the project development, then plateaued. In the end, almost all long methods were removed.

Secondly, we analyze if methods were created as long methods or if they became too long over time. We queried the count of changes before a method became a long method. By count of changes we mean the number of commits that altered the given method. The result of this query shows that of 158 unique long method instances, 131 methods were too long when they were added. Ten methods became too long after one and two changes, four became too long after three changes and one became too long after three, five, and six changes, each.

Lastly, we ran app analysis with the InsiderSecAnalyser enabled, which saved vulnerabilities into the application database as nodes and added relationships to vulnerable classes and methods. We queried class changes where a vulnerability was removed and collected commits for each application where this change occurred. We found removed vulnerabilities in three applications. In the Arex application, two vulnerabilities were removed in the same class during two commits. In iCepa application, two vulnerabilities were removed in two different classes during two different commits. In the Tweetometer application, one vulnerability was removed.

Example queries used to extract this data are described in Appendix B.3.

5.3. Discussion

We discuss our results on Code Smell analysis in the following section.

5.3.1. RQ1.1: Code Smells in iOS applications

We identified the most common code smells in iOS applications. In terms of the percentage of applications affected by a specific code smell the five most common code smells are Lazy Class, Long Method, Message Chain, Ignoring Low Memory Warning, and Data Class. In terms of the total number of instances per code smell type, the five most common code smells are Internal Duplication, Lazy Class, Long Method, Message Chain, and Primitive Obsession. This shows that the analyzed open-source Swift applications have many small classes and a few large classes with long and perhaps complicated methods. For developers and educators, this means that it makes sense to pay more attention to balancing the complexity of code.

Different from what is often seen in iOS developer blogs, Massive View Controller was one of the least common code smells. Under 18% of the applications were affected by this smell. At the same time, the Massive View Controller smell is very often discussed in the developer community [Mir20; Hud19; Kha15; DeL17; tea19; Law19]. It might be that being aware of this possible smell developers try to actively avoid introducing it. If this is true, it would be beneficial to give more attention to more common code smells, such as Internal Duplication, Long Method, or Message Chain.

One of the most common code smells, IgnoringLowMemoryWarning, occurs when developers miss to implement a low memory warning method that is called when the device is low on memory. This method allows the app time to reduce its memory footprint to avoid being shut down. In some applications it might not be possible to deal reasonably with a low memory alert. But any application that uses or cashes a slightly smaller amount of data (which might just be images in a table view), it makes sense to implement the method. Regardless of being well documented in official documentation our results show that most application have at least one view controller where the implementation is missing. This can lead to poor user experience when using a smelly application.

Many applications having a single LongMethod code smell instance is not a big problem as a longer method is sometimes warranted. LongMethod being one of the most often occurring code smells, however, shows that there is a need to educate app developers on better practices. Code smells like LongMethod, MessageChange and LongParameterList make application code difficult to read and modify.

The most frequent code smell InternalDuplication is something that should be easily preventable if following best practices on code reuse. Duplicated code can lead to maintainability problems as changes would require editing code in multiple places. Duplicated code might not affect user experience directly, but worsen the effort needed to maintain the application.

Overall our results show that developers of open source iOS applications could increase the quality of their applications in terms of both maintainability and usability when applying code smell detection tools. This gives incentive to continue developing our tool set to imporove tool support for Swift developers.

5.3.2. RQ1.2: Code Smells in iOS vs Android

We saw that the code smell distribution can vary a lot between different mobile app platforms. Same code smells can occur on both platforms but different code smells are most common. During our analysis, we discovered that contrary to the assumption by Mannan et al. [Man+16], not all object-oriented code smells occur in all object-oriented languages. For example, RefusedParentBequest, which was detected by Infusion and used for Android analysis in [Man+16], was excluded from the list of code smells for GraphifySwift as it does not apply to Swift.

Namely, there is no *protected* keyword in Swift.

The only code smell that did not occur in iOS applications was DistortedHierarchy. DistortedHierarchy finds chains of inheritance that are too deep. The lack of this smell is a possible indication that deep inheritance trees do simply not corresponds to common patterns when writing applications in Swift. A recommendation by Apple is to use immutable structures instead of classes where possible. Given that structures cannot inherit from parent structures their use might make the occurrence of the DistortedHierarchy smell less likely.

We saw that the frequencies and proportions of code smells in iOS and Android differ. But contrary to previous results they are not always higher in Android apps. Some code smells, such as DataClass, LazyClass, MiddleMan, PrimitiveObsession, and SpeculativeGeneralityProtocol are more common in iOS apps. This means that apps on one platform are not necessarily smellier than on the other platform, but different proportions of code smells might be caused by different programming paradigms and frameworks used.

This result should be interesting to developers that move from one platform to the other. These kinds of results show that we have to pay attention to different issues on different platforms. On Android, we have to be more careful to not write too long and complex methods. On iOS, especially with protocol-oriented programming, it makes sense to consider if we generalize too much and create too general protocols or too many small classes.

Our results show that tool developers need to take into account differences between platforms as developers on iOS and Android need support with different code smells. This results supports the need for dedicated tool support for Swift developers.

5.3.3. RQ1.3: Code Smell Evolution

The small preliminary study on code smell evolution showed that most too-long methods that were added were already too long at creation. This corresponds to the findings of Chatzigeorgiou et al. [CM14] that smelly code is introduced when the corresponding artifact is created.

5.4. Threats to Validity

Internal Validity: In our case, internal validity might be affected by how our code smell tool detects code smells. For code smells that had already been studied on iOS we validated our implementation by replicating the study performed by Habchi et al. [Hab+17] which we report in Section 4.1. For the differences in results, we checked the source code and determined that the differences were due to the scope of the projects analyzed and not due to the code smell detection methods themselves. For other code smells we made sure to use code smell definitions from reputable sources. All code smell definitions used are cited in Appendix A.2.

External Validity: Since iOS applications on the App Store are encrypted, using closed-source applications for analysis is a lot more difficult for iOS applications than for Android applications. To mitigate this risk we compared the distribution of code smells in open-source applications that are on the app store to the whole set of applications analyzed. The distributions differ slightly but are very similar. This shows that open source applications that were not written to be submitted to the app store are not completely different in terms of code smells than the applications that were meant to be submitted to the app store.

Construct Validity: We used standard definitions of code smells found in the literature. In code smell queries we use thresholds calculated based on the application set analyzed. Using thresholds is a common approach for detecting code smells. We used the same method to determine thresholds as was used by Hecht et al. [Hec+15] and Habchi et al. [Hab+17]. Thresholds might differ between languages, but since they are calculated based on the set of apps analyzed, language-specific differences should be resolved.

Reliability: For iOS we used a collaborative list of open-source iOS applications written in Swift. All these applications are available on GitHub. The list of successfully analyzed iOS applications can be found on the GraphifySwift GitHub page. GraphifySwift code smell analysis tool is open source and also available on the tool GitHub page[5]. For Android analysis we used the list of apps analyzed by [Hab+17], the list of successfully analyzed apps can be found in the list of apps[6]. PAPRIKA is open source and also available on GitHub. The adapted code smell queries used for Android analysis can be found in the list of Android code smell queries[7].

[6]`https://figshare.com/articles/dataset/iOS_and_Android_app_analysis_data/13103012`

[7]`https://figshare.com/articles/conference_contribution/GraphifySwift_queries_adapted_for_PAPARIKA_for_Android_code_smell_analysis/13102994`

6. LIBRARY DEPENDENCY NETWORK ANALYSES

Third-party libraries allow developers to use existing solutions for common tasks and speed up development. For almost every popular programming language there is at least one package manager that can be used to manage these dependencies.

A recent vulnerability in the popular Log4J java logging library affected around four percent of all projects in the Maven repository [WR21]. In 2015, a vulnerability in the popular iOS third-party library AFNetworking was found. The vulnerability affected around 1000 iOS applications with millions of users [Kum15]. Apps can not only be affected when they directly depend on these libraries but also when their direct or indirect dependencies depend on these vulnerable library versions.

The library dependency network (LDN) of a package manager contains all libraries distributed through this package manager and their dependency relationships with other libraries. Dependency networks, including their growth and vulnerability, have been studied thoroughly for many package managers such as, among others, npm, RubyGems, and Cargo [Kik+17; DMG19]. Both Decan et al. [DMG19] and Kikas et al. [Kik+17] highlighted differences between package managers and how policies and quality of the standard library of a language can affect the dependency network structure. So far the dependency networks for the package managers used in iOS development (CocoaPods, Carthage, and Swift Package Manager) have not been studied.

6.1. Method

In this section we describe the research questions and the corresponding methodology.

6.1.1. Research Questions

The goal of the Swift LDN analysis is twofold. Firstly, we aim to understand the Swift LDN better by analyzing its evolution and the spread of vulnerabilities in the LDN. Secondly, we use the Swift LDN as an example to study general aspects affecting LDNs that have not been studied before. We ask how the introduction of new package managers influences the LDN and provide insights for tool developers who wish to work on successful package managers. We also analyze how different version requirement types influence library dependency updates and provide insights for developers who wish to keep their dependencies updated and secure. We start by building a dataset for the Swift LDN as no such database exist so far. We answer the following research questions:

- RQ 2.1: How has the Swift LDN evolved?
- RQ 2.2: How has the package manager use evolved in the Swift LND?

- RQ 2.3: Do version requirement types influence library dependency updates?
- RQ 2.4: How far do vulnerabilities spread in the Swift LDN?

6.1.2. Building the dataset

For data collection, it was necessary to identify libraries that are available through each package manager, to collect dependency and vulnerability data for each library. The total process of the LDN dataset creation is visualized in Figure 18. The whole process consisted of five steps. Some of the analysis steps were done manually, for others we applied GraphifyEvolution and SwiftDependency-Chekcer. Which steps were performed with which tool is shown using different colors. The orange color corresponds to GraphifyEvolution which we described in detail in Section 4.2. The yellow color corresponds to LibraryDependency-Analysis which is a helper script we wrote that can be found on GitHub[1]. Green corresponds to SwiftDependencyChecker which we described in detail in Section 4.3. Blue corresponds to steps that were performed manually. Purple corresponds to data sources and white corresponds to data that was discarded.

Developers can include third-party libraries in projects in multiple ways. Libraries can be either directly downloaded and imported manually or developers can use package managers. The use of a package manager makes it easier to include a new library and foremost it makes it easier to keep the library up to date. The package managers used in Swift development (i.e. for iOS, macOS, and watchOS applications) are CocoaPods, Carthage, and Swift Package Manager. These package managers are fundamentally different. CocoaPods is a package manager with a central database of libraries. If a developer wants to distribute their library through CocoaPods they need to create a Podspec file and add it to the CocoaPods Spec repository. This repository is public and can be accessed by anyone. Swift Package Manager (Swift PM) is the official package manager for Swift. It is however not the most popular package manager and compatibility with iOS projects was not added until 2019 [Ell20]. Swift PM does not have a central list of libraries. Any library that includes a Package.swift manifest file can be included through Swift PM by providing its repository address. Carthage, similarly to Swift PM, is a decentralized package manager. A library can be included by providing its repository address, binary location, or path on the local file system.

Step 1: Due to the differences in the package managers, we identified libraries for CocoaPods differently than for Carthage and Swift PM. For CocoaPods, we cloned the Spec repository[2] and extracted all repository URLs from the Podspec files. The extracted list included 79557 repository URLs. Among these URLs were incorrect values, such as "./", ".git", "someone@gmail.com". Some URLs were correct URLs, but they were links to private repositories on company do-

[1] `https://github.com/kristiinara/LibraryDependencyAnalysis`

[2] `https://github.com/CocoaPods/Specs`

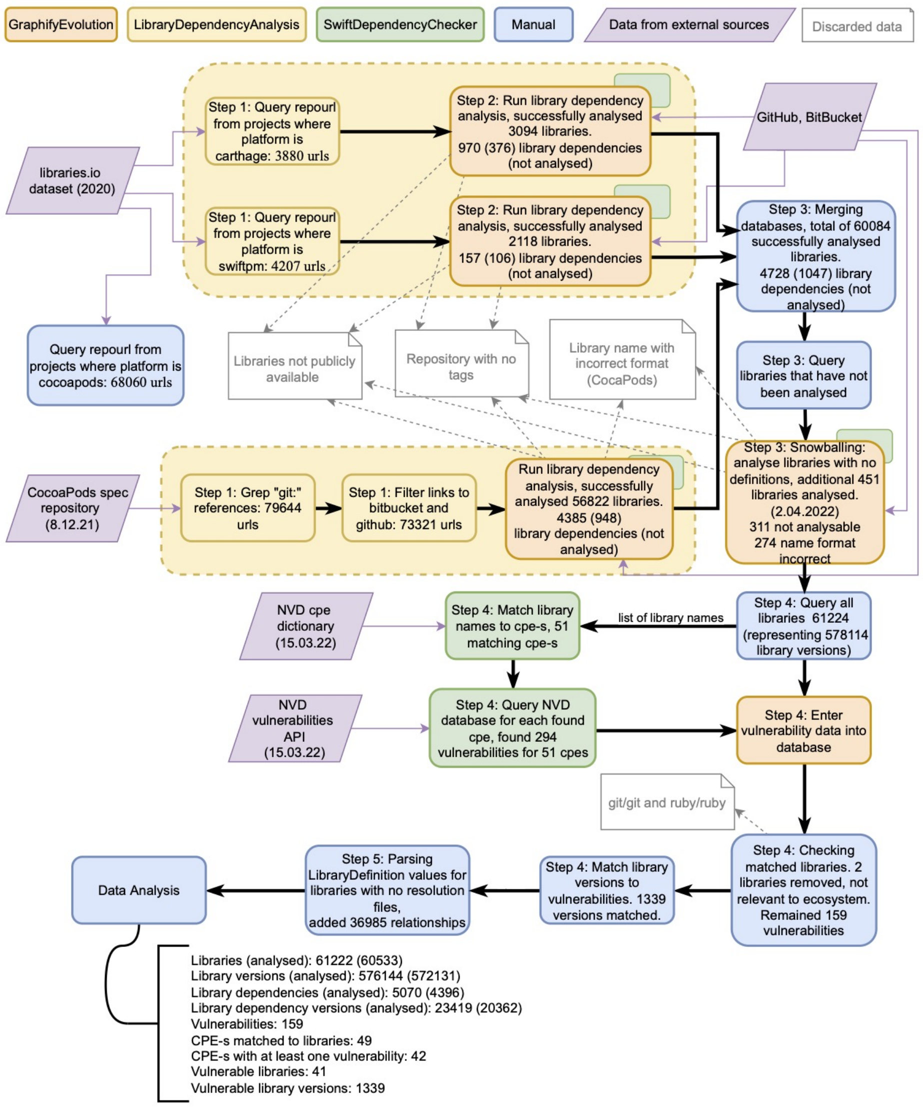

GraphifyEvolution
LibraryDependencyAnalysis
SwiftDependencyChecker
Manual
Data from external sources
Discarded data
libraries.io dataset (2020)
Step 1: Query repourl from projects where platform is carthage: 3880 urls
Step 1: Query repourl from projects where platform is swiftpm: 4207 urls
Step 2: Run library dependency analysis, successfully analysed 3094 libraries. 970 (376) library dependencies (not analysed)
Step 2: Run library dependency analysis, successfully analysed 2118 libraries. 157 (106) library dependencies (not analysed)
GitHub, BitBucket
Step 3: Merging databases, total of 60084 successfully analysed libraries. 4728 (1047) library dependencies (not analysed)
Query repourl from projects where platform is cocoapods: 68060 urls
Libraries not publicly available
Repository with no tags
Library name with incorrect format (CocaPods)
Step 3: Query libraries that have not been analysed
CocoaPods spec repository (8.12.21)
Step 1: Grep "git:" references: 79644 urls
Step 1: Filter links to bitbucket and github: 73321 urls
Run library dependency analysis, successfully analysed 56822 libraries. 4385 (948) library dependencies (not analysed)
Step 3: Snowballing: analyse libraries with no definitions, additional 451 libraries analysed. (2.04.2022) 311 not analysable 274 name format incorrect
NVD cpe dictionary (15.03.22)
Step 4: Match library names to cpe-s, 51 matching cpe-s
list of library names
Step 4: Query all libraries 61224 (representing 578114 library versions)
NVD vulnerabilities API (15.03.22)
Step 4: Query NVD database for each found cpe, found 294 vulnerabilities for 51 cpes
Step 4: Enter vulnerability data into database
git/git and ruby/ruby
Step 4: Checking matched libraries. 2 libraries removed, not relevant to ecosystem. Remained 159 vulnerabilities
Step 4: Match library versions to vulnerabilities. 1339 versions matched.
Step 5: Parsing LibraryDefinition values for libraries with no resolution files, added 36985 relationships
Data Analysis
Libraries (analysed): 61222 (60533)
Library versions (analysed): 576144 (572131)
Library dependencies (analysed): 5070 (4396)
Library dependency versions (analysed): 23419 (20362)
Vulnerabilities: 159
CPE-s matched to libraries: 49
CPE-s with at least one vulnerability: 42
Vulnerable libraries: 41
Vulnerable library versions: 1339

Figure 18. Dataset creation diagram

mains. We discarded all URLs that did not contain github.com or bitbucket.org to combat this issue and 73243 URLs (92%) remained. We looked through these values and noticed that some of the URLs contained references to passwords and usernames of the form username:password@github.com. We decided to strip these values before the analysis. This leads to us not being able to access some of the repositories. Nevertheless, we assumed that it might have been the developers' intention to not make the library code accessible to everyone, although the password and username combination would be accessible to anyone through the public Spec repository. For Carthage and Swift PM we used the libraries.io dataset [Lib22] to get a list of library names. The set of library names is not complete since the dataset was compiled in 2020 and new libraries may have been created after that. We extracted 3880 names for Carthage libraries and 4207 names for Swift PM libraries.

Step 2: We ran our analysis on these three sets of libraries. The analysis was successful for 56822 CocoaPods libraries, 2118 Swift PM libraries, and 3094 Carthage libraries. We then merged the three databases. The merged database contained 60084 successfully analyzed libraries. We then queried libraries that are referenced as dependencies but that are not analyzed. There were a total of 4728 library dependencies of which 1047 were not analysed. We gathered the names of these libraries and performed a second round of analysis. We refer to this as library snowballing. There may still exist libraries, that are available through one of the three package managers, that are missed by our approach. These libraries however will not have any dependents and would not be essential to dependency network analysis.

Dependency data was collected by parsing the manifest files Package.swift, Podfile and Cartfile, and package manager resolution files Package.resolved, Podfile.lock and Cartfile.resolved. We extracted Library names, versions, and version constraints from the manifest files and stored the data as library definitions. We parsed package manager resolution files and extracted library dependencies with names and versions. Package resolution files contain the exact version of each library that the package manager deemed to be the best match at the time when the developer last updated the dependencies. Package resolution files also contain information on all transitive dependencies. Information on both direct and transitive dependencies was stored in the database. To match library names extracted from the three package managers it is necessary to "translate" library names. The translation was done by finding information on the library repository URL from the CocoaPods Spec repository. We used the repository username/projectname combination as the library name.

Step 3: We queried libraries that are referenced as dependencies from other libraries but were not yet analyzed. These libraries included open-source libraries that had not been analyzed yet and closed-sourced or local libraries that were not accessible to us. For each library name that is a dependency through CocoaPods, we were able to extract the repository URL from the Spec repository. These libraries should have been already analyzed. If some of these URLs were discarded previously but the repository is accessible to us the project was included in the analysis. For each library name that was not a dependency through CocoaPods, the name was of the form username/projectname. For each name, we tried to query the repository. If we were able to access the repository the library was analyzed. If we were not able to access the repository the library was ignored. The snowballing process added 451 additional libraries to the database.

Step 4: For vulnerability data we used the NVD database[3]. For each project that has publicly reported vulnerabilities, there is a unique CPE value. We downloaded the CPE dictionary[4]. We then went through the dictionary and extracted all repository URLs and their corresponding CPEs. We are analyzing open-source libraries, therefore we decided to extract repository URLs and ignore all entries that did not include a source reference. We found 5885 CPE values. Next, for each library name, we checked if it matched the entries in the CPE list. We found 51 matching values. For each CPE value, we queried the NVD database to find vulnerabilities related to each CPE. We checked each library that was matched to a CPE value to determine if the library was indeed part of the CocoaPods, Carthage and Swift PM ecosystem. We removed two libraries that were included through libraries.io but were not relevant to the ecosystem. We then matched library versions from the vulnerability data with library versions in our database. We found 159 vulnerabilities in total that affected 41 libraries and 1339 library versions.

Step 5: We prepared for data analysis by adding additional library dependencies based on LibraryDefinition relationships and by creating direct dependency relationships between library nodes. These direct relationships between library versions make it possible to more easily query transitive dependency chains. Matching LibraryDefinition objects to Library objects was done using cypher queries in a semi-manual way. Versions were transformed into tuples of major, minor, and patch, and depending on the constraint type a different cypher query is constructed to find matching Library objects. Used cypher queries are available on GitHub[5]. First, we queried LibraryDefinition dependencies and matched them to Library objects. The match is made by first parsing the version constraint and then matching all Library objects with the same name and matching version. For example a LibraryDefinition with a version constraint "$\rightsquigarrow$ 1.2.3" was matched to Library objects with the same name and versions "1.2.3", "1.2.4" and "1.2.5" but not to "1.3.0". For LibraryDefinition objects where both name and version matched a MATCHES relationship was added. For LibraryDefinition objects where the name matched but no version matches were found NAME_MATCHES relationships were added to all versions. We then queried libraries that include dependencies through manifest files, but where the resolution file was missing from

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

[4]`https://nvd.nist.gov/products/cpe`

[5]`https://github.com/kristiinara/LibraryDependencyAnalysis/blob/main/scripts/parsing_library_definitions.txt`

the repository and therefore, no dependency relationships to Library objects were added. We queried matched Library nodes for each LibraryDefinition and filtered out the latest possible version that matched the tag commit version for the given repository. If a match was found we added DEPENDS_ON relationships to the Library nodes. Finding the most recent dependency version that would have been available at the time the library version was released emulates how the package manager would have resolved the version constraint.

Lastly, we added direct dependency relationships between libraries. For this, we queried all dependency relationships:

```
MATCH
    (l:Library)<-[:IS]-(a:App)
        -[:DEPENDS_ON]->(d:Library)
```

and created direct relationships between library versions:

```
CREATE
    (l)-[:LIBRARY_DEPENDS_ON]->(d)
```

6.1.3. RQ 2.1: Evolution of LDNs

Although several studies analyzed LDNs, especially for npm and Maven, no studies exist for the LDNs of CocoaPods, Carthage and Swift PM. To better understand the Swift ecosystem we first analyse how the combined Swift LDN and the LDNs of the three package managers have evolved over time. Analysing the evolution of the LDNs lays the groundwork in understanding the ecosystem. Many problems in a LDN are amplified through a rapid growth. For example the potential risk from vulnerable libraries grows if the number of dependencies grows.

Using the created LDN dataset we plot the cumulative number of all libraries and library versions including libraries that have no dependencies and no dependents. This might include unused libraries.

In further analysis, we only consider connected libraries, i.e., libraries with at least one dependent or dependency. We first look at how the number of libraries has grown for each package manager. For this, we find the first version of each library, group by the month of its commit timestamp, and count the number of unique libraries cumulatively. We plot the cumulative curve for each package manager. We then calculate how the number of library versions has grown for each package manager. Again, we group the library versions by the months of commit timestamps. We count the number of library versions released each month and take the cumulative sum. The cumulative curve is plotted for each package manager.

Lastly, we plot the mean number of direct and transitive dependencies for each month, as a monthly snapshot. The monthly snapshot is calculated by finding library versions released during each month and for each library taking the last library version for each month. The number of direct and transitive dependencies is found by querying LIBRARY_DEPENDS_ON chains with a length of up to 10.

A maximum threshold for the dependency chain needs to be set for performance reasons, we did, however, confirm that very few dependencies existed beyond that level. The ratio of the total number of dependencies to the number of libraries is then calculated and plotted.

6.1.4. RQ 2.2: Evolution of Package Managers

The objective of this research question is to understand what properties make a newly proposed package manager attractive to developers. Existing research, so far, has analyzed each package manager ecosystem separately. The LDNs of many package managers have been studied and compared. For example, Kikas et al. [Kik+17] created a dependency dataset and studied the LDNs of JavaScript, Ruby, and Rust. Decan et al. [DMG19] used the libraries.io dataset to study the growth of LDNs of seven package managers. Decan et al [Dec+16] analysed the intersection of GitHub and CRAN in the R ecosystem. They showed that GitHub is not only used to develop R libraries, but also to distribute them as the growth of CRAN has diminished some of the benefits of distributing libraries thorugh the package manager. No study so far, however, has analyzed the evolution of multiple package managers in the same ecosystem. Many ecosystems exist where multiple package managers can be used. Sometimes a non-official package manager is released that serves a niche subset of the ecosystem (for example Bioconductor[6] for Python that is used for bioinformatics-related libraries). Sometimes a new package manager is released, but the underlying library repository stays the same (for example npm and Yarn for JavaScript [Yar23]). In other cases, each package manager has its own separate library dependency network. The Swift ecosystem is a fairly unique ecosystem that contains multiple popular package managers, that have been introduced over the lifetime of the ecosystem allowing us to analyze how the introduction of new package managers affects the library dependency network. First, we need to understand how existing package managers have been used and how the introduction of new package managers has influenced the ecosystem. We then discuss properties of package managers that may increase their adoption by developers.

First, we are interested in how the overall popularity of package managers has evolved over time. Our assumption is that over time a larger percentage of developers adapt package managers for their projects. For the years 2012 to 2021, we count the number of unique libraries that have declared a dependency through CocoaPods, Carthage, Swift PM or that were using no package manager. For each year we find the newest version of each library and group unique libraries by their use of package managers. If a library uses multiple package managers then it is counted under each of the used package managers. We calculate and report the percentage of libraries using CocoaPods, Carthage, Swift PM, and no package manager for each year.

[6] `https://www.bioconductor.org`

After analyzing the popularity of package managers we question if package managers are used concurrently. Our assumption is that most popular libraries are available through more than one package manager and are therefore also using multiple package managers. For an average library, however, using multiple package managers should not be necessary. Libraries can belong to a package manager in two different ways. They can either have dependencies or dependents through a package manager. If a library has dependencies through a package manager then the developers of that library actively use this package manager. If a library has dependents through a package manager then developers of other libraries include this library as a dependency through the given package manager. For all libraries that have dependencies, we count the number of libraries that include dependencies through each package manager and each combination of package managers. For all libraries that have dependents, we count the number of libraries that are included as dependencies through each package manager and each combination of package managers. The percentages of all these combinations are calculated for four different years: 2016, 2018, 2020, and 2021. Each yearly snapshot is derived by only including libraries, that have had updates in the given year and taking into account the last version of that library within the year.

The first package manager in the Swift ecosystem, CocoaPods, was released in 2011. During the last 10 years, two more package managers were introduced to the Swift ecosystem. We ask how the introduction of new package managers influences the evolution of the package manager ecosystem. It is probable that new package managers were released because of a lack of desired features in the existing package managers. We see two possible evolution patterns:

1. Each new package manager brings new users that adopt using third-party libraries in their applications.
2. New package managers introduce new features and take over users from older package managers.

To better understand the evolution of the package managers' ecosystems, we first ask if existing libraries are switching to the newest package manager and then ask if new libraries prefer the newest package manager. Given that a migration between package managers might be costly we assume that there is little migration between package managers, but that more and more new libraries would prefer the newest package manager. We analyze how libraries migrate between package managers by recording yearly changes of package manager use and drawing a Sankey diagram. For this, we group libraries by year and package manager. For each year, library and package manager we then record if the library used the same package manager in the previous year. If not, we record if the library used a different package manager in the previous year. If yes, we count this as a migration between package managers. We then plot and interpret a Sankey diagram displaying the migrations between package managers and between using a package manager and not using a package manager. We also plot and interpret the percentage of libraries migrating from each of the package managers to using no package manager or other package managers. Additionally, we report how many new libraries use each package manager each year

6.1.5. RQ 2.3: Upgrades vs Version Requirement Types

Kikas et al. [Kik+17] studied the evolution of three library dependency networks. When looking at the dependency version requirement types, they found that the most popular requirement types were different for the three ecosystems. JavaScript libraries preferred the ^ notation (up to the next major version), Ruby libraries preferred the any notation (allowing any version of a library) while Rust libraries mostly required an exact version of a library. Our expectation is that the choice of the version requirement type affects how library dependencies are updated. With the insight of how version requirement types affect dependency updates we can make recommendations to developers to help reduce technical lag.

First, we measure the technical lag as dependency updating lag for each package manager. Since developers seem to be reluctant to update their library dependencies our expectation is that this is also the case in the Swift ecosystem. We do, however, not know to what extent. Given that Swift is a relatively new language and not backward compatible [Ils22], we expect the lag time to be lower than, e.g., for Java projects. Since all three analyzed package managers belong to the same ecosystem, we do not expect big differences between them. For each analyzed project version, i.e., App node, we find library versions it depends on. We then find the latest version of each dependent library that was released before the analyzed project version. If the latest library version is already used as a dependency then the dependency updating lag time is zero. If a newer library version exists, then the dependency updating lag time is calculated by subtracting the commit timestamp of the newest library version from the analyzed project version. We record how many dependencies have a dependency updating lag and calculate the mean dependency updating lag time for dependencies that are not up to date. Next, we calculate monthly snapshots of the library dependencies and calculate the mean dependency updating lag time for each month. We then plot the mean dependency updating lag time in days for each package manager.

Second, we analyze how a chosen version requirement type affects the dependency updating speed. Previous work has shown that developers are reluctant to update library dependencies [DMC18; Sal+18]. Thus, our expectation is that ~> is the most often used version requirement type for all the three analysed package managers. This requirement type allows updating until (but not including) the next minor or major version. This provides benefits from updating, such as including security fixes, while avoiding problems from major changes to the library functionality. Furthermore, we expect that version requirements that do not provide an upper bound result in a smaller updating lag time. If our assumption holds, it provides a good incentive for developers to use such version requirements. We analyze the frequency of dependency version requirements used with each of the package managers by finding all

```
    (App)-[DEPENDS_ON]->(LibraryVersion)
```

chains. The LibraryVersion node contains information on the version requirement type and the DEPENDS_ON relationship contains information on the package manager used. Then, we group the version requirements by package manager and version requirement type and plot the frequencies for each package manager. To investigate how the used version requirement affects the dependency updating lag, we first need to match the correct LibraryDefinition and Library nodes. The App-to-LibraryDefinition node relationship corresponds to how the developer-defined the library dependency in the package manager manifest file. The App-to-Library node relationship corresponds to which actual library version was resolved by the package manager a the given time. We match LibraryDefinition and Library nodes by first finding the Library and LibraryDefinition nodes connected to the same App node. We then pair the Library and LibraryDefinition nodes where the name of the Library node ends with the name of the LibraryDefinition node. We use the ends with a match instead of an exact match to account for cases where the LibraryDefinition only contains a shorter version of the library name. To analyze the relationship between version requirement types and the dependency updating lag, we first find the percentage of dependencies with lag for each version requirement type and package manager. Then we calculate the mean dependency updating lag time for each version requirement type over all dependencies that are not up to date. We plot the dependency updating lag time for each version requirement type. To take into account possible differences between package managers we plot the dependency updating lag time per version requirement type additionally for each package manager.

Third, we analyze whether rerunning the package manager version resolution affects the updating of dependency versions. We expect that when developers do not rerun the package manager version resolution then library dependency versions are not updated although the version requirement would allow an update, yielding increased dependency updating lag time. To analyze whether or not rerunning the package manager version resolution affects version updating, we check whether the library dependency versions between two consecutive versions of a project changed. We then distinguish between (i) library dependency versions that were directly extracted from package manager resolutions files in the project repositories and (ii) library dependency versions that were calculated based on the version requirement in the package manager manifest file. For the latter case, the calculation is done as if the developers of the library had rerun the package manager version resolution for each new library version. We match App nodes connected to matching LibraryDefinition and Library node pairs. We then find the consecutive version of the App node and check if it is connected to the same

Library node as the previous version. If the Library node changes and it is a previous version of the same library we record this as a downgrade. If the Library node is a later version of the library we record this as an upgrade. If the Library node remains the same we record this as no change. In addition, we check whether the Library node was originally extracted from the package manager resolution file. Once having found all upgrades, downgrades, and no changes in versions, we plot the difference between version changes where the library version was extracted from the package manager resolution file and version changes that were calculated based on the package manager manifest file.

Lastly, we investigate how often a vulnerable library dependency could be fixed by upgrading the library dependency version. Additionally, we check how often the vulnerable library dependency could be fixed by simply rerunning the package manager version resolution. Previous research shows that the most vulnerable dependencies could be fixed by upgrading the library dependency version [Der+17]. Our expectation is that this should also be the case in the Swift ecosystem. We also expect that in some cases these vulnerable dependencies could already be fixed by rerunning the package manager version resolution without changing the version requirement itself. The Swift LDN dataset includes, in addition to data on library dependencies, data on vulnerable library versions. For each project version with a dependency to a vulnerable library version, we check if the vulnerable library dependency could be fixed by upgrading the library dependency version. Additionally, we check if simply rerunning the package manager version resolution would have resulted in fixing the vulnerable dependency. For this, we find App-Library-Vulnerability chains which indicate that the project version depends on a vulnerable library version. For each of these App and Library pairs we find the corresponding LibraryDefintion. We then check if there is a Library node that is a future version of the previously found Library node, but which is not connected to a vulnerability and where the library version was released before the App commit time. In other words, we find the next library version that does not have a publicly reported vulnerability. We then check if the LibraryDefintion is connected to the new Library node. If yes, then it means that the vulnerable library dependency could be fixed by re-running the package manager version resolution. If not, then the vulnerable library dependency could be fixed by upgrading the library dependency version. If no such Library node was found then it means that the vulnerable library dependency could not be fixed at the time of the project version release. We then plot the number of projects with vulnerable library dependencies that could have been fixed by a version update, that could have been fixed by rerunning the version resolution and that could not have been fixed through a version update.

6.1.6. RQ 2.4: Spread of Vulnerabilities

Third-party solutions are often better vetted than custom solutions. The Open Web Application Security Project (OWASP), for example, strongly recommends against the use of custom encryption algorithms[OWA16]. Nevertheless, vulnerabilities can be found in even very popular and well-tested libraries. For example, in December 2021, a security vulnerability was discovered in the widely used Log4J Java logging library. This vulnerability affected 4% of all the Java applications [WR21] and made them vulnerable to remote code execution attacks. The spread of vulnerabilities in package manager library dependency networks has been studied for some package managers. Zerouali et al. [Zer+22] studied how long it takes for vulnerabilities in npm and RubyGems to be fixed and how these vulnerabilities spread through the library dependency network. They found that around 40% of libraries have a direct or transitive dependency on a vulnerable library version. Düsing et al. [DH21] analyzed how vulnerabilities in transitive dependencies affect the NuGet, npm, and Maven package manager library dependency networks. They also studied how fast developers update their library dependencies when a vulnerability is publicly disclosed. They found that there is a significant difference in how many libraries are affected by vulnerable dependencies depending on the package manager. They also found that developers probably rely on automated dependency updates, which are triggered when a vulnerability is disclosed.

Although there are many studies that analyze library dependency networks, especially for npm and Maven, there are no studies analyzing the Swift LDN. Our goal is a) to understand the scope of the library dependency network affected by vulnerabilities, b) if vulnerable dependencies could be effectively fixed via upgrading, and c) if there is enough public information available about these vulnerabilities such that the functionality of existing tools could be complemented with more detailed yet lightweight vulnerability analyses. This understanding can help tool developers in developing tool support for Swift developers that can help reduce impact from vulnerable dependencies. The analysis on publicly available information helps estimate the effort needed for more detailed vulnerability analyses.

To better understand the risks imposed by vulnerabilities in the library dependency network, we ask how libraries in Carthage, CocoaPods and Swift PM are affected by vulnerabilities. To get started, it is necessary to investigate which libraries have publicly reported vulnerabilities. We are aware that the actual number of vulnerabilities will be higher, as not every vulnerability is publicly reported or even detected. Nevertheless, it is reasonable to look at publicly reported vulnerabilities instead of running a vulnerability scanner, to avoid the multitude of false positive results that these tools usually produce. Looking at publicly reported vulnerabilities we can be reasonably certain that these vulnerabilities are true positives and no manual double-checking is required. We expect to find publicly reported vulnerabilities in a certain amount of third-party libraries of the Swift ecosystem. Yet, this is not sufficient. Since we expect vulnerabilities to spread through dependency chains, we analyze the library dependency network, i.e., the occurrences and lengths of dependency chains along which vulnerabilities might propagate. In addition, we refine our analysis by including information about the predominantly used project language of the vulnerable library and the severity level of the vulnerability. Libraries in the Swift ecosystem can be written in different languages. The most common languages are Swift, Objective-C, C and C++ [DGC], with Swift and Objective-C covering the vast majority of the libraries. We expect the vulnerable libraries to have a similar distribution of languages as the rest of the ecosystem.

To understand how vulnerable library versions may impact other libraries, we first find all library versions that are connected to vulnerable library versions through DEPENDS_ON chains. A dependency chain of length zero implies that the library version itself is vulnerable. A dependency chain of length one implies that the library version has a direct dependency on a vulnerable library version. Dependency chains longer than one imply that the library version has a transitive dependency on a vulnerable library version. For each library version that depends on a vulnerable library, we find the shortest path to a vulnerable library version. We do this because we assume that the risk of using vulnerable code is higher when the dependency chain is the shortest. We then report the number of libraries for each dependency chain length by filtering out duplicate library names. The resulting numbers indicate how many libraries have publicly reported vulnerabilities and how many libraries depend on vulnerable libraries (either through direct or transitive dependencies).

Additionally, we analyze how the language of the vulnerable library and the severity level of the vulnerability is associated with how far the vulnerabilities spread in the library dependency network. We gather library dependency chains for libraries that depend on vulnerable library versions and plot the number of affected libraries for each dependency level. For libraries with multiple dependencies to vulnerable libraries, we count the library on each dependency level where it depends on a vulnerable library version. We first plot the dependency level graph distinguished by the programming language and then by the severity level of the vulnerability. The language of the library is determined by querying the main project language from GitHub.

The simplest way to fix a dependency on a vulnerable library version is to upgrade to a library version where the vulnerability is fixed if such a fix exists. Given that many developers are wary of upgrading their library dependencies [Zim+19; Li+21; Zer+22] our hypothesis is that, as in other programming language ecosystems, many dependencies to vulnerable libraries remain unchanged although an easy fix is possible via upgrading the library dependency version. To check our hypothesis, we analyze how often vulnerable dependencies could have been fixed by upgrading the library dependency version.

Figure 19 shows eight dependency chains:

- ABC1: (A v1)←(B v1)←(C v1)
- ABC2: (A v1)←(B v1)←(C v2)
- ABC3: (A v2)←(B v2)←(C v3)
- AB1: (A v1)←(B v1)
- AB2: (A v2)←(B v2)
- BC1: (B v1)←(C v1)
- BC2: (B v1)←(C v2)
- BC2: (B v2)←(C v3)

Three of the dependency chains corresponded to vulnerable dependencies: ABC1, AB1, and ABC2. For dependency chain ABC2, Library C could have resolved the vulnerable dependency by upgrading the dependency to Library B from version 1 to version 2. We study how often such chains to vulnerable dependencies could have been fixed via upgrading the dependency version. For this analysis, we first filter out library dependencies where the package manager resolution file was missing. These dependency versions were calculated based on the manifest file and are therefore not suitable for upgradeability analysis.

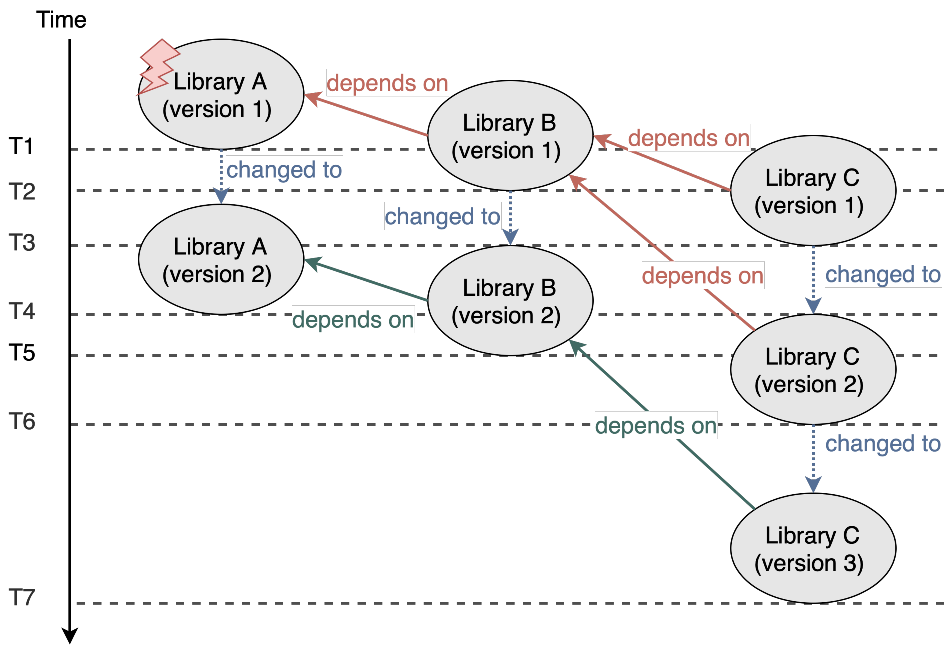


Figure 19. Illustration of dependency chains in a library dependency network with three libraries A, B, and C.

To analyze how vulnerable dependencies could be fixed via upgrading, we first identify all dependency chains to vulnerable library versions. For each of these chains, we then check if a newer version of the direct dependency (like

B:v2 for dependency chain ABC2 in Figure 19) exists that is not dependent on the vulnerable library version. The process of finding the newer version of the direct dependency takes into account release times for each of the library versions such that the release time of the dependency has to always be before the release time of the dependent. This means that in Figure 19 it would have been possible to upgrade the dependency chain ABC2, but not the dependency chain ABC1 because B:v2 was released after C:v1. For each dependency level, we plot the number of dependency chains that could have been fixed via an upgrade and the number that could not have been fixed via an upgrade. Additionally, we count how many library dependencies could have been fixed for each vulnerability severity level and vulnerable library programming language.

The above analysis shows the upgradeability of vulnerable dependencies over the whole time frame of the dataset. To understand the potential impact of upgrades to the most recent state of the library dependency network, we also analyze how many of the latest versions of libraries' vulnerable dependencies could have been fixed via upgrading.

Tools exist that can find dependencies to vulnerable libraries when using CocoaPods, Carthage, or Swift Package Manager. There are, however, no tools for Swift and Objective-C that could perform more detailed analyses and determine if a vulnerability from a library dependency really affects the program. For such analyses, it would either be necessary to have data on where the vulnerability is located in the library or an extensive analysis of the vulnerable library would be needed. Our goal is to check if information about the location of a vulnerability in the code is publicly available for the reported vulnerabilities in the Swift ecosystem. For each vulnerability, we check the public vulnerability description on NVD and record if it contains information about the class or method that contains the vulnerability. Additionally, we check, if available, the patch link to see if the patch of the vulnerability reveals where the vulnerability was fixed in the code.

6.2. Results

In this section wer present results for the four research questions.

6.2.1. RQ 2.1: Evolution of LDNs

We analyzed 60,533 libraries in total. Figure 20 shows the cumulative number of libraries (red) and library versions (blue) over time. The subset of connected libraries and their versions are shown as dotted lines. The total number of libraries grew very fast after the release of the Swift programming language in 2014. From 2019 onward the number of new libraries added has slightly slowed down. A similar pattern can be observed for library versions. Moreover, we see similar trends for the subset of connected libraries, i.e., libraries that either use a package manager or are used through a package manager.

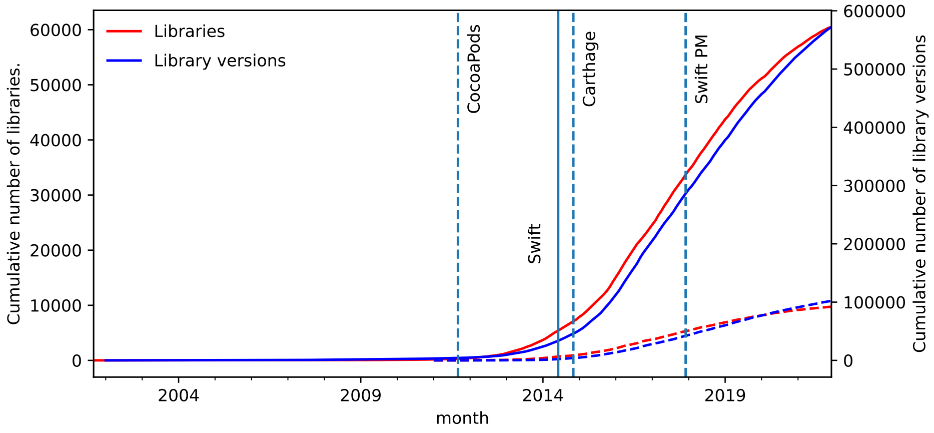


Figure 20. Cumulative number of libraries and library versions. Solid lines show numbers for all libraries, and dotted lines for connected libraries.

In the following analysis, we constrain ourselves to libraries that have at least one dependency or dependent. This means that these libraries are indeed part of at least one package manager LDN. These libraries are called connected libraries. We analyzed the cumulative number of connected libraries for each package manager. In total, there are 9,755 connected libraries. Of these libraries, 6,600 belonged to the CocoaPods LDN, 2,856 belonged to Carthage, and 2,150 belonged to Swift PM. A library can belong to multiple package manager LDNs.

The change in the number of libraries can be seen in Figure 21. The number of libraries is growing fastest for the newest and smallest package manager Swift PM. The number of libraries for CocoaPods is still growing, but the growth has slowed after 2019. The growth of the number of libraries for Carthage has almost completely stagnated. Figure 22 shows the same data as proportion form the combined number of libraries. The proportion of CocoaPods and Carthage has been declining since 2017, while the proportion of Swift PM libraries has been increasing.

Figure 23 shows the mean number of direct dependencies for each monthly snapshot. For CocoaPods, the mean number of direct dependencies fluctuated strongly until 2016. After 2016 the mean number leveled to values around three, which is slightly higher than the mean number of direct dependencies for Carthage and Swift PM, each averaging around 2.5. The mean number of direct dependencies has a slight upwards trend for all three package managers. At the same time the median number of direct dependencies has fluctuated between one and two for Carthage and Swift PM. For CocoaPods the median number has mostly fluctuated between one, two and tree, expect for in 2014 and 2015 where the median value jumped to four for a few months.

In addition, we calculated the mean number of direct and transitive dependencies for all connected libraries. The data shown in Figure 24 is not differentiating

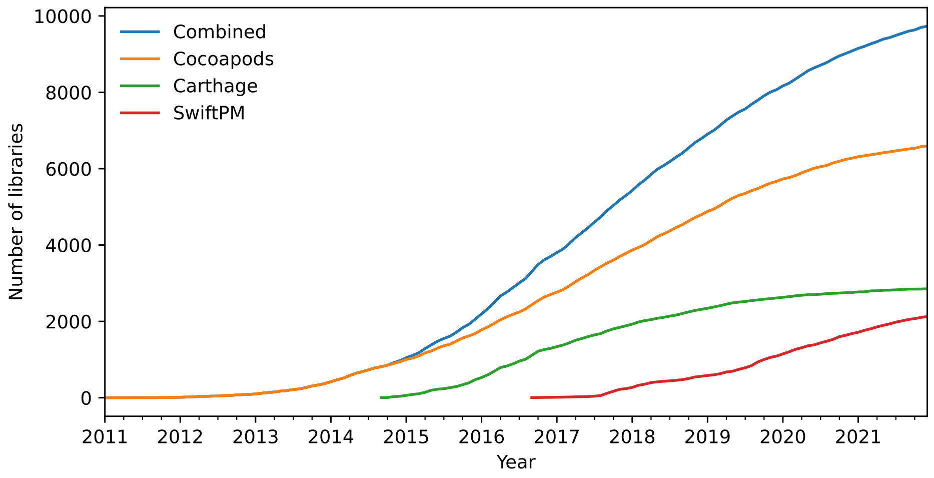


Figure 21. Cumulative number of libraries.

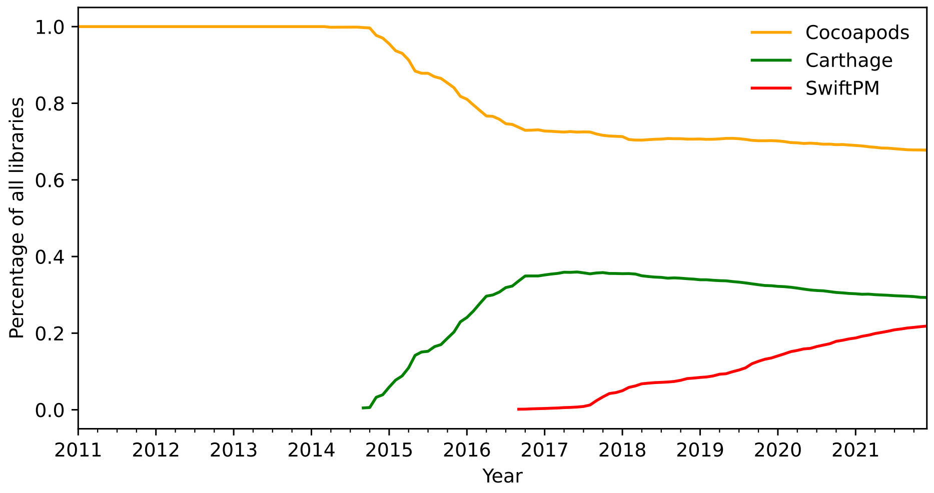


Figure 22. Percentage of cumulative number of libraries.

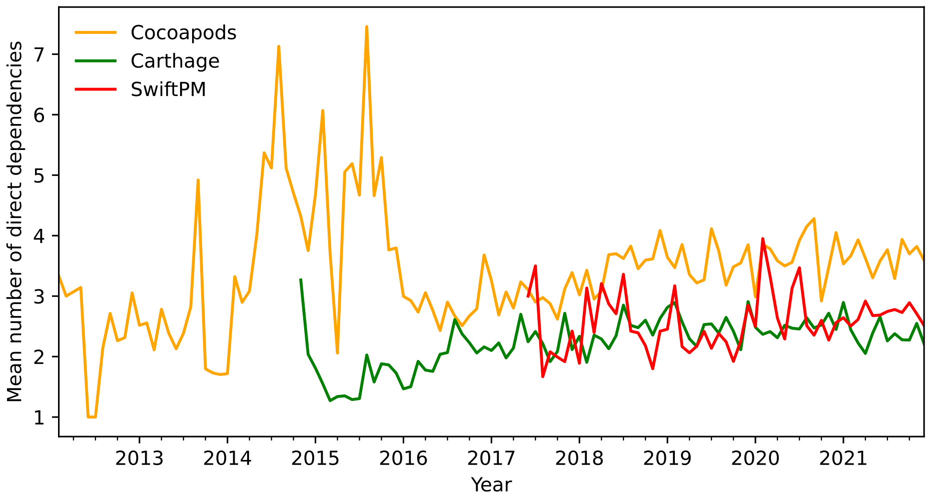


Figure 23. Mean number of direct dependencies for each monthly snapshot.

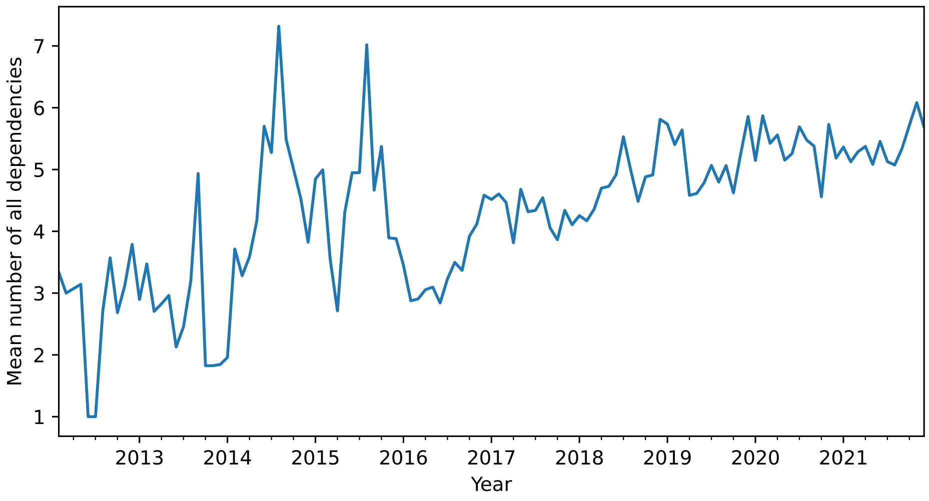


Figure 24. Mean number of direct and transitive dependencies for each monthly snapshot.

between package managers, as calculating transitive dependency chains conditional to package managers were difficult. We did, however, count the number of unique library names as total dependencies in order to not accidentally include the same library twice if it was referenced through multiple package managers. The mean number of dependencies in Figure 24 shows a clear upwards trend. Similarly to the mean number of direct dependencies, the mean number of all dependencies fluctuates considerably until 2016. Between 2016 and 2022, however, there is a clear upwards trend where the mean number of direct and transitive dependencies rises from around 3 to 5.5.

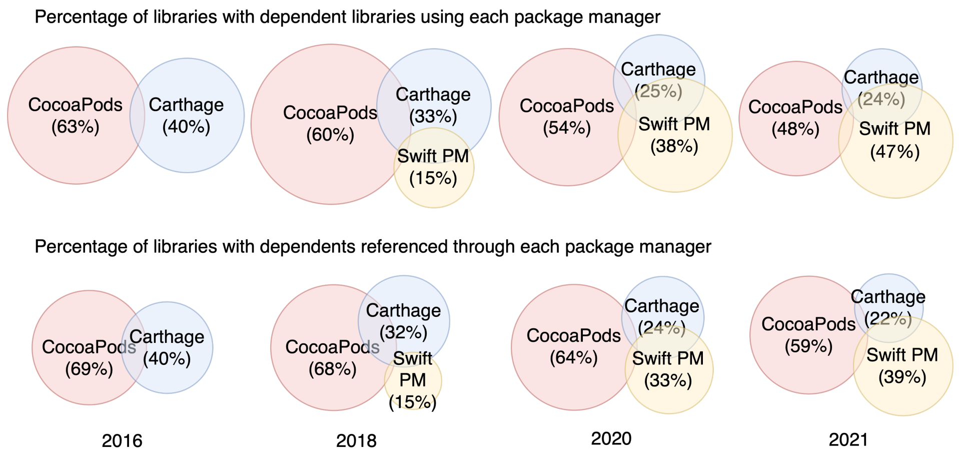


Figure 25. Overlap of package managers (four snapshots)

6.2.2. RQ 2.2: Evolution of Package Managers

We analyzed how many libraries are using a package manager vs. not using a package manager. Table 6 shows that the percentage of libraries using no package manager has steadily decreased from 97.8% in 2012 to 84.3% in 2021. Surprisingly, the overall number of actively maintained libraries in the Swift ecosystem grew up to 2018 and has been falling since.

Table 6. Percentage of libraries using each package manager for years 2012 to 2022.

Year	None	Carthage	CocoaPods	Swift PM	Total
2012	97.8	0.0	2.7	0.0	1067
2013	95.6	0.0	6.1	0.0	3085
2014	93.6	0.5	8.5	0.0	5837
2015	92.5	3.4	7.9	0.0	9920
2016	91.6	5.5	7.3	0.0	15068
2017	90.3	5.4	7.7	0.9	16432
2018	88.5	5.1	8.0	2.2	16523
2019	87.4	5.2	8.6	4.3	15668
2020	86.3	4.7	8.4	6.7	12667
2021	84.3	4.6	8.0	9.0	9504

For all analyzed libraries we counted how many libraries used no package managers, one package manager, or multiple package managers. We took into account the last version of each library, in total 60527 library versions. We found that 52869 (87.3%) libraries did not use any package managers. Of the 7540 libraries that had dependencies 4718 (62.6%) libraries used only CocoaPods, 1141 (15.1%) libraries used only Carthage, and 1001 (13.3%) libraries used only Swift PM. In total 6860 (91.0%) of libraries with dependencies only use one package

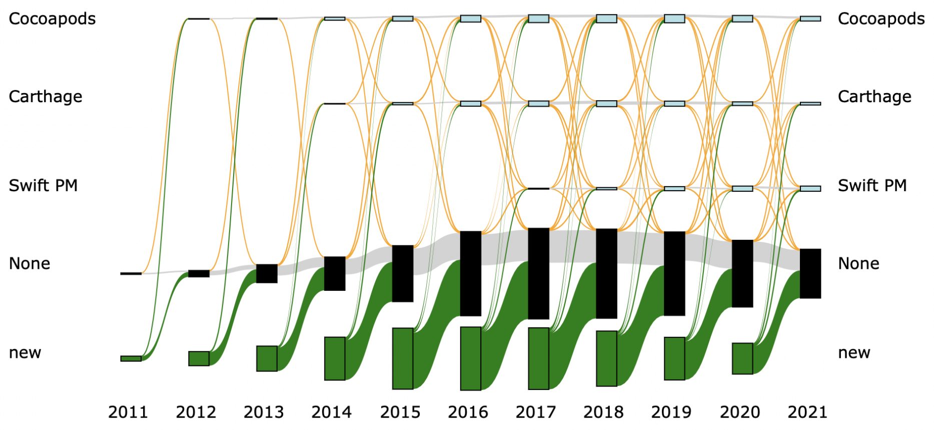


Figure 26. Sankey diagram for all libraries.

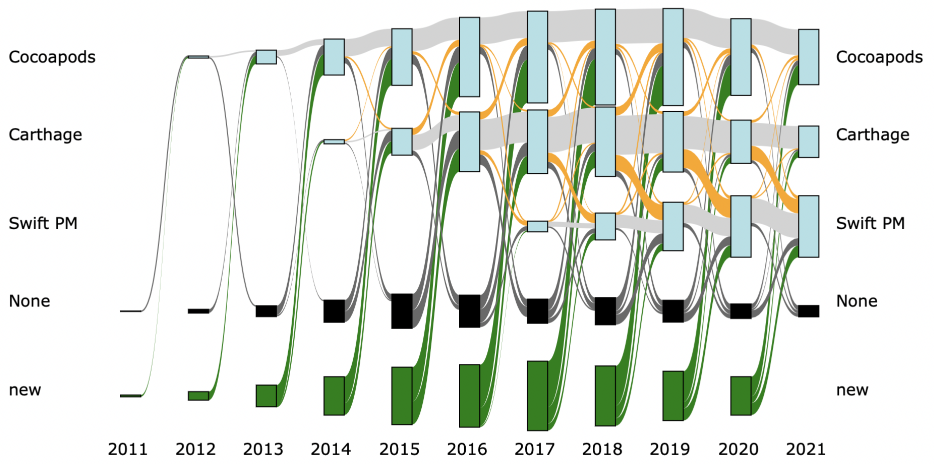


Figure 27. Sankey diagram for libraries that use a package manager.

manager. The remaining 680 (9%) libraries use multiple package managers divided between Carthage and Swift PM with 352 (4.7%) libraries, Carthage and CocoaPods with 162 (2.1%) libraries, CocoaPods and Swift PM with 126 (1.7%) libraries and 41 (0.5%) libraries use all three package managers.

We also calculated these numbers for four snapshots for the years 2016, 2018, 2020, and 2021. Years 2016 and 2018 were considered to capture the change in the LDNs after the introduction of Swift PM in 2017. Years 2020 and 2021 were considered to see the current trends in the LDNs. The snapshots were constructed by only considering the last version of a library for each year. If a library did not have any versions released during a specific year it was not counted. The first row in Figure 25 shows how the concurrent use of the three package managers has evolved. In 2016, 63% of libraries with dependencies used CocoaPods and 40% of libraries used the Carthage package manager. After Swift PM was introduced in 2017 more and more libraries started using it. In the following years 370 (14.5%), 857 (38.2%), and 857 (46.6%) of libraries used Swift PM in 2018, 2020, and 2021 respectively. Multiple package managers were concurrently used by 62 (3%), 258 (11%), 377 (17%), and 317 (17%) libraries in 2016, 2018, 2020, and 2021 respectively.

We analyzed all dependencies between libraries and counted the number of libraries that are included through each package manager. For each library, we took into account the last version of the dependent library. In total, there were 3891 libraries with dependents. We found that 2410 libraries (61.9%) were only used through CocoaPods. Additionally, 562 (14.4%) and 469 (12.0%) libraries were only used through Carthage and Swift PM respectively. The remaining 450 (11.6%) libraries were included through multiple package managers. This number comprises the following usages: 121 (3.1%) libraries through all three package managers, 120 (3.1%) libraries through Carthage and CocoaPods, 103 (2.6%) libraries through Carthage and Swift PM and 79 (2.0%) libraries through CocoaPods and Swift PM. Overall 2730 (70.2%) libraries were used through CocoaPods, 906 (23.3%) libraries were used through Carthage, and 799 (20.5%) libraries were used through Swift PM.

We also analyzed the dependencies between libraries for four snapshots for the years 2016, 2018, 2020, and 2021. The snapshots were calculated by only considering dependents that had a released version in the given year. The last version of the dependent in each year was taken into account. For each library that had dependents in the given year, we counted how many of these dependents were declared through each package manager. The distribution of libraries for these four different snapshots can be seen in the second rows of Figure 25. In 2016, 721 (68.7%) libraries were referenced through CocoaPods, 415 (39.6%) libraries were referenced through Carthage, and 87 (8.3%) libraries were referenced through both package managers. After Swift PM was introduced in 2017 the percentage of libraries referenced through Swift PM grew to 207 (14.5%) in 2018, 489 (33.1%) in 2020 and 508 (38.6%) in 2021. At the same time, the number of libraries included through Carthage shrunk from 415 (39.6%) in 2016, to 288 (21.9%) in 2021.

We analyzed how the introduction of package managers has influenced the evolution of the Swift package manager ecosystem by studying how libraries migrate between package managers and if new libraries prefer the newest package managers. Figures 26 and 27 provide an overview of the package manager ecosystem evolution. Here 'None' signifies libraries that use no package manager and 'New' signifies newly added libraries in the following year. Figure 26 shows the evolution for all libraries. We see, as discussed earlier, that most libraries do not use a package manager. There is however migration between all package managers and from no package manager to using a package manager and the other way around. Figure 27 zooms into the same picture by discarding libraries that use no package manager and that also do not participate in migrations between using a package manager and not using a package manager. We can see that there are rather large migrations from Carthage to Swift PM between the years 2018 and 2021. In the following sections, we analyze these migrations in more detail.

We analyzed the number of libraries that migrate to other package managers for each of the package managers. Figure 28 shows migrations from CocoaPods. We see that most libraries keep using CocoaPods. There are however small migrations to the newer package managers after Carthage and Swift PM are released. The percentage of libraries migrating from CocaPods to Swift PM is slowly growing. Figure 29 shows migrations from Carthage. In contrast to CocoaPods there is a large percentage of libraries migrating away from Carthage from the beginning. Since the release of Swift PM, the largest migration is towards Swift PM. Figure 30 shows the migrations from Swift PM. There are small migrations towards Carthage and CocoaPods. The majority of the libraries use Swift PM, however, keep using Swift PM. We analyzed which package managers are used by new libraries. Figure 31 shows how the percentage of libraries using each package manager has changed over time. After Carthage was released in 2014 the percentage of libraries using CocoaPods has stayed between 50% and 70%. The most popular years among new libraries for Carthage were 2015, 2016, and 2017. After the release of Swift PM in 2017 its popularity among new libraries has steadily increased.

6.2.3. RQ 2.3: Upgrades vs Version Requirement Types

First, we studied to what extent dependency updates were lagging behind. The proportions of dependencies having a lag were similar for all package managers, i.e., 43% of dependencies in CocoaPods, 32% of dependencies in Carthage, and 39% of dependencies in Swift PM. For dependency updates that had a lag, the mean lag time was 92 days for CocoaPods, 45 days for Carthage, and 58 days for Swift PM, as shown in Table 7. The median lag time was 34.4 days, 17,4 days, and 25.6 days for CocoaPods, Carthage, and Swift PM, respectively. We used the

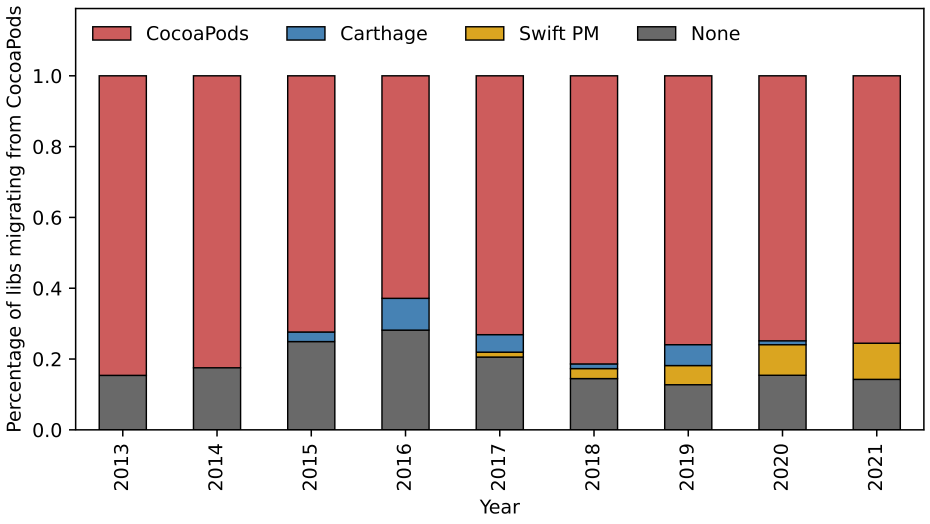


Figure 28. Percentage of libraries migrating from CocoaPods

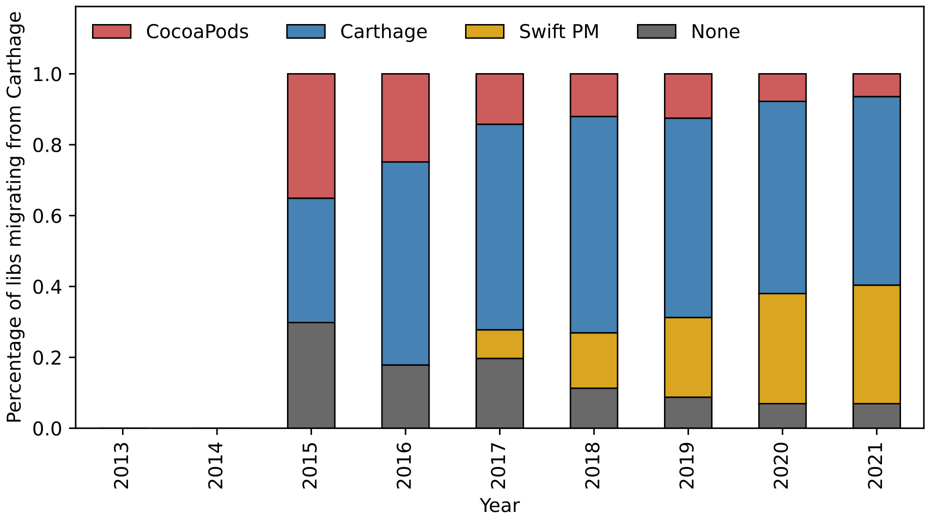


Figure 29. Percentage of libraries migrating from Carthage

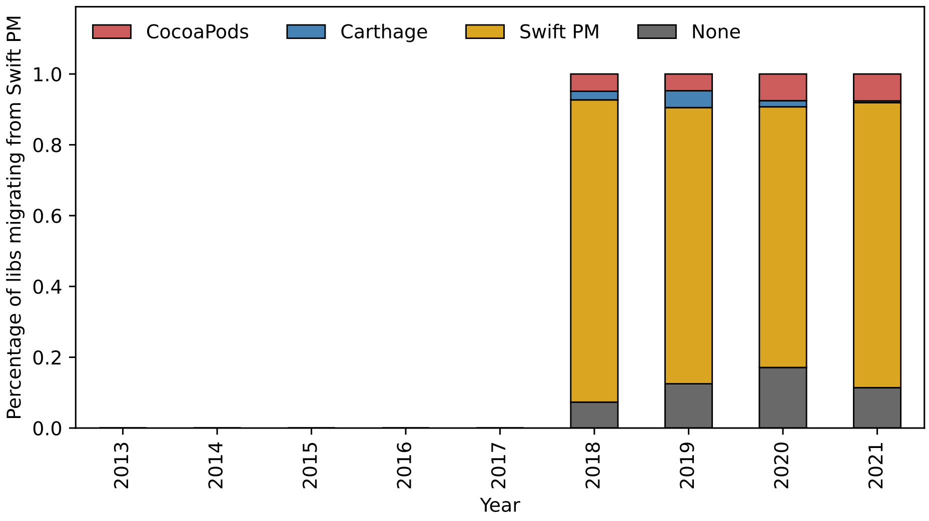


Figure 30. Percentage of libraries migrating from Swift PM

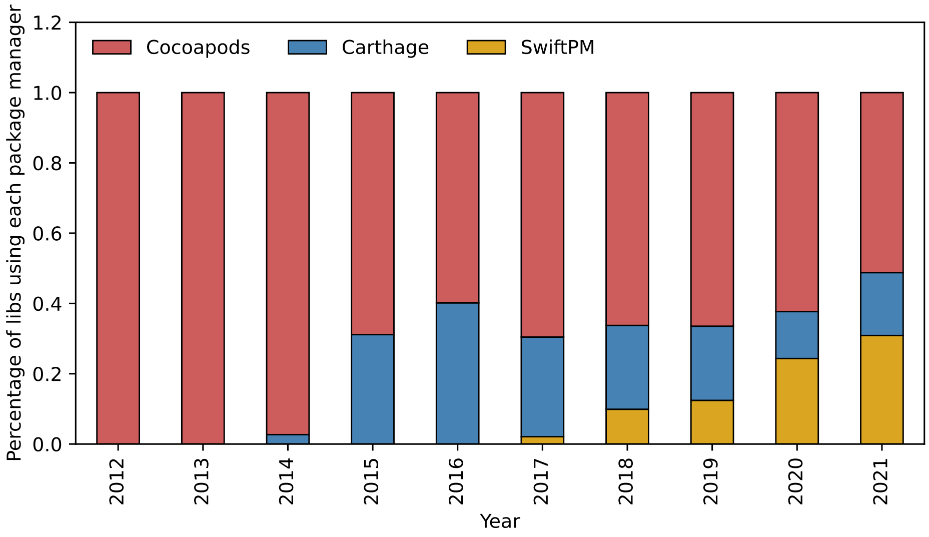


Figure 31. Percentage of new libraries using each of the package managers.

Table 7. Lag time in days for dependencies with updating lag for each PM.

PM	count	mean	std	min	25%	50%	75%	max
CocoaPods	52975	91.6	170.5	<0.1	10.6	34.4	97.2	2625.4
Carthage	19957	44.8	86.1	<0.1	4.7	17.4	50.0	1827.0
Swift PM	7863	58.2	86.8	<0.1	6.2	25.6	75.5	869.8

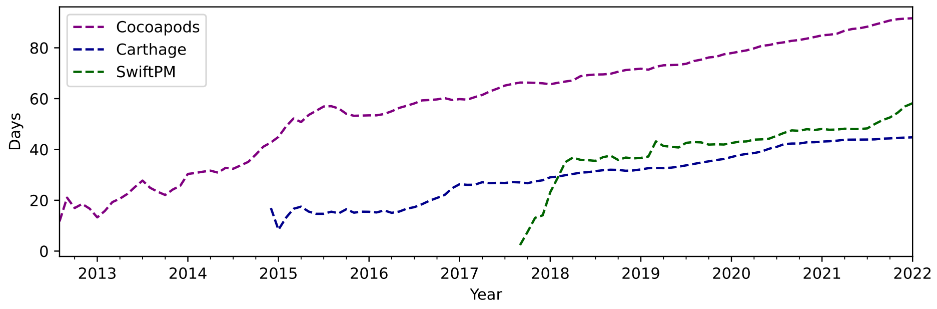


Figure 32. Mean dependency updating lag time per package manager.

Mann-Whitney U test to compare the distributions for dependency updating lag between the three package managers. For each combination of two package managers the p-value from the U test was smaller then 0.0001 and therefore smaller than the Bonferroni corrected significance threshold of 0.008. We conducted this test separately for dependencies with updating lag and for all dependencies.

The mean lag time for monthly snapshots is shown in Figure 32. The data indicates that the lag time has been growing over time for all package managers. In 2022, the mean dependency updating lag time for CocoaPods is over 80 days, for Carthage, it is over 40 days, and for Swift PM around 50 days.

We investigated the relationship between version requirements and the dependency updating lag. Our expectation was that version requirements without an upper bound would have a smaller lag time than version requirements with upper bound or strict version requirements. We analyzed which version requirements are used through CocoaPods, Carthage, and Swift PM. Figure 33 shows the proportion of each version requirement type. The analysis showed that when using CocoaPods, developers use the latest version option almost exclusively. For Carthage, the most often used requirement type is ~>, followed by the exact version and the latest version. For Swift PM the most common requirement type is >=, which is very similar to the latest, followed by ~> and the exact version.

Table 8 presents for each package manager the percentages of dependencies with updating lag of the four most popular version requirement types. For CocoaPods types ~> and >= were excluded as there were no or too few such dependencies. For all three package managers, we see that the more restrictive version requirement types have a larger percentage of dependencies with lag than the less restrictive requirement types. For Carthage and Swift PM this difference is par-

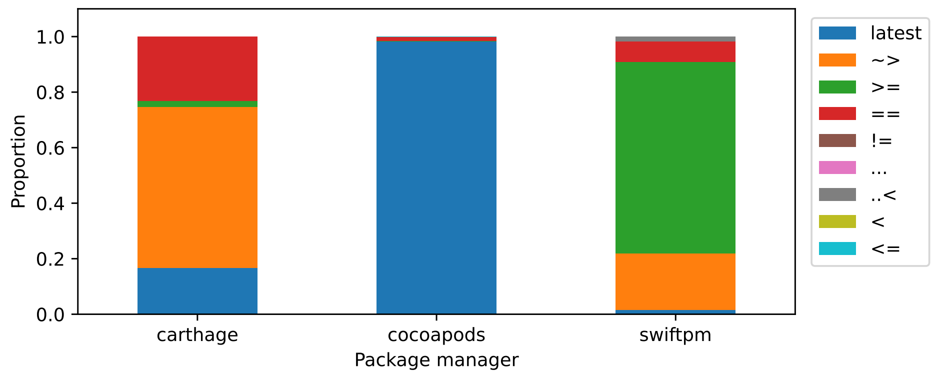


Figure 33. Proportion of requirement type

Table 8. Percentage of dependencies with updating lag per package manager and version requirement type.

PM	==	~>	>=	`latest`
CocoaPods	52%	-	-	44%
Carthage	46%	29%	27%	25%
Swift PM	47%	40%	38%	25%

ticularly big, as for the most restrictive == version requirement type almost 50% of dependencies experience lag, while for the least restrictive requirement type `latest` the percentage is only 25%.

Table 9 presents information on the lag time for each dependency requirement type and package manager. We excluded version requirement types with less than 50 uses (!= with 10 uses and ~> with 4 uses for CocoaPods; ..< with 30 uses for Swift PM). The overall trend for CocoaPods and Carthage is that dependencies with more restrictive version requirement types inhibit longer updating lag time. Exceptions to this rule are requirement types that are used relatively little, i.e., >= for Carthage. These results confirm the initial expectation that less restrictive version requirement types yield smaller lag time.

The results for Swift PM are not as conclusive. The lag times seem to be relatively similar for all four version requirement types.

Figure 3 (a) shows the evolution of lag time for the four most used version requirement types overall package managers. The biggest lag time can be observed for the latest and the exact version requirements.

Figures 3 (b)-(d), however, indicate that looking at each package manager separately, the latest requirement has one of the smallest lag times. For all package managers, the largest lag time corresponds to the exact version requirement. The results for Swift PM, again, are less conclusive and the lag time seems to converge for all version requirement types. Results for CocoaPods and Carthage, however, show that the choice of version requirement type has a noticeable effect on the lag

Table 9. Lag time in days for dependencies with updating lag.

PM	type	count	mean	std	25%	50%	75%	max
CocoaPods	==	907	207.5	280.6	16.6	69.4	296.8	1268.2
	`latest`	52054	89.6	167.3	10.5	34.0	96.1	2625.4
Carthage	==	5527	62.0	118.8	8.7	28.7	71.2	1827.0
	~>	11535	38.5	68.3	3.9	14.3	42.4	1146.8
	>=	299	48.8	68.6	3.6	15.3	53.3	270.4
	`latest`	2596	35.5	69.5	3.0	11.8	35.0	751.7
Swift PM	==	695	63.9	101.8	7.7	29.0	70.8	671.4
	~>	2232	55.6	67.0	8.0	30.1	81.1	613.3
	>=	4777	58.4	91.5	5.4	23.2	74.2	869.8
	`latest`	121	65.8	84.0	12.9	33.7	90.6	390.7

time.

Similar results can be seen for the median dependency updating lag time in Figure 6.2.3 with the exception of Swift PM where the median dependency lag time is similar between the requirement types, but not converging like the mean dependency lag time.

Third, we investigated how often library dependency versions are updated depending on the version requirement and if the library dependency version was extracted directly from the package manager resolution file or if it was resolved based on the manifest file. Figure 36 shows the proportions of library upgrade, downgrade, and no change for Carthage and Swift PM. We chose the most frequent library version requirement types for the respective package managers. From the data of both package managers, it is evident that library versions are upgraded more often when the package manager version resolution is rerun for each new project version (left column of each column pair). We have no results for CocoaPods, as all library dependency versions were extracted from package manager resolution files.

Fourth, we investigated how not updating library dependency versions can lead to more vulnerable projects. Figure 37 shows the number of projects with a direct dependency on a vulnerable library over time that could have been and could not have been fixed by a library dependency version upgrade. We found that 30% of the projects with a direct dependency on a vulnerable library could have been fixed by simply rerunning the package manager version resolution (3%) or by updating the library version in the manifest file and then rerunning the package manager version resolution (27%).

6.2.4. RQ 2.4: Spread of Vulnerabilities

We found a total of 149 known vulnerabilities in 61222 libraries. This corresponds to 24.3 vulnerabilities per 10000 libraries. We found that only 5.9% of connected

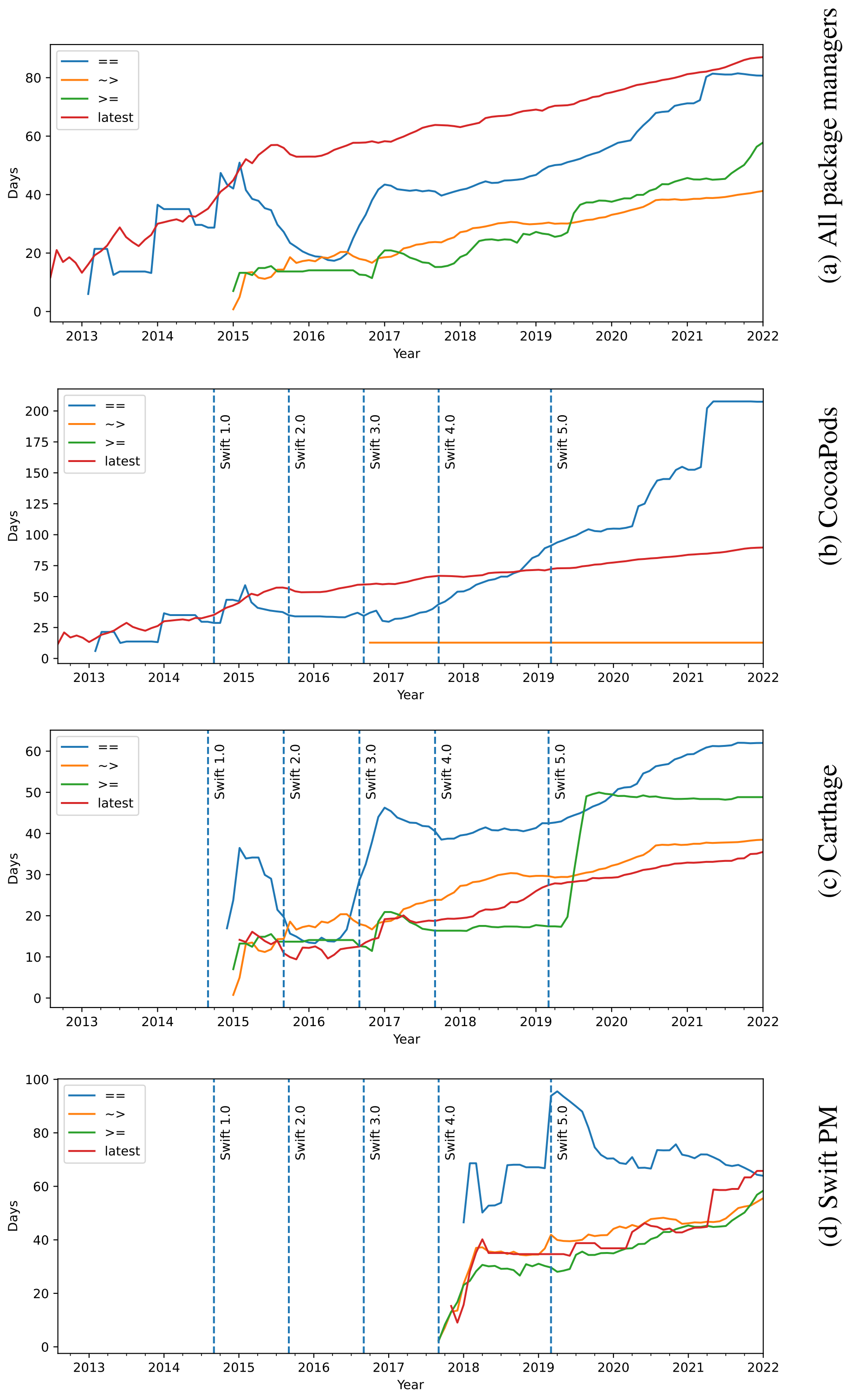


Figure 34. Mean dependency updating lag time per requirement type for (a) all package managers combined, (b) CocoaPods, (c) Carthage, and (d) Swift PM.

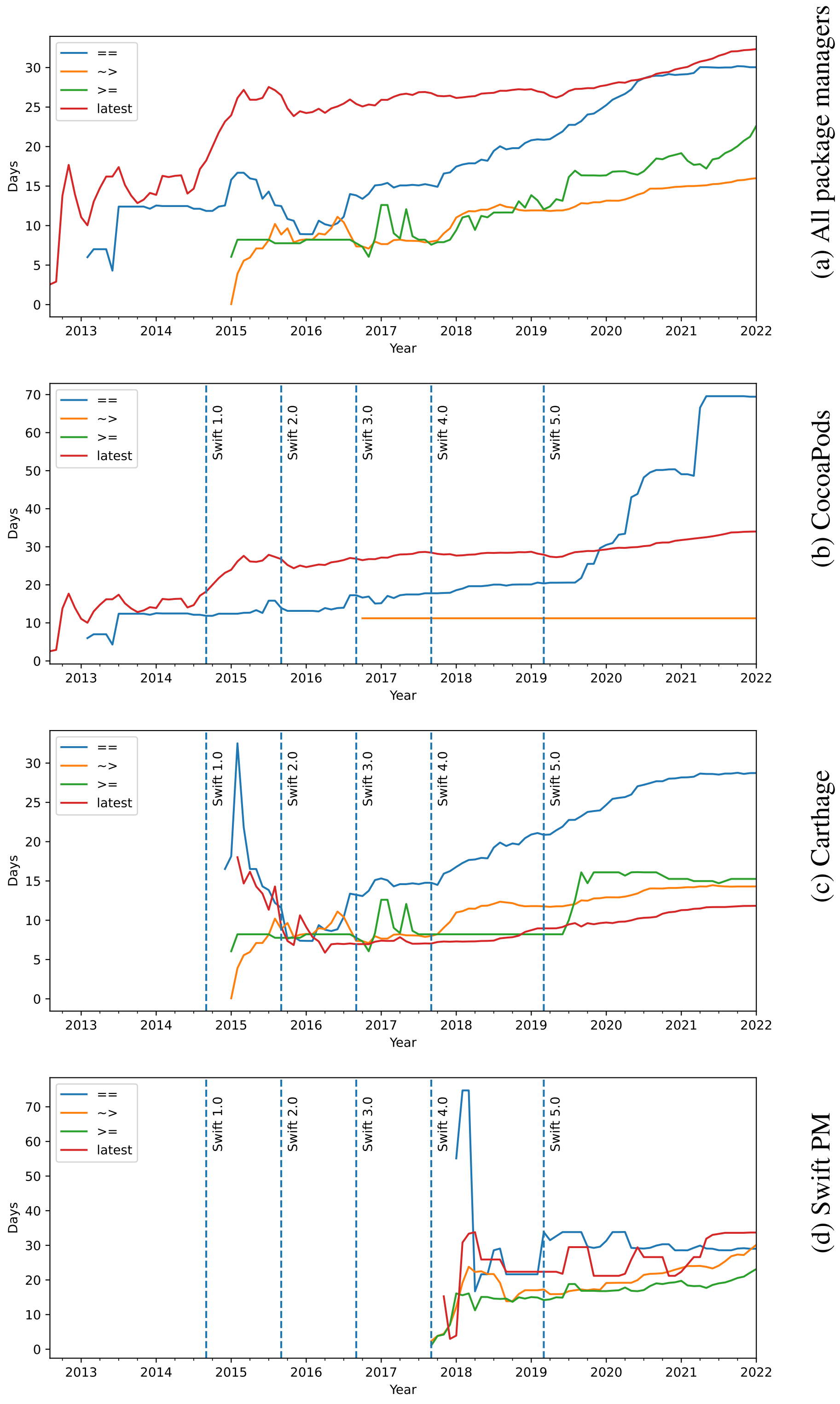


Figure 35. Median dependency updating lag time per requirement type for (a) all package managers combined, (b) CocoaPods, (c) Carthage, and (d) Swift PM.

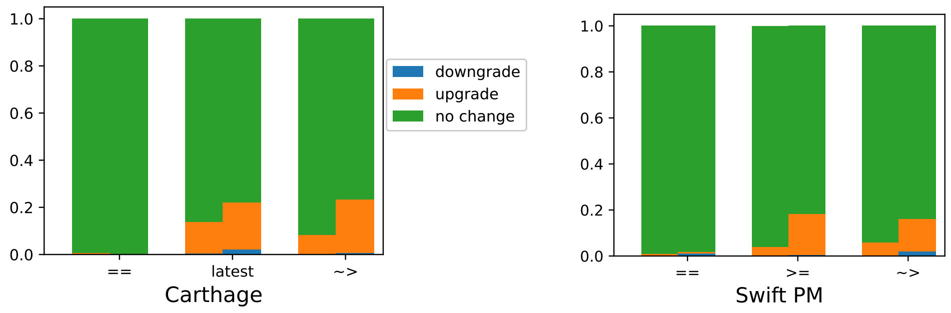


Figure 36. Proportions of downgrades and upgrades for each new project version for dependencies through Carthage and Swift PM (library version taken from resolution file is shown in the left column of each column pair).

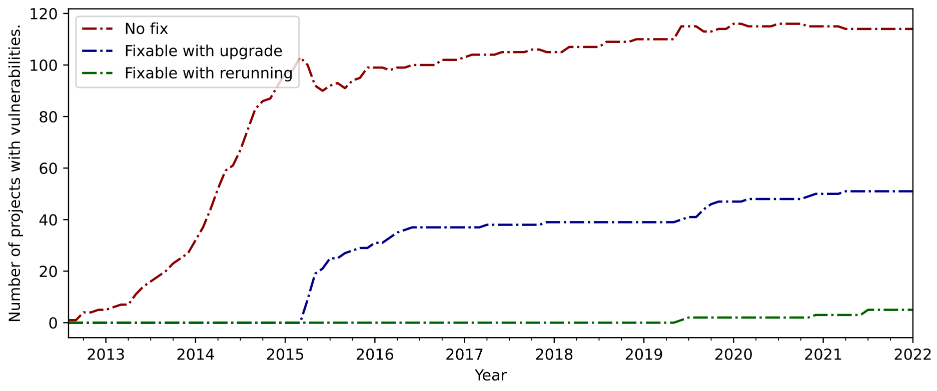


Figure 37. Number of projects with dependencies to vulnerable libraries that could not have been fixed by an upgrade of the dependency (red line) versus those that could have been fixed (blue and green lines).

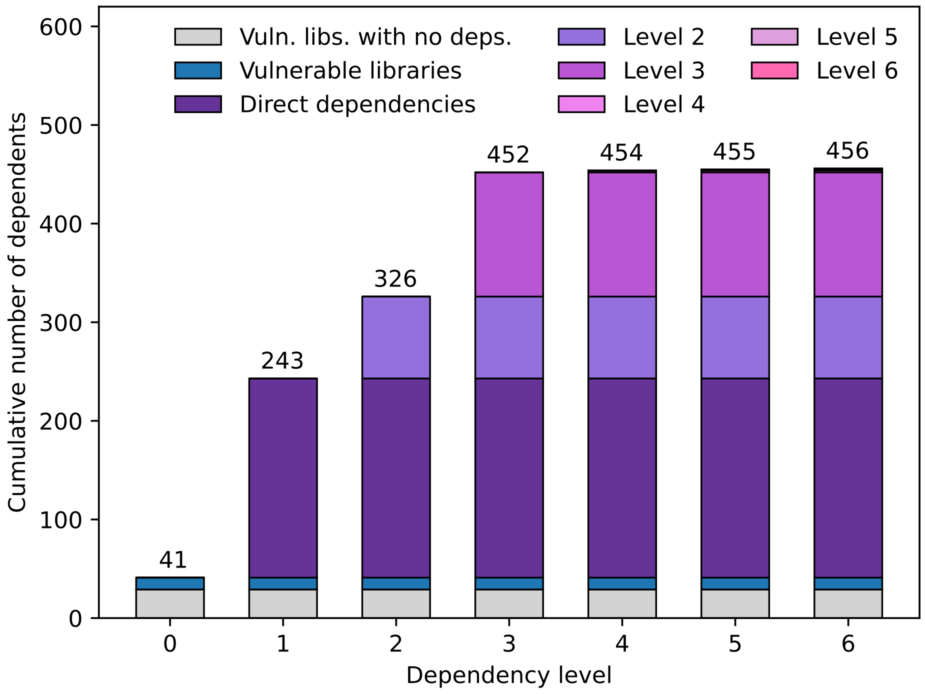


Figure 38. The cumulative number of libraries affected by vulnerabilities for each dependency level classified by the shortest dependency level to a vulnerable library version for each library.

libraries had dependencies to vulnerable library versions. For 3% of connected libraries, even the latest version of the library had a dependency on a vulnerable library version.

Figure 38 shows how publicly reported vulnerabilities propagate through the Swift library dependency network. There are 41 libraries with publicly reported vulnerabilities (dependency tree level 0). Of those libraries only 12 have dependents. There are 202 libraries without a publicly reported vulnerability that have a direct dependency on at least one vulnerable library version (dependency tree level 1). A considerable number of libraries are added on level two (83) and level three (126). Libraries with dependencies to multiple vulnerable library versions are counted at the lowest dependency tree level where a dependency to a vulnerable library version exists. In total, 415 libraries have dependencies on vulnerable library versions, and 456 libraries are affected by publicly reported vulnerabilities in total if we include libraries that are vulnerable themselves. Moreover, we can say that in case a library has at all a (possibly indirect) dependency on a vulnerable library version, then the longest chain to the first vulnerable library version has at most six levels in the dependency tree.

Table 10 shows the results of the analysis that explored whether the programming language in which a library is written has an influence on how vulnerabilities potentially spread through the dependency network. We determined the programming language of each vulnerable library based on data from GitHub on the main project language of the library. Table 10 indicates that most vulnerabilities originate in libraries written in C (88) and C++ (24). Libraries written in Swift and Objective-C contribute only 19 and three vulnerabilities, respectively. However, the highest impact on the Swift ecosystem comes from vulnerabilities in libraries written in Swift and Objective-C. Table 10 shows that vulnerable libraries written in Swift and Objective-C have significantly more dependents (98 for Swift and 313 for Objective-C) than projects written in other programming languages (56 and 14 for C and C++). Figure 39 shows how far vulnerabilities from libraries written in the different languages spread across the dependency network. Although there are some libraries that have dependencies to libraries written in C and C++, there are significantly longer dependency chains to libraries written in Swift and, especially, to libraries written in Objective-C. In the case of Objective-C dependency chains to vulnerable library versions can have up to 14 levels of indirection. Differently to Figure 38, libraries in Figure 39 are counted on each level of indirection they occur.

Table 10. Vulnerabilities by project language

Project language	vulnerabilities	libraries	dependent libraries
C	88	19	56
C++	24	8	14
Swift	**19**	**6**	**98**
Go	12	1	1
JavaScript	4	4	4
Objective-C	**3**	**3**	**313**

In the following, we present our results on how vulnerabilities of different severity propagate throughout the library dependency network. Vulnerabilities have four levels of severity: CRITICAL, HIGH, MEDIUM, and LOW. Table 11 provides information on the distribution of severity levels of the vulnerabilities found in the Swift ecosystem, as well as dependent libraries affected by these vulnerabilities. Most vulnerabilities (80) are of level HIGH, 31 and 37 vulnerabilities are CRITICAL and MEDIUM, respectively. Only one vulnerability has the level LOW. Most libraries (353) are affected by MEDIUM-level vulnerabilities through dependencies. Figure 40 shows how vulnerabilities with different severity levels propagate through the library dependency network. Vulnerabilities with severity level MEDIUM propagate the furthest through the dependency network. However, vulnerabilities with severity levels CRITICAL and HIGH can both be observed with levels of indirection in the dependency tree up to Level 5.

We analyze how many vulnerable dependencies could have been fixed via a dependency upgrade at the time a library version was released. For the upgradability analysis, we require that the version data for direct dependencies originates

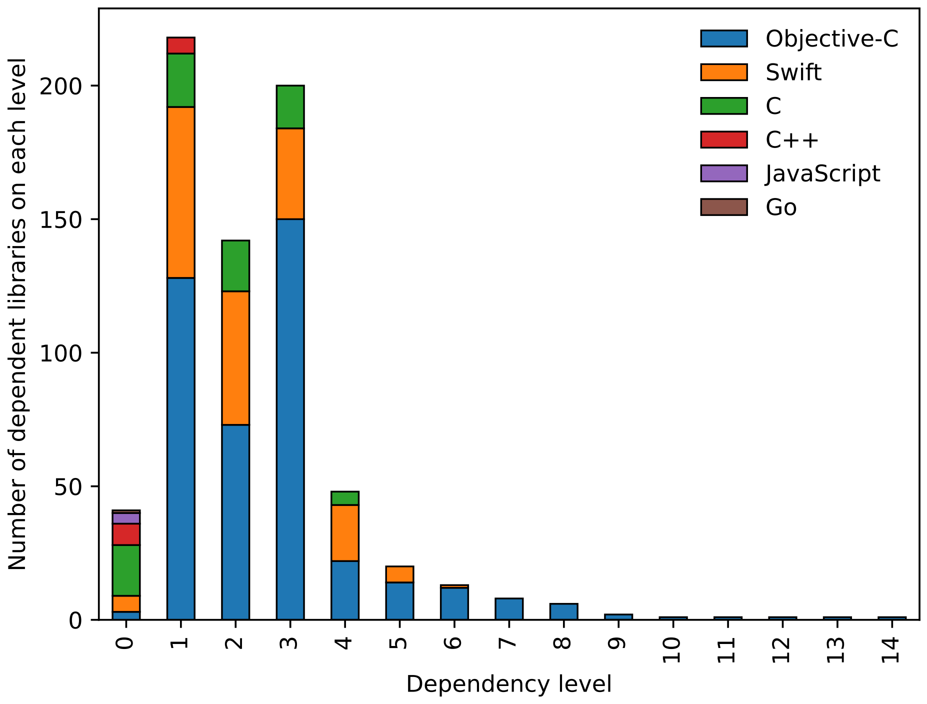


Figure 39. Number of libraries affected by vulnerabilities for each dependency level classified by main project language.

Table 11. Vulnerabilities by severity

Vulnerability severity	vulnerabilities	libraries	dependent libraries
CRITICAL	31	15	73
HIGH	80	31	136
MEDIUM	37	14	353
LOW	1	1	1

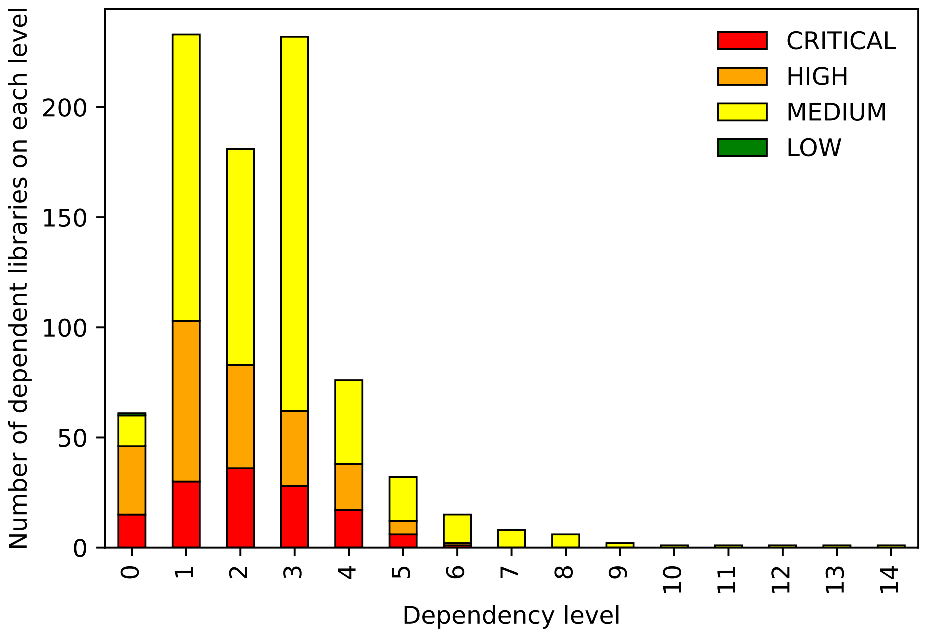


Figure 40. Number of libraries affected by vulnerabilities for each dependency level classified by severity level of the vulnerability.

from package manager resolution files and that the direct dependency is to a library included in the set of libraries available for our analysis. After filtering out dependencies that did not meet our criteria 341 out of 415 libraries with vulnerable dependencies remain. First, we analyze how updating direct dependencies would have fixed a vulnerable dependency for different levels of indirection. Figure 41 shows that 27% (498 of 1833 in total) of vulnerable direct dependencies could have been fixed via an upgrade. Furthermore, upgrading direct dependencies would also have fixed 16% (244 of 1555 in total) of second-level vulnerable dependencies and 64% (694 of 1082 in total) of third-level vulnerable dependencies. Note that the levels of the dependency tree in Figure 41 are greater than 0 and less than 10. They must be greater than 0 because there are no dependencies at level 0. There is no data beyond level 9 as the data on those library chains happened to not be compatible with the upgradability analysis.

Table 12. Vulnerable dependency fixes by severity

Vulnerability	all versions		latest version	
severity	fixed	not fixed	fixed	not fixed
CRITICAL	31%	69%	71%	29%
HIGH	33%	67%	52%	48%
MEDIUM	30%	70%	39%	61%

Next, we analyze how many vulnerable dependencies could have been fixed

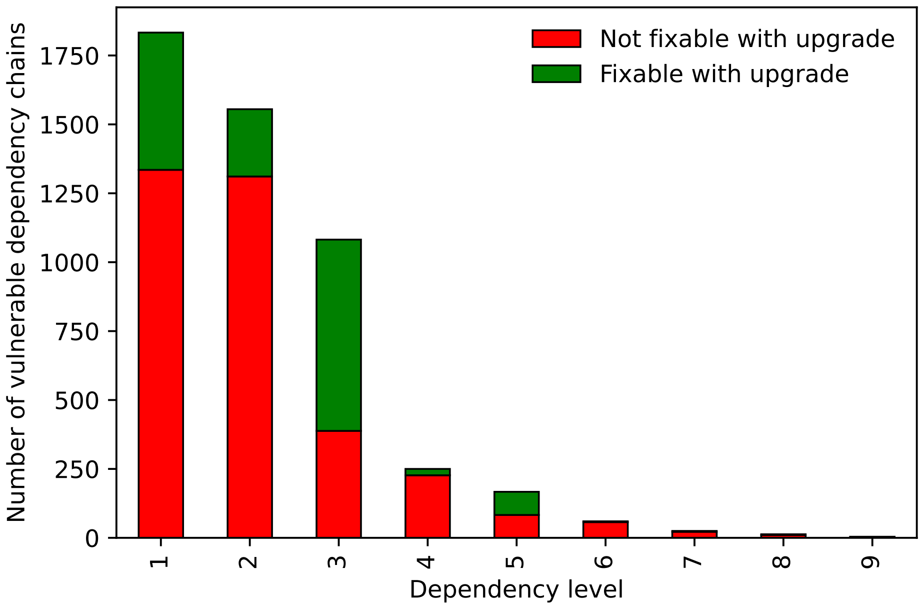


Figure 41. Number of dependency chains to vulnerable library versions that could be fixed (green) and not fixed (red) by an upgrade of the first dependency in the dependency chain. The numbers are shown for each dependency level.

via upgrading depending on the severity of the vulnerability. Table 12 shows that over all dependency chains, the probability of fixing the vulnerability via a dependency upgrade is around 30%. However, if we look at the latest version of each library, the percentage of fixing dependencies to critical vulnerabilities via upgrading is 71%, fixing dependencies to vulnerabilities of level HIGH is 52%, and fixing dependencies to vulnerabilities of level MEDIUM is 39%. Finally, we explore whether there are differences in the percentages of fixing vulnerabilities via upgrading between the project languages of the vulnerable libraries. Table 13 shows percentages of vulnerable dependencies being fixed via upgrading for each of the four most prominent languages. Over all library versions, the probability of fixing a vulnerable dependency is around 30%, with the exception of C++ where the probability is considerably smaller. For the latest versions of each library, the probability of fixing a vulnerable dependency via an upgrade is highest for C (67%) and Swift (60%), and lowest for Objective-C (38%) and C++ (33%). Looking at the success rates of fixing a vulnerable dependency via upgrading from the point of view of the different vulnerabilities in Table 14 we see that for 25% of vulnerabilities, the success of upgrading is over 89% and for another 25% of vulnerabilities the failure of fixing the dependency via an upgrade is over 94%. These numbers indicate that fixing a vulnerable dependency via upgrading is very successful for some of the vulnerabilities and not possible for others.

We checked whether the public descriptions of vulnerabilities in the Swift

Table 13. Vulnerable dependency fixes by project language

Project language	over all versions fixed	not fixed	latest version only fixed	not fixed
C	24%	76%	67%	33%
C++	6%	94%	33%	67%
Objective-C	30%	70%	38%	62%
Swift	36%	64%	60%	40%

Table 14. Percentage of dependents that can be fixed through upgrading per vulnerability

Percentage of dependents that	count	mean	std	min	25%	50%	75%	max
can be fixed	14	0.43	0.43	0.0	0.06	0.25	0.90	1.0
cannot be fixed	14	0.57	0.43	0.0	0.10	0.75	0.94	1.0

ecosystem include information on the class or method that contains the described vulnerability. This kind of detailed information could be used to fine-tune the analysis and detection of vulnerable dependencies by identifying the piece of code that contains the vulnerability. In situations where a library is dependent on a vulnerable library version, it might be good to know whether the code of the vulnerable library is used by the dependent library. Analyzing the description of each vulnerability and including data from patch links showed that most vulnerability descriptions do not include detailed enough information to determine the vulnerable class or method. Table 15 shows that not a single vulnerability in a project written in Swift specified both the vulnerable method and the vulnerable class. Similarly, very little information is available about vulnerabilities in projects written in Objective-C. There is more information on vulnerabilities in projects written in C and C++ but these vulnerabilities also affect significantly fewer libraries in the Swift ecosystem.

Table 15. Precision of public information on vulnerabilities

Project language	vulnerabilities	method	class	both
Swift	19	1	7	0
Objective-C	3	1	1	1
C	88	43	29	16
C++	24	18	15	12

6.3. Discussion

We discuss our findings in the following section.

6.3.1. RQ 2.1: Evolution of LDNs

From Figure 20 it is evident that some of the libraries in the Swift ecosystem were created before the introduction of the first package manager, CocoaPods, or before the release of the language Swift. Firstly, it is important to note that, although we call it the Swift ecosystem, it also encompasses Objective-C libraries. Objective-C, the predecessor of Swift, was introduced in 1984 and it is inter-operable with Swift. Additionally, libraries written in C and C++ can be used in both Swift and Objective-C projects. Some of these libraries are also available through these package managers. Some of the C/C++/Objective-C libraries written before the release of CocoaPods were later added to the package managers. In our analysis, we use git tags and commit timestamps to date library versions. Therefore, as we have no information on when a library or library version was added to a package manager, we simply assume that it was added when the library version was released. In the following, we will ignore any library versions added before September 2011, i.e., prior to when CocoaPods was introduced.

When Decan et al. [DMG19] analyzed the LDNs of seven package managers, they found that the number of libraries grew for each package manager either linearly or exponentially. Given these results, we expected to see a similar trend in the number of libraries and library versions for CocoaPods, Carthage, and Swift PM. However, the growth we observed has slowed in recent years. The decline in growth for CocoaPods and Carthage is also clear when looking at the number of new library versions added per month. To check if the number of updates per month for CocoaPods is really declining, we conducted an additional analysis of the git history of the CocoaPods Spec repository. In this analysis, we searched for the latest git commit for each month and then queried the difference between commits of consecutive months. Additions and deletions were recorded for each file. We discarded all file names that were not Podspec files and then counted the number of additions for each month. This analysis confirmed that the number of updates (i.e., file additions) was indeed falling. Our hypothesis is that more and more developers are moving to Swift PM. Therefore, Carthage has lost most of its appeal. First, when Apple introduced Swift PM it was a standalone terminal application that could be used to create macOS applications and packages. Furthermore, in 2019, Apple added support for iOS and Xcode, which is the official IDE for iOS and macOS development. Now a dependency can be declared through Swift PM by simply searching for a library in Xcode. This makes Swift PM the easiest-to-use package manager in the Swift ecosystem.

We expected the number of direct dependencies and total dependencies to grow over time as has been observed by Kikas et al. [Kik+17]. We observed that the number of direct dependencies was very volatile and growing rapidly between 2012 and 2016. After 2016 the number of direct dependencies stabilized and started growing slowly. Although the mean number of direct dependencies is consistently higher for CocoaPods than for Carthage, the trend is similar for all three package managers. Our hypothesis is that this change was a consequence of the introduction of Swift in 2014 and developers migrating from Objective-C to Swift in the following years. The growth of the mean number of dependencies after 2016 for all three studied package managers is lower than for JavaScript, Ruby, and Rust [Kik+17]. The mean number of direct dependencies is comparable to other package managers [DMG19], but the mean number of transitive dependencies is significantly lower. For example, the median number of transitive dependencies for a library available through the Cargo, NuGet, and npm package manager is 41, 27, and 21, respectively. In comparison, the median number of transitive dependencies in the LDNs of the Swift ecosystem is only two.

6.3.2. RQ 2.2: Evolution of Package Managers

We saw that the percentage of libraries not using a package manager decreased from 97.8% in 2012 to 84.3% in 2021. We compared this trend to other package managers by additionally calculating the percentage of libraries with dependencies over time for 10 package managers with sufficient dependency data in libraries.io[Lib22] (Maven, Packagist, NPM, CPAN, Hex, NuGet, Pub, Puppet, PyPi, and Rubygems). We found that for all 10 package managers the percentage of libraries with dependencies grew over time. Different from the Swift ecosystem, however, the general trend is significantly steeper with most libraries using package managers to declare dependencies by 2020. The only package managers with a slightly similar trend to the Swift ecosystem are Maven and PyPi with 43.1% and 28.5% of libraries declaring dependencies through a package manager for Maven and PyPi respectively. However, libraries in these package managers still declare dependencies more often than in the Swift ecosystem. The reason for this might be that developers in the Swift ecosystem seem rather conservative in terms of declaring dependencies, a sentiment that can also be observed in developer forums [kut22].

Three package managers are used in the Swift ecosystem: CocoaPods, Carthage, and Swift PM. We expected that the LDNs of these ecosystems overlap, but that there are also libraries that are available only through CocoaPods, Carthage, or Swift PM respectively. This assumption proved to be true. Another, more silent, assumption was that CocoaPods is the largest package manager. The assumption was based on the number of libraries reported by different sources, claiming the number of libraries served by CocoaPods to be around 89000 and the number of libraries served by Carthage and Swift PM to be around 4500 each. Our analysis showed that although this is true, the difference between the package managers is not as big as assumed. In 2016, 63% of libraries with dependencies used CocoaPods and 40% of libraries used Carthage. In 2021 48% of libraries with dependencies used CocoaPods and 47% of libraries used Swift PM.

An explanation for the smaller difference is that, while CocoaPods has an official central repository, Carthage and Swift PM do not. Therefore, it is not possible to gather all libraries served through Carthage and Swift PM. At the same time, the official CocoaPods repository has many incorrect references to libraries. When looking at libraries that either use or are referenced through a package manager the difference between package manager sizes is significantly smaller. This is an interesting insight for developers who might choose CocoaPods with the expectation of it providing access to 10 times more libraries. We saw that this expectation might not hold true for libraries that are referenced and popular.

Over time the popularity of CocoaPods remains stable. Some libraries switch from CocoaPods to other package managers, but the percentage of these libraries is relatively low. Many libraries, on the other hand, switch away from Carthage. Over time more project switch from Carthage to Swift PM than from Carthage to CocoaPods, which might be due to the underlying similarity of Carthage and Swift PM. Additionally Swift PM is integrated into Xcode, making it very easy to use for developers. In conclusion, the introduction of a new package manager does not necessarily make libraries switch to the newest package manager. Wether libraries switch between package managers is dependent on the features of the package managers involved. Unique features of a package manager, such as a central repository, can provide stable popularity among developers. If Apple wanted to bring more libraries to Swift PM, it might be beneficial to add some features that only exist for CocoaPods so far, for example, a centralized repository (or perhaps a repository for officially vetted libraries).

6.3.3. RQ 2.3: Upgrades vs Version Requirement Types

The lag time is similar for Carthage and Swift PM, but considerably larger for CocoaPods. For all package managers, the lag time is growing linearly with approximately the same speed. When Swift PM was released in the second half of 2017 it understandably started out with a near-zero lag time which then quickly rose to the same level as that of Carthage. A possible explanation for the growing lag time is that there are projects that never update their dependencies.

It is surprising that the version requirement type used with CocoaPods is almost always `latest`. In terms of how requirements are declared in the Podfile, this means that developers simply declare the library name without a version requirement. The official documentation for CocoaPods[7] suggest the use of ∼>, but it seems that developers prefer to use an even simpler notation.

For Carthage, version requirements are more restrictive. Over half of the time, a version requirement specifying the major or minor version is given. This corresponds to the recommendation in the Carthage documentation[8]. Surprisingly, an exact version is used relatively often (20%). Perhaps developers using Carthage prefer to be more in control of when a library version is installed and they update the exact version requirement manually when needed. Most library dependencies

[7]`https://guides.cocoapods.org/using/using-cocoapods.html`

[8]`https://github.com/Carthage/Carthage`

in Swift PM are declared with the >= requirement. This is the version requirement type suggested by the official documentation[9]. It can be seen as equivalent to the `latest` requirement in terms of potential dependency updating lag, as it does not limit new updates. We checked the version requirement types used in the README files of 30 popular libraries and found no connection between the version requirement type in the README and how developers declared library version requirements for these libraries. A possible explanation for the difference between the version requirement choices for CocoaPods and Carthage are that Carthage was created as an alternative to CocoaPods. Some developers did not like that using CocoaPods forced them to use the Xcode workspace generated by the package manager. Carthage was introduced as a more lightweight alternative that gave the developers more control over the app project and how the library dependencies were integrated. It might be that the desire for more control carried over to how developers declare their dependency requirements.

The results for Swift PM are less conclusive. It might be that the nature of Swift PM - not being solely a package manager but a build system - results in it being used differently than the other package managers. The lag time seems to converge for all version requirement types. Results for CocoaPods and Carthage, however, show that the choice of version requirement type has a noticeable effect on the lag time. This is an indication that developers should prefer less strict version requirements, where possible. If a less strict version requirement is not possible, then developers should update library dependency versions manually on a regular base to keep the lag time low. Another option is relying on an automatic dependency monitoring tool such as Dependabot or Renovatebot in case developers use a supported package manger.

Our results indicate that it is not enough if developers choose a library dependency version requirement type that allows for frequent automatic updates, potentially resulting in shorter lag time for dependency updates. It is also necessary to rerun the library dependency version resolution by running the package manager more often. There might be different reasons for developers not wanting to rerun the package manager version resolution, e.g., fear of incompatibilities, no time to check if everything works correctly, and forgetting about the need to rerun the package manager. Although sometimes problems can be introduced by upgrading library dependency versions, some of these concerns could be alleviated by using a more restrictive version requirement such as $\sim>$.

The results on upgrading vulnerable library versions show that, at least in terms of vulnerable dependencies, keeping the library dependency versions up to date results in safer projects. In this analysis, we did not consider transitive dependencies, but it is possible that the dependency updating lag accumulates over dependencies of dependencies and therefore results in even fewer projects that include the necessary security fixes. To better understand the scope of the project in the

[9] `https://www.swift.org/package-manager/`

Swift LDN we analyzed the vulnerability propagation in RQ 2.4.

6.3.4. RQ 2.4: Spread of Vulnerabilities

The total amount of 149 vulnerabilities in 61222 libraries in the Swift ecosystem corresponds to 24.3 vulnerabilities per 10000 libraries. This ratio is much higher than that for npm where Zimmermann et al. found the ratio to be around 8 in 2018 [Zim+19]. The difference between the two ecosystems could be due to the high number of very small libraries in npm. In contrast, Li et al. found the ratio to be 113.5 for Java projects [Li+21]. The Java ecosystem is older and might have larger libraries, but we do not have a definite reason for the big difference.

Our results show that only 5.9% of connected libraries have dependencies to vulnerable library versions. For 3% of connected libraries, its latest version is still dependent on a vulnerable library version. In contrast, Düsing et al[DH21] found that 9% of libraries in npm had direct dependencies to vulnerable libraries. Zimmermann et al. [Zim+19] found that 40% of npm projects they studied depended (directly or transitively) on vulnerable libraries. Alfadel et al. [Alf+20] found that 67% of all npm applications had at least one vulnerable direct dependency. Zerouali et al. [Zer+22] found that for more than 15% of npm and RubyGems libraries, the latest version of the library is directly dependent on a vulnerable library version. Additionally, they found that for 42.1% of all npm libraries and for 30% of RubyGems libraries the latest version of the library had a transitive dependency on a vulnerable library version. Therefore, in comparison to other ecosystems, the Swift ecosystem is considerably less affected by vulnerable library dependencies. A possible reason could be that libraries in the Swift ecosystem have fewer dependencies on average than libraries in other ecosystems, such as npm. Another possibility is that there are fewer vulnerabilities reported for the libraries in the Swift ecosystem but our analysis shows that this is not true, at least in comparison to npm.

Looking at the severity of the vulnerabilities, the vulnerabilities with severity level MEDIUM spread the most in the library dependency network. A possible explanation is that vulnerabilities with a MEDIUM severity level are not taken as seriously as vulnerabilities with higher severity levels and, therefore, are able to exist longer and spread further in the library dependency network. Most vulnerabilities in the Swift ecosystem originate from libraries written in C and C++. When looking at the impact on the whole library dependency network, however, vulnerabilities in libraries written in Swift and Objective-C spread considerably further. Libraries written in Swift and Objective-C have more dependents and therefore a higher impact on the overall library dependency network. Domınguez-Alvarez et al. [DGC] found that most libraries available through the CocoaPods package manager are written in Swift and Objective-C. It might be, that libraries written in C and C++ are very specialized and therefore not used by many other libraries.

Overall, around 30% vulnerable dependencies could have been fixed via an update of direct dependencies. Surprisingly, there is not much difference between vulnerability severity levels when looking at upgradability over all library versions. When looking at the latest version of each library, however, our results show that vulnerabilities with severity level CRITICAL could have been fixed in 70% of the cases. This is a strong indication for developers to keep up with library dependency upgrades as a means to avoid dependence on vulnerable libraries. If upgrading to each new version is not possible, developers should at least check if their dependencies have publicly reported vulnerabilities, for example using automated tooling such as SwiftDependencyCheker.

We did not check if vulnerable library dependencies could have been fixed via downgrading. Although this might be possible in some cases, it is rather difficult in the Swift ecosystem due to the backwards incompatibility of Swift. Additionally to this the affected versions data from NVD is not always accurate making it difficult to find the correct unaffected library version. Given the lack of detailed data on where the vulnerability is located it would also be difficult to correctly assert blame to a vulnerability introducing commit.

Currently, tools exist that can be used to check for vulnerable dependencies when using CocoaPods, Carthage, or Swift Package Manager. There are, however, no tools for the Swift ecosystem that could check if a vulnerability in a dependency really affects the developed application. Existing tools could be extended if detailed information about the exact code location of a vulnerability was available. Our results suggest that NVD does not include enough detailed information on vulnerabilities in libraries written in Swift and Objective-C. Therefore, the best solution for developers is to upgrade to a version of the library where the vulnerability has been fixed - if such a version is available.

6.4. Threats to Validity

We discuss threats to validity in the following section.

6.4.1. Construct Validity

We only look at libraries declared through package managers. It might be possible that some projects are using dependencies, but through other means (e.g. by manually downloading them). Our analysis is based on third-party libraries. Additional analyses are needed to confirm if our results can be generalized to all projects written in Swift, including apps.

Due to the dependency updating lag definition used, our analysis only takes into account library updating lag times in libraries that are actively developed. The updating lag time is calculated based on the library dependency versions that were available at the time the project version was released. This means that if a project is no longer updated its dependency lag time is not growing. The advantage of this approach is that projects that have been discontinued will not inflate the dependency updating lag time. Nevertheless, it is possible that some of these projects are still being used and could therefore result in other projects having outdated transitive dependencies.

In our analysis, we assume that every vulnerable dependency implies that the dependent library is (indirectly) vulnerable as well. However, the presence of vulnerable dependencies does not necessarily imply that the library is actually vulnerable. In a preliminary study [Zap+18] Zapata et al. analyzed dependencies of 60 projects using the npm package manager and showed that most projects with vulnerable dependencies do not actually use the vulnerable code. Hejderup et al. [Hej+22] analyzed libraries written in Rust and showed that not all resolved dependencies are really called, which means that dependencies to vulnerable libraries might not necessarily affect the library itself. Given that our results show that relatively few libraries depend on vulnerable libraries in the library dependency networks of the Swift ecosystem, a more detailed analysis would not affect this conclusion. A detailed analysis of call graphs might reduce the percentage of libraries with dependencies to vulnerable libraries even further. However, as we show in our answer to RQ3, the data needed for such an analysis is often not available.

Our analyses using information about the programming languages, in which the vulnerable libraries are written, depend on the information provided by GitHub about the main programming language. A library could, however, be written in several programming languages and the vulnerability itself could be located in code that was written in a programming language different from the main programming language. To understand the level of correctness of the information provided by GitHub in the context of our analysis, for those vulnerabilities where class/file level data was available, we also checked the language of the class/file and compared it to the main programming language indicated by GitHub. In only two of the 92 cases where such information is available, the vulnerability is located in a file written in a different language than the main programming language of the library. None of these two cases occurs in libraries classified as written in Objective-C or Swift.

6.4.2. Internal Validity

The Swift LDN dataset includes open-source libraries only. Additionally, some libraries were excluded as the repository contained no tags. The library dependency data mostly relies on package manager resolution files. Not every library that uses a package manager includes the corresponding resolution files in the repository. For such repositories, the package manager manifest files are parsed and the dependency requirements are resolved. Building the dependency graph by only declaring the exact version of a dependency means that transitive dependencies could in practice be resolved differently. When a transitive dependency is resolved at a later date then it is possible that the actual version of the transitive dependency would not match the version in our dataset. The data on the version ranges do, however, exist in the dataset and could be checked as future work.

For the vulnerability data, we rely on data from NVD. This means that we need to trust that the data is correct. It is possible that there are incorrect entries if they have not been checked by third parties. We do, however, believe the data to be reliable as it is an official and public database that is continuously reviewed and maintained.

6.4.3. External Validity

We claim that our results hold for all open source libraries in the LDNs of the Swift ecosystem, i.e. all open source libraries that are available through CocoaPods, Carthage, and Swift PM. For CocoaPods, the official repository that contains information on libraries available through CocoaPods was used. For Carthage and Swift PM, the information on libraries.io was used as the initial set of libraries. To make sure that newer libraries than 2020 are included and that we do not rely solely on libraries.io snowballing was used to analyze referenced dependencies that were not analyzed in our dataset yet. This additional step should ensure that we also include libraries that should be in the dataset but that did not exist in the initial set of libraries. We analyzed the LDNs of the Swift ecosystem between September 2011 and December 2021. We make no claims to how the LDNs might evolve in the future. We saw in our analysis that the introduction of a new package manager can disrupt any trends that might have existed before. All data and all tools are open source and available on public repositories. We tried to describe each manual process in as much detail as possible. Therefore, our study should be reproducible.

The vulnerability data in the dataset is based on public data from the NVD. When using other vulnerability databases, for example, Snyk, the results might be different. Vulnerabilities in NVD are publicly reported, which adds to the trustworthiness of the data.

7. CONCLUSION

Code quality has many dimensions that can be measured, among them, maintainability and security. Low maintainability can lead to high cost of maintaining and updating an application. Security impacts can lead to compromised systems, loss of trust to the owner of the application, and financial loss in a case of a data breach. From a user's perspective, higher maintainability can lead to a higher quality user experience, such as a more responsive user interface or less strain on the device battery. Given the data an average user entrusts to their smartphone, it is important to ensure the security of these applications.

Most smartphones run either Android or iOS operating systems, with iOS being the second most popular mobile operating system. So far the tool support to improve code quality for iOS applications, especially iOS applications written in Swift, has been lacking. Different aspects of code quality can be measured, two of these aspects are maintainability and security. With the aim to improve tool support for analysing the quality of iOS applications we implemented three open-source tools, validated these tools, and showed how the tools can be used to conduct interesting and insightful empirical analyses. We presented three contributions:

- **Contribution 1:** Tool support for analyzing the quality of iOS applications.
- **Contribution 2:** Empirical evidence on code smells in open-source iOS applications.
- **Contribution 3:** Extensive analysis of library dependency networks in the Swift ecosystem.

7.1. Tools

To improve tool support for analyzing code quality in iOS applications we implemented three open-source tools:

GraphifySwift, a first prototype that can be used to measure the maintainability of iOS applications by detecting common object-oriented code smells in Swift applications. GraphifyEvolution can be used by app developers to detect code smells in the applications they are developing and by researchers to analyze a bulk of source code repositories.

GraphfiyEvolution is a more advanced tool that builds on our experiences developing and using GraphifySwift. GraphfiyEvolution is a modular tool that can be used to analyze code smells and beyond. It is easily extendable and can be used to incorporate analysis results from many external tools. Additionally to the capability of analyzing a snapshot of a project, it allows analyzing the evolution of projects. This can be applied for example for code smell or vulnerability evolution analysis. Although GraphfiyEvolution is mostly aimed at researchers interested in analyzing the evolution of applications, it would be possible to extend the tool to, for example, visualize the architecture of an application, including code smells.

SwiftDependencyChecker, a lightweight tool that detects vulnerable library dependencies. It can be seamlessly integrated into the Xcode build process making it easy for developers to be notified if their library dependencies have publicly reported vulnerabilities. The option to include SwiftDependencyChecker in the Xcode build process allows developers to be notified about vulnerable library dependencies used without the need to actively run a detection tool.

7.2. Empirical Analyses

We showed how the tools can be used to conduct interesting and insightful empirical analyses. First, we applied GraphfiySwift to analyze code smells in open-source iOS applications (RQ1). Second, we applied GraphifyEvolution combined with SwiftDependencyChecker to analyze the evolution of the Swift library dependency network (RQ2).

7.2.1. Code Smells in Open-Source iOS Applications

We analyzed the frequency and distribution of object-oriented code smells in iOS applications. We compared code smell distributions on iOS and Android and found significant differences between the two platforms. Our results indicate that tool developers need to take platform-specific coding standards into account when building supporting software for developers.

7.2.2. Library Dependency Network Analysis

We created a LDN dataset with over 60 thousand libraries for the iOS ecosystem and analyzed its evolution. We saw that although the LDN is growing, the growth is not as fast as on many other platforms. We analyzed the evolution of the package managers in the Swift LDN and discussed properties of package managers that increase their acceptance by developers. These insights could be used by package manager developers to improve adoption of their tools. Our analysis shows that unique characteristics of package managers might influence the rate of adaption by users. Based on this we hypothesise that Apple could improve adaption of Swift PM by implementing features that are currently only available for CocoaPods, such as a central repository of libraries.

When analysing how vulnerabilities spread in the LDN of the iOS ecosystem, we found that libraries in this LDN are considerably less affected by vulnerable libraries than many other platforms, such as npm. This does not mean that there is no risk in using third-party libraries, but due to a smaller average number of transitive dependencies, this risk is lower than on other platforms. We showed that many projects could fix their vulnerable dependencies by upgrading the library dependency version motivating the need for tools such as SwiftDependencyChecker or GitHub Dependabot. Although these tools overestimate the number of vulnerabilities as vulnerable code from third-party libraries is not always used in the given project, it might not be feasible to increase the detail of analysis. We found that not enough public data exists on the publicly reported vulnerabilities in the third-party libraries in the iOS ecosystem that could be incorporated into dependency analysis tools without additional manual work. Developers can increase the security of their projects by keeping their library dependencies up to date and by using dependency analysis tools that detect vulnerable dependencies.

7.3. Future Research Directions

GraphifyEvolution provides many options for future analysis due to its modular nature. We have implemented two additional external analyzers: the extraction of import statements and the extraction of the file language. The idea behind the import statement analysis is threefold. First, we would like to analyze the use of system libraries, which can additionally be used for the classification of classes to fit them into architecture patterns. This would allow the analysis of how different architecture patterns (e.g. MVC, MVVM) evolve in an iOS project and how this evolution is connected to code smells. Second, we would like to analyze how often libraries in iOS are included in projects without using package managers. Including library dependencies manually would make these libraries invisible to analyses such as ours that relied on package manager metadata. Third, this analysis would allow us to see if and where the vulnerable library dependencies are used in a project. If no import statements correspond to a vulnerable library dependency, then most likely the project itself is not vulnerable. When a corresponding import statement exists, however, it would be possible to extend GraphifyEvolution to analyze if the vulnerable code is really called. Additionally, it would be possible to implement other code smell queries, for example for security-related code smells.

The dataset created by running GraphifyEvolution provides class, method, and variable-level information about a project. This data could be used to visualize code smells or other project-specific data. Two prototypes have been created [Sto21; Bad21] that analyze a Java project and visualize its evolution based on the output of GraphifyEvolution. Such visualizations could be extended to show the health of a codebase and alert developers or project managers if the maintainability of a project starts to decline.

The Swift LDN dataset we created can be used for additional analyses, e.g. analysis of updating patterns or switching between libraries. We analyze when and how developers switch between package managers and provided insights for tool developers on how they could increase the acceptance of their package managers. Similar analyses could be done on third-party libraries to see when and why developers switch from one third-party library to another. This could provide useful insights for library developers on how to keep existing users and how to gain new users.

BIBLIOGRAPHY

[Abd20] Sayed Abdelhafiz. *Arbitrary code execution on Facebook for Android through a download feature*. medium.com. `https://dphoeniixx.medium.com/arbitrary-code-execution-on-facebook-for-android-through-download-feature-fb6826e33e0f`. Last accessed Mar. 31, 2023. 2020.

[ACS23] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. "Empirical analysis of security vulnerabilities in python packages". In: *Empirical Software Engineering* 28.3 (2023), p. 59.

[AFT19] Umberto Azadi, Francesca Arcelli Fontana, and Davide Taibi. "Architectural smells detected by tools: a catalogue proposal". In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE. 2019, pp. 88–97.

[Alf+20] Mahmoud Alfadel et al. "On the Threat of npm Vulnerable Dependencies in Node. js Applications". In: (2020). URL: `https://doi.org/10.48550/arXiv.2009.09019`.

[Alf+21] Mahmoud Alfadel et al. "On the use of dependabot security pull requests". In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE. 2021, pp. 254–265.

[All+16] Kevin Allix et al. "Androzoo: Collecting millions of android apps for the research community". In: *Proceedings of the 13th international conference on mining software repositories*. 2016, pp. 468–471.

[ash23] ashishb. *A collection of android security related resources*. github.com. `https://github.com/ashishb/android-security-awesome` (accessed Apr. 23, 2023). 2023.

[Bad21] Turkhan Badalov. "Software Analytics: Visualization of Source Code Evolution". 2021.

[BCN19] Oliver A Blanthorn, Colin M Caine, and Eva M Navarro-López. "Evolution of communities of software: using tensor decompositions to compare software ecosystems". In: *Applied Network Science* 4 (2019), pp. 1–22.

[Blo19] JArchitect Blog. *Stable Abstractions Principle is your friend to fight the design rigidity.* Jan. 2019. URL: `https://javadepend.com/Blog/?p=585` (visited on 01/21/2020).

[Bog+21] Chris Bogart et al. "When and how to make breaking changes: Policies and practices in 18 open source software ecosystems". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30.4 (2021), pp. 1–56.

[BRH13] David Bowes, David Randall, and Tracy Hall. "The inconsistent measurement of message chains". In: *2013 4th International Work-*

shop on Emerging Trends in Software Metrics (WETSoM). IEEE. 2013, pp. 62–68.

[CA17] Luis Cruz and Rui Abreu. “Performance-based guidelines for energy efficient mobile applications”. In: *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE. 2017, pp. 46–57.

[CA19] Luis Cruz and Rui Abreu. “Improving energy efficiency through automatic refactoring”. In: *Journal of Software Engineering Research and Development* 7 (2019), pp. 2–1.

[Chi+21] Bodin Chinthanet et al. “Lags in the release, adoption, and propagation of npm vulnerability fixes”. In: *Empirical Software Engineering* 26 (2021), pp. 1–28.

[CM14] Alexander Chatzigeorgiou and Anastasios Manakos. “Investigating the evolution of code smells in object-oriented systems”. In: *Innovations in Systems and Software Engineering* 10.1 (2014), pp. 3–18.

[COH19] Filipe Roseiro Cogo, Gustavo A Oliva, and Ahmed E Hassan. “An empirical study of dependency downgrades in the npm ecosystem”. In: *IEEE Transactions on Software Engineering* 47.11 (2019), pp. 2457–2470.

[Cur23] David Curry. *App Store Data (2023).* businessofapps.com `https://www.businessofapps.com/data/app-stores/`. Last accessed 29 Sep. 2023. 2023. (Visited on 09/23/2023). (accessed: 29.09.2023).

[Dan+21] Andreas Dann et al. “Identifying challenges for oss vulnerability scanners-a study & test suite”. In: *IEEE Transactions on Software Engineering* 48.9 (2021), pp. 3613–3625.

[Dec+16] Alexandre Decan et al. “When GitHub meets CRAN: An analysis of inter-repository package dependency problems”. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 493–504.

[DeL17] Dave DeLong. *A Better MVC, Part 3: Fixing Massive View Controller*. Sept. 2017. URL: `https://davedelong.com/blog/2017/11/06/a-better-mvc-part-3-fixing-massive-view-controller/` (visited on 11/06/2017).

[Der+17] Erik Derr et al. “Keep me updated: An empirical study of third-party library updatability on android”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2187–2200.

[DGC] Daniel Domınguez-Alvarez, Alessandra Gorla, and Juan Caballero. “On the Usage of Programming Languages in the iOS Ecosystem”. In: ().

[DH21] Johannes Düsing and Ben Hermann. “Analyzing the Direct and Transitive Impact of Vulnerabilities onto Different Artifact Repositories". In: *Digital Threats: Research and Practice* (2021). URL: `https://doi.org/10.1145/3472811`.

[DM19] Alexandre Decan and Tom Mens. "What do package dependencies tell us about semantic versioning?" In: *IEEE Transactions on Software Engineering* 47.6 (2019), pp. 1226–1240.

[DMC17] Alexandre Decan, Tom Mens, and Maëlick Claes. "An empirical comparison of dependency issues in OSS packaging ecosystems". In: *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE. 2017, pp. 2–12.

[DMC18a] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the evolution of technical lag in the npm package dependency network". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, pp. 404–414.

[DMC18b] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the impact of security vulnerabilities in the npm package dependency network". In: *Proceedings of the 15th international conference on mining software repositories*. 2018, pp. 181–191.

[DMG19] Alexandre Decan, Tom Mens, and Philippe Grosjean. "An empirical comparison of dependency network evolution in seven software packaging ecosystems". In: *Empirical Software Engineering* 24.1 (2019), pp. 381–416.

[DYA13] LaShanda Dukes, Xiaohong Yuan, and Francis Akowuah. "A case study on web application security testing with tools and manual testing". In: *2013 Proceedings of IEEE Southeastcon*. IEEE. 2013, pp. 1–6.

[Ell20] Tom Elliott. *Swift Package Manager for iOS*. `https://www.raywenderlich.com/7242045-swift-package-manager-for-ios` (accessed Jan. 21, 2022). 2020.

[Enc+11] William Enck et al. "A study of android application security." In: *USENIX security symposium*. Vol. 2. 2. 2011.

[Fow18] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[GFS21] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. "Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities". In: *Journal of Systems and Software* 172 (2021), p. 110653.

[GGN17] Mohammad Ghafari, Pascal Gadient, and Oscar Nierstrasz. "Security smells in Android". In: *2017 IEEE 17Th international working conference on source code analysis and manipulation (SCAM)*. IEEE. 2017, pp. 121–130.

[GJW14] Marion Gottschalk, Jan Jelschen, and Andreas Winter. "Saving Energy on Mobile Devices by Refactoring." In: *EnviroInfo*. 2014, pp. 437–444.

[Gon+17] Jesus M Gonzalez-Barahona et al. "Technical lag in software compilations: Measuring how outdated a software deployment is". In: *Open Source Systems: Towards Robust Practices: 13th IFIP WG 2.13 International Conference, OSS 2017, Buenos Aires, Argentina, May 22-23, 2017, Proceedings 13*. Springer International Publishing. 2017, pp. 182–192.

[GSM17] Aakanshi Gupta, Bharti Suri, and Sanjay Misra. "A systematic literature review: Code bad smells in Java source code". In: *International Conference on Computational Science and Its Applications*. Springer. 2017, pp. 665–682.

[GSS13] SG Ganesh, Tushar Sharma, and Girish Suryanarayana. "Towards a Principle-based Classification of Structural Design Smells." In: *Journal of Object Technology* 12.2 (2013), pp. 1–1.

[Hab+17] Sarra Habchi et al. "Code Smells in iOS Apps: How do they compare to Android?" In: *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE. 2017, pp. 110–121.

[Hec+15a] Geoffrey Hecht et al. "Detecting antipatterns in android apps". In: *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*. IEEE Press. 2015, pp. 148–149.

[Hec+15b] Geoffrey Hecht et al. "Tracking the software quality of android applications along their evolution (t)". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 236–247.

[Hec15] Geoffrey Hecht. "An approach to detect Android antipatterns". In: *Proceedings of the 37th International Conference on Software Engineering - Volume 2*. IEEE Press. 2015, pp. 766–768.

[Hej+22] Joseph Hejderup et al. "Präzi: from package-based to call-based dependency networks". In: *Empirical Software Engineering* 27.5 (2022), p. 102. DOI: 10.1007/s10664-021-10071-9.

[HG22] Joseph Hejderup and Georgios Gousios. "Can we trust tests to automate dependency updates? a case study of java projects". In: *Journal of Systems and Software* 183 (2022), p. 111097.

[HMR19] Sarra Habchi, Naouel Moha, and Romain Rouvoy. "The rise of android code smells: Who is to blame?" In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 445–456.

[HRM19] Sarra Habchi, Romain Rouvoy, and Naouel Moha. "On the survival of android code smells in the wild". In: *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE. 2019, pp. 87–98.

[Hua+19] Jie Huang et al. "Up-to-crash: Evaluating third-party library updatability on android". In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, pp. 15–30.

[Hud19] Paul Hudson. *How to refactor massive view controllers*. July 2019. URL: `https://www.hackingwithswift.com/articles/159/how-to-refactor-massive-view-controllers` (visited on 04/07/2019).

[Ils22] Michael Ilseman. *Swift ABI Stability Manifesto*. github.com. `https://github.com/apple/swift/blob/main/docs/ABIStabilityManifesto.md` (accessed Aug. 17, 2022). 2022.

[ITW21] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. "A comparative study of vulnerability reporting by software composition analysis tools". In: *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2021, pp. 1–11.

[KDG09] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. "An exploratory study of the impact of code smells on software change-proneness". In: *2009 16th Working Conference on Reverse Engineering*. IEEE. 2009, pp. 75–84.

[Kha15] Soroush Khanlou. *Massive View Controller*. Dec. 2015. URL: `https://khanlou.com/2015/12/massive-view-controller/` (visited on 12/23/2015).

[Kik+17] Riivo Kikas et al. "Structure and evolution of package dependency networks". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (2017), pp. 102–112. DOI: `10.1109/MSR.2017.55`.

[Kor+20] Gizem Korkmaz et al. "Modeling the impact of Python and R packages using dependency and contributor networks". In: *Social Network Analysis and Mining* 10 (2020), pp. 1–12.

[Kul+18] Raula Gaikovina Kula et al. "Do developers update their library dependencies?" In: *Empirical Software Engineering* 23.1 (2018), pp. 384–417.

[Kum15] Mohit Kumar. *Critical SSL Vulnerability Leaves 25,000 iOS Apps Vulnerable to Hackers*. thehackernews.com. `https://thehackernews.com/2015/04/ssl-vulnerability-iOS-security.html` (accessed Jul. 4, 2022). 2015.

[kut22] kutjelul. *Are iOS developers more purist than other types of software engineers?* reddit.com. `https://www.reddit.com/r/`

iOSProgramming/comments/ugvrta/are_ios_developers_more_purist_than_other_types/ (accessed: March. 01, 2023). 2022.

[Law19] Raymond Law. *Clean Swift iOS Architecture for Fixing Massive View Controller*. Feb. 2019. URL: https://clean-swift.com/clean-swift-ios-architecture/ (visited on 02/19/2019).

[Leh80] M.M. Lehman. “Programs, life cycles, and laws of software evolution”. In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. DOI: 10.1109/PROC.1980.11805.

[Li+21] Qiang Li et al. “PDGraph: A Large-Scale Empirical Study on Project Dependency of Security Vulnerabilities”. In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2021), pp. 161–173.

[Lib22] Libraries.io. *Supported Package Managers*. https://libraries.io (accessed: Feb. 17, 2022). 2022.

[Lin+14] Mario Linares-Vásquez et al. “Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in Java mobile apps”. In: *Proceedings of the 22nd International Conference on Program Comprehension*. ACM. 2014, pp. 232–243.

[LM07] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[Man+16] Umme Ayda Mannan et al. “Understanding code smells in Android applications”. In: *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE. 2016, pp. 225–236.

[Mee18] Mary Meeker. *Internet Trends 2018*. https://www.kleinerperkins.com/perspectives/internet-trends-report-2018/. Last accessed 20 Sep 2019. 2018. (Visited on 09/20/2019). (accessed: 20.09.2019).

[Mir20] Mehdi Mirzaie. *What is Massive View Controller and How to avoid it in Swift*. Sept. 2020. URL: https://medium.com/divar-mobile-engineering/what-is-massive-view-controller-and-how-to-avoid-it-in-swift-2fca3e7dc00 (visited on 09/20/2020).

[MM18] Bruno Gois Mateus and Matias Martinez. “An empirical study on quality of Android applications written in Kotlin language”. In: *Empirical Software Engineering* (2018), pp. 1–38.

[Moh+09] Naouel Moha et al. “Decor: A method for the specification and detection of code and design smells”. In: *IEEE Transactions on Software Engineering* 36.1 (2009), pp. 20–36.

[MP17] Samim Mirhosseini and Chris Parnin. “Can automated pull requests encourage software developers to upgrade out-of-date dependencies?” In: *2017 32nd IEEE/ACM international conference on automated software engineering (ASE)*. IEEE. 2017, pp. 84–94.

[MP21] Fabio Massacci and Ivan Pashchenko. “Technical leverage in a software ecosystem: Development opportunities and security risks”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 1386–1397.

[Mun05] Matthew James Munro. “Product metrics for automatic identification of" bad smell" design problems in java source-code”. In: *11th IEEE International Software Metrics Symposium (METRICS’05)*. IEEE. 2005, pp. 15–15.

[Ngu+20] Duc Cuong Nguyen et al. “Up2dep: Android tool support to fix insecure code dependencies”. In: *Annual Computer Security Applications Conference*. 2020, pp. 263–276.

[Och+22] Lina Ochoa et al. “Breaking bad? Semantic versioning and impact of breaking changes in Maven Central: An external and differentiated replication study”. In: *Empirical Software Engineering* 27.3 (2022), p. 61.

[Olb+09] Steffen Olbrich et al. “The evolution and impact of code smells: A case study of two open source systems”. In: *2009 3rd international symposium on empirical software engineering and measurement*. IEEE. 2009, pp. 390–400.

[OWA16] OWASP. *M5: Insufficient Cryptography*. owasp.org. `https://owasp.org/www-project-mobile-top-10/2016-risks/m5-insufficient-cryptography` (accessed Mar. 3, 2022). 2016.

[OWA23a] OWASP. *OWASP Android Security Inspector Toolkit*. owasp.org. `https://owasp.org/www-project-android-security-inspector-toolkit/` (accessed Apr. 23, 2023). 2023.

[OWA23b] OWASP. *Source Code Analysis Tools*. owasp.org. `https://owasp.org/www-community/Source_Code_Analysis_Tools` (accessed Apr. 15, 2023). 2023.

[Pai+17] Thanis Paiva et al. “On the evaluation of code smells and detection tools”. In: *Journal of Software Engineering Research and Development* 5.1 (2017), pp. 1–28.

[Pal+13] Fabio Palomba et al. “Detecting bad smells in source code using change history information”. In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press. 2013, pp. 268–278.

[Pal+14] Fabio Palomba et al. “Mining version histories for detecting code smells”. In: *IEEE Transactions on Software Engineering* 41.5 (2014), pp. 462–489.

[Pal+17] Fabio Palomba et al. “Lightweight detection of android-specific code smells: The adoctor project”. In: *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE. 2017, pp. 487–491.

[Pal+18] Fabio Palomba et al. “On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation”. In: *Empirical Software Engineering* 23.3 (2018), pp. 1188–1221.

[Pas+18] Ivan Pashchenko et al. “Vulnerable open source dependencies: Counting those that matter”. In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2018, pp. 1–10.

[Per18] Sarah Perez. *App Store generated 93% more revenue than Google Play in Q3*. techcrunch.com `https://techcrunch.com/2018/10/11/app-store-generated-93-more-revenue-than-google-play-in-q3/`. Last accessed 20 Sep. 2019. 2018. (Visited on 09/20/2019). (accessed: 20.09.2019).

[PPS18] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. “Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, pp. 449–460.

[PPS20] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. “Detection, assessment and mitigation of vulnerabilities in open source dependencies”. In: *Empirical Software Engineering* 25.5 (2020), pp. 3175–3215.

[Pra+21] Gede Artha Azriadi Prana et al. “Out of sight, out of mind? How vulnerable dependencies affect open-source projects”. In: *Empirical Software Engineering* 26 (2021), pp. 1–34.

[RAC21] Sofia Reis, Rui Abreu, and Luis Cruz. “Fixing vulnerabilities potentially hinders maintainability”. In: *Empirical Software Engineering* 26 (2021), pp. 1–27.

[RP20a] Kristiina Rahkema and Dietmar Pfahl. “Comparison of Code Smells in iOS and Android Applications.” In: *QuASoQ@ APSEC*. 2020, pp. 79–86.

[RP20b] **Rahkema, Kristiina** and Dietmar Pfahl. “Empirical study on code smells in ios applications”. In: *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*. 2020, pp. 61–65.

[RP21] Kristiina Rahkema and Dietmar Pfahl. “GraphifyEvolution-A Modular Approach to Analysing Source Code Histories”. In: *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*. IEEE. 2021, pp. 24–27.

[RP22a] Kristiina Rahkema and Dietmar Pfahl. “Analysing the Relationship Between Dependency Definition and Updating Practice When Using Third-Party Libraries”. In: *Product-Focused Software Process Improvement: 23rd International Conference, PROFES 2022, Jyväskylä, Finland, November 21–23, 2022, Proceedings*. Springer. 2022, pp. 90–107.

[RP22b] Kristiina Rahkema and Dietmar Pfahl. “Dataset: dependency networks of open source libraries available through CocoaPods, Carthage and Swift PM”. In: *Proceedings of the 19th International Conference on Mining Software Repositories*. 2022, pp. 393–397.

[RP22c] Kristiina Rahkema and Dietmar Pfahl. “SwiftDependencyChecker: detecting vulnerable dependencies declared through CocoaPods, carthage and swift PM”. In: *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*. 2022, pp. 107–111.

[RP23a] Kristiina Rahkema and Dietmar Pfahl. “Analysis of Library Dependency Networks of Package Managers Used in iOS Development”. In: *Proceedings of the 10th IEEE/ACM International Conference on Mobile Software Engineering and Systems*. 2023, pp. 23–27.

[RP23b] Kristiina Rahkema and Dietmar Pfahl. “Vulnerability Propagation in Package Managers Used in iOS Development”. In: *Proceedings of the 10th IEEE/ACM International Conference on Mobile Software Engineering and Systems*. 2023, pp. 60–69.

[Sal+18] Pasquale Salza et al. “Do developers update third-party libraries in mobile apps?” In: *Proceedings of the 26th Conference on Program Comprehension*. 2018, pp. 255–265.

[Sal+20] Pasquale Salza et al. “Third-party libraries in mobile apps: When, how, and why developers update them”. In: *Empirical Software Engineering* 25 (2020), pp. 2341–2377.

[Sha19] Tushar Sharma. “Extending Maintainability Analysis Beyond Code Smells”. PhD thesis. 2019.

[Sta22] StatCounter. *Mobile Operating System Market Share North America*. statcounter.com. `https://gs.statcounter.com/os-market-share/mobile/north-america/2022`. Last accessed Mar. 26, 2023. 2022.

[Sta23] StatCounter. *Mobile Operating System Market Share Worldwide*. statcounter.com. `https://gs.statcounter.com/os-market-`

share / mobile / worldwid / 2022. Last accessed Mar. 26, 2023. 2023.

[Sto21] Miron Storožev. “Exploration of Techniques to Visualise Code Quality”. 2021.

[Str+20] Jacob Stringer et al. “Technical lag of dependencies in major package managers”. In: *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2020, pp. 228–237.

[Suw+17] Hirohiko Suwa et al. “An Analysis of Library Rollbacks: A Case Study of Java Libraries”. In: *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*. IEEE. 2017, pp. 63–70.

[Tea14] Ptidej Team. *TraditionBreaker.rules*. bitbucket.org. https://bitbucket.org/ptidejteam/ptidej-5/src/a4560ec05f549802aa65d5a41841961b91773dbf/SAD%20Rules%20Creator/rsc/081009/TraditionBreaker.rules?at=master#TraditionBreaker.rules-1:2 (accessed January 21, 2020). 2014.

[tea19] Appcoda editorial team. *Avoiding Massive View Controller using Containment & Child View Controller*. Jan. 2019. URL: https://www.appcoda.com/container-view-controller/ (visited on 01/04/2019).

[Tea22] Microsoft 365 Defender Research Team. *Vulnerability in TikTok Android app could lead to one-click account hijacking*. microsoft.com. https://www.microsoft.com/en-us/security/blog/2022/08/31/vulnerability-in-tiktok-android-app-could-lead-to-one-click-account-hijacking/. Last accessed Mar. 31, 2023. 2022.

[Tuf+15] Michele Tufano et al. “When and why your code starts to smell bad”. In: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press. 2015, pp. 403–414.

[Tuf+17] Michele Tufano et al. “When and why your code starts to smell bad (and whether the smells go away)”. In: *IEEE Transactions on Software Engineering* 43.11 (2017), pp. 1063–1088.

[Ver13] Daniël Verloop. “Code smells in the mobile applications domain”. In: (2013).

[Wha19] WhatsApp. *WhatsApp Security Advisories*. whatsapp.com. https://www.whatsapp.com/security/advisories/archive/. Last accessed Mar. 31, 2023. 2019.

[WR21a] James Wetter and Nicky Ringland. *Understanding the Impact of Apache Log4j Vulnerability*. security.googleblog.com. https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html (accessed: Apr. 13, 2022). 2021.

[WR21b] James Wetter and Nicky Ringland. *Understanding the Impact of Apache Log4j Vulnerability*. Jan. 2021. URL: `https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html`.

[Yar23] Yarn. *Migrating from npm*. yarnpkg.com. `https://classic.yarnpkg.com/lang/en/docs/migrating-from-npm/` (accessed: Apr. 20, 2023). 2023.

[Zap+18] Rodrigo Elizalde Zapata et al. "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2018, pp. 559–563.

[Zer+18] Ahmed Zerouali et al. "An empirical analysis of technical lag in npm package dependencies". In: *New Opportunities for Software Reuse: 17th International Conference, ICSR 2018, Madrid, Spain, May 21-23, 2018, Proceedings 17*. Springer. 2018, pp. 95–110.

[Zer+19] Ahmed Zerouali et al. "A formal framework for measuring technical lag in component repositories—and its application to npm". In: *Journal of Software: Evolution and Process* 31.8 (2019), e2157.

[Zer+22] Ahmed Zerouali et al. "On the impact of security vulnerabilities in the npm and rubygems dependency networks". In: *Empirical Software Engineering* 27.5 (2022), pp. 1–45.

[Zha+08] Min Zhang et al. "Improving the precision of fowler's definitions of bad smells". In: *2008 32nd Annual IEEE Software Engineering Workshop*. IEEE. 2008, pp. 161–166.

[Zim+19] Markus Zimmermann et al. "Small world with high risks: A study of security threats in the npm ecosystem". In: *28th USENIX Security Symposium (USENIX Security 19)* (2019), pp. 995–1010.

Appendix A. CODE SMELLS

A.1. Code Smell Thresholds

Table 16. Thresholds used in replicating [Hab+17]

metric	vhigh
number_of_attribues	15
number_of_methods	18.5
number_of_instructions_class	375
class_complexity	41.5
number_of_instructions_method	10.4
number_of_methods_in_interface	6

Table 17. Thresholds calculated using the box plot technique for iOS apps

metric	Q1	median	Q3	very_high	max
number_of_attribues	1.0	2.0	5.0	11.0	182
number_of_methods	1.0	2.0	6.0	13.5	136
number_of_instructions_ class	5.0	21.0	66.0	157.5	2939
number_of_comments	7.0	10.0	17.25	32.625	225
class_complexity	1.0	4.0	15.0	36.0	466
complexity_method_ratio	1.0	1.33	3.0	6.0	48.0
coupling_between_ object_classes	0.0	0.0	0.0	0.0	8
number_of_methods_ and_attributes	2.0	5.0	10.0	22.0	269
lack_of_cohesion_ in_methods	0.0	1.0	9.0	22.5	6124
cyclomatic_complexity	1.0	1	3.0	6.0	156
number_of_direct_calls	0.0	0	1.0	2.5	46
number_of_callers	0.0	0	0.0	0.0	89
number_of_instructions_ method	3.0	7	15.0	33.0	2620
number_of_parameters	0.0	1	1.0	2.5	10
number_of_ chained_message_calls	0.0	1	2.0	5.0	8
number_of_ switch_statement	0.0	0	0.0	0.0	13
number_of_methods_ in_interface	1	1	2	3.5	32

Table 18. Thresholds calculated using the box-plot technique for Android

metric	Q1	med	Q3	vhigh	max
number_of_attributes	1	2	7	16	1482
number_of_methods	2	5	11	24.5	430
number_of_instructions_class	16	45	115	263.5	19215
class_complexity	3	9	22	50.5	3906
complexity_method_ratio	1	1.47	2.22	46	653.5
coupling_between_object_classes	0	1	3	7.5	83
number_of_methods_and_attributes	4	8	18	39	1486
lack_of_cohesion_in_methods	0	4	26	65	72643
number_of_calls_between_classes	2	4	8	17	1120
cyclomatic_complexity	1	1	2	3.5	1301
number_of_called_methods	0	1	2	5	4560
number_of_callers	0	0	1	2.5	892
number_of_instructions_method	3	4	10	20.5	12701
number_of_parameters	0	1	1	2.5	49
number_of_methods_in_interface	1	2	4	8.5	132
primitive_variable_use	1	2	3	6	454

A.2. Code Smell Definitions

Code smell name *alternative names (*split smells)*	Included in [Fow18]	[Man+16]	[Hab+17]	Code smell description
AlternativeClasses-WithDifferent-Interfaces	x			Matches classes that have at least a minimum count of methods in common that have the same types of parameters (considering methods with at least a minimum number of parameters), but where the classes do not extend/implement the same parent class/protocol and do not extend/implement each other [Fow18]. Thresholds: *minimum_common_method_count* is 2, *minimum_number_of_parameters* is 2.
BlobClass		x		Query matches all classes where *lack_of_cohesion_in_methods* is very high, *number_of_methods* is very high and *number_of_attributes* is very high. [Hec+15]. Thresholds: determined using the box-plot technique.
LargeClass	x			
BLOB			x	
BrainMethod		x		Matches methods with high *cyclomatic_complexity*, a max nesting depth of several, many accessed variables that belong to classes with high *number_of_instructions* [LM07]. Thresholds: *max_nesting_depth_of_several* set to 3, *many_accessed_variables* is short-term memory cap set to 7, other thresholds determined using the box-plot technique.
Comments	x			Matches all classes where *number_of_comments* is high [Fow18]. Thresholds: determined using box-plot technique.

Code smell name *alternative names (*split smells)*	Included in			Code smell description
	[Fow18]	[Man+16]	[Hab+17]	
ComplexClass *CC*			x	Matches classes that have very high *class_complexity* [Hec+15]. Thresholds: determined using box-plot technique.
CyclicClass-Dependency *CyclicDependency*		x		Matches all cycles between Class-Variable-Class, finds the shortest cycle and counts it as a code smell instance [AFT19] [Man+16].
DataClass	x	x		Matches all classes where number of methods is 0 [Fow18].
DataClump-Arguments *DataClumps**	x	x		Matches methods with at least a high number of arguments with the same name and type [Zha+08]. Thresholds: *high_number_of_repeating_arguments* is 3 by def.
DataClumpFields *DataClumps**	x	x		Matches classes that have at least a high number of variables with the same name and type [Zha+08]. Thresholds: *high_number_of_repeating_variables* is 3 by def.
DistortedHierarchy		x		Matches classes that have an unusually deep inheritance tree. Finds classes with depth of inheritance larger than the short term memory cap [Man+16]. Thresholds: *short_term_memory_cap* is set to 7.
DivergentChange *SchizophrenicClass*	x	x		Matches methods that call a very high number of methods [Fow18][GSS13]. Thresholds: *very_high_number_of_direct_calls* determined using box-plot technique.
ExternalDuplicat-ionQuery *DuplicatedCode**	x	x		Matches classes that belong to different modules and that share duplicated code [Man+16].

Code smell name *alternative names (*split smells)*	Included in [Fow18]	[Man+16]	[Hab+17]	Code smell description
FeatureEnvy	x	x		Matches methods that access more variables outside of the class than inside of the class [LM07]. Thresholds: $few\_access\_to\_local\_variables$ commonly between 2 and 5, set to 2. $locality_fraction$ common threshold set to 0.33.
GodClass		x		Matches classes where class cohesion is tight, number of weighted methods is very high and access to foreign data is at least few [LM07]. Thresholds: $tight\_class\_cohesion$ is 0.33, $few\_access\_to\_foreign\_variables$ is 2 by definition, $high\_number\_of\_weighted\_methods$ is found using the box-plot technique.
IgnoringLow-MemoryWarning *ILMW*			x	Matches classes that are view controllers and do not override method called didReceiveMemoryWarning [Hab+17].
Inappropriate-Intimacy	x			Matches pairs of classes that have more method calls between them than a high number of calls between classes. [Pal+18] [Fow18]. Thresholds: $high\_number\_of\_calls\_between\_classes$ determined using box-plot technique.
IntensiveCoupling		x		Matches methods where number of method calls is larger than the short memory cap and coupling dispersion is lower than half or if the number of methods it calls is larger than few and coupling dispersion is smaller than few and the maximum nesting depth of the method is larger than shallow [LM07]. Thresholds: $short\_term\_memory\_gap$ is 7, $half\_coupling\_dispersion$ is 0.5, $quarter\_coupling\_dispersion$ is 0.25, $shallow\_number\_nesting\_depth$ is 1 and $few\_coupling\_intensity$ is 2.

Code smell name *alternative names (*split smells)*	Included in			Code smell description
	[Fow18]	[Man+16]	[Hab+17]	
InternalDuplication *DuplicatedCode**	x	x		Matches classes that belong to the same module and that share duplicated code [Man+16].
LazyClass	x			Matches classes with either no methods, or medium number of instructions and low class complexity method ratio or smaller than medium coupling between object classes and some depth of inheritance [Mun05]. Thresholds: $some\_depth\_of\_inheritance$ is set to 1 by definition. Other thresholds determined using the box-plot technique.
LongMethod *BlobOperation* *LM*	x	x	x	Matches matches all methods where the number of instructions is bigger than a very high number of instructions [Hec+15]. Thresholds: determined using the box-plot technique.
LongParameter-List	x			Matches methods that have a very long parameter list [Fow18]. Thresholds: determined using the box-plot technique.
MassiveView-Controller *MAVC*			x	Matches classes that are view controllers and where number of methods, number of attributes and number of instructions is very high [Hab+17]. Thresholds: determined using the box-plot technique.
MessageChain	x	x		Matches all methods, where the maximum number of chained message calls is larger than very high [BRH13]. Thresholds: determined using the box-plot technique.

Code smell name *alternative names (*split smells)*	Included in [Fow18]	[Man+16]	[Hab+17]	Code smell description
MiddleMan	x			Matches all classes where more than half of the methods are delegation methods. Delegation methods have at least one reference (uses/calls) to another class but have less than a small number of lines [Zha+08]. Thresholds: *delegation_to_all_methods_ratio* set to 0.5, *small_number_of_lines* determined using the box-plot technique.
MissingTemplate-Method		x		Matches methods that call the same methods and use the same variables [Fow18]. Thresholds: *minimal_common_method_and _variable_count* set to 5, *minimal_method_count* set to 2.
Parallel-Inheritance-Hierarchies	x			Matches parallel hierarchy trees for classes that start with the same prefixes [Fow18][GSS13]. Thresholds: *prefix_length* is 3, *minimum_number_of_classes_in _hierarchy* is 5.
PrimitiveObsession	x			Matches variables whose type is not a type defined in this application and that are used by multiple methods [GSM17]. Thresholds: *very_high_primitive_variable_use* is determined using the box-plot technique.
RefusedBequest *RefusedParent-*Bequest	x	x		Not implemented. Definition relies on protected members [LM07] and there are no protected variables/methods in Swift.
SAPBreaker *SAPBreakers**		x		Matches classes where class abstractness + instability is far from the 1-x mainline [Blo19][Man+16]. Thresholds: *allowed_distance_from_main* is set to 0.5.

Code smell name *alternative names (*split smells)*	Included in			Code smell description
	[Fow18]	[Man+16]	[Hab+17]	
SAPBreakerModule *SAPBreakers**		x		Matches modules where class abstractness + instability is far from the 1-x mainline [Blo19][Man+16]. Thresholds: $allowed\_distance\_from\_main$ is set to 0.5.
ShotgunSurgery	x	x		Matches all methods that are called by more than a $very\_high\_number\_of\_callers$ [Fow18][Man+16]. Thresholds: determined using the box-plot technique.
SiblingDuplication		x		Matches classes that have a common parent class (somewhere in the hierarchy) and that share duplicated code. [Man+16]
*DuplicatedCode**	x			
Speculative-GeneralityMethod *Speculative-Generality**	x			Matches methods that have unused parameters [Zha+08].
Speculative-GeneralityProtocol *Speculative-Generality**	x			Matches interfaces that are not implemented or extended [Zha+08].
SwissArmyKnife *SAK*			x	Matches classes that are interfaces and have a very high $number\_of\_methods$ [Hec+15]. Thresholds: determined using the box-plot technique.
SwitchStatements	x			Matches all methods, where the number of switch statements is higher than $very\_high\_number\_of\_switch\_statements$ [Fow18]. Thresholds: determined using the box-plot technique.

Code smell name *alternative names (*split smells)*	Included in [Fow18]	[Man+16]	[Hab+17]	Code smell description
TraditionBreaker		x		Matches classes that do not have any sub-classes, where *number_of_methods_and_attributes* is low and where they inherit from a class whose *number_of_methods_and_attributes* is very high [Moh+09][Tea14]. Thresholds: determined using the box-plot technique.

A.3. Code Smell Queries for GraphifySwift

A.3.1. Long method

```
MATCH (c:Class)-[r:CLASS_OWNS_METHOD]->(m:Method)
WHERE m.number_of_instructions > veryHighNumberOfInstructions
RETURN
    distinct(m.app_key) as app_key,
   count(distinct m) as number_of_smells
```

A.3.2. Large class / Blob class

```
MATCH (cl:Class)
WHERE
    cl.lack_of_cohesion_in_methods
            > veryHighLackOfCohesionInMethods AND
    cl.number_of_methods >  veryHighNumberOfMethods AND
    cl.number_of_attributes > veryHighNumberOfAttributes
RETURN
    distinct cl.app_key as app_key,
    count(distinct cl) as number_of_smells
```

A.3.3. Shotgun surgery

```
MATCH (other_m:Method)-[r:CALLS]->(m:Method)
                <-[:CLASS_OWNS_METHOD]-(c:Class)
WITH
    c,
    m,
    COUNT(r) as number_of_callers
WHERE number_of_callers > veryHighNumberOfCallers
RETURN
```

```
    distinct(m.app_key) as app_key,
    count(distinct m) as number_of_smells
```

A.3.4. Switch statements

```
MATCH (c:Class)-[r:CLASS_OWNS_METHOD]->(m:Method)
WHERE m.number_of_switch_statements >
                veryHighNumberOfSwitchStatements
RETURN
    distinct(m.app_key) as app_key,
    count(distinct m) as number_of_smells
```

A.3.5. Lazy class

```
MATCH (c:Class)
WHERE
    c.number_of_methods = 0 OR
    (c.number_of_instructions < mediumNumberOfInstructions AND
     c.class_complexity/c.number_of_methods <=
                           lowComplexityMethodRatio) OR
     (c.coupling_between_object_classes <
                        mediumCouplingBetweenObjectClasses AND
     c.depth_of_inheritance > numberOfSomeDepthOfInheritance)
RETURN
    distinct(c.app_key) as app_key,
    count(distinct c) as number_of_smells
```

A.3.6. Message chains

```
MATCH (c:Class)-[CLASS_OWNS_METHOD]-(m:Method)
WHERE m.max_number_of_chaned_message_calls >
                        veryHighNumberOfChainedMessages
RETURN
    distinct(m.app_key) as app_key,
    count(distinct m) as number_of_smells
```

A.3.7. Data class

```
MATCH (c:Class)
WHERE c.number_of_methods = 0
RETURN
    distinct(c.app_key) as app_key,
    count(distinct c) as number_of_smells
```

A.3.8. Refused bequest / refused parent bequest

Not applicable for swift since swift does not have the keyword protected.

A.3.9. Comments

```
MATCH
   (c:Class) where c.number_of_comments >
                        veryHighNumberOfComments
RETURN
   distinct(c.app_key) as app_key,
   count(distinct c) as number_of_smells
```

A.3.10. Cyclic dependencies (dependencies between classes)

```
MATCH
    (c:Class)-[:CLASS_OWNS_VARIABLE]->(v:Variable)
                            -[:IS_OF_TYPE]->(c2:Class)
WHERE
    c <> c2
MATCH
    cyclePath=shortestPath((c2)-
            [:CLASS_OWNS_VARIABLE|:IS_OF_TYPE*]->(c))
WITH
    c,
    v,
    [n in nodes(cyclePath) | n.name ] as names
RETURN
    distinct(c.app_key) as app_key,
    count(distinct c) as number_of_smells
```

A.3.11. Cyclic dependencies (dependencies between modules)

Normally when discussing cyclic dependencies the dependencies between modules are meant. Dependencies between modules are not possible in swift as the build of such projects fails.

A.3.12. Intensive coupling

```
MATCH
    (c:Class)-[r:CLASS_OWNS_METHOD]->(m1:Method)
                            -[s:CALLS]->(m2:Method),
    (c2:Class)-[r2:CLASS_OWNS_METHOD]->(m2)
WHERE id(c) <> id(c2)
WITH
    c,
    m1,
    count(distinct m2) as method_count,
    collect(distinct m2.name) as names,
    collect(distinct c2.name) as class_names,
    count(distinct c2) as class_count
WHERE
    ((method_count >= maxNumberOfShortMemoryCap and
     class_count/method_count <= halfCouplingDispersion) or
    (method_count >= fewCouplingIntensity
     and class_count/method_count <=
                    quarterCouplingDispersion))
    and m1.max_nesting_depth >= shallowMaximumNestingDepth
RETURN
    distinct(m1.app_key) as app_key,
    count(m1) as number_of_smells
```

A.3.13. Distorted hierarchy

```
MATCH (c:Class)
WHERE c.depth_of_inheritance > shortTermMemoryCap
RETURN
    distinct(c.app_key) as app_key,
    count(distinct c) as number_of_smells
```

A.3.14. Tradition Breaker

```
MATCH (c:Class)-[r:EXTENDS]->(parent:Class)
WHERE
    NOT ()-[:EXTENDS]->(c) AND
    c.number_of_methods + c.number_of_attributes <
                    lowNumberOfmethodsAndAttributes AND
    parent.number_of_methods + parent.number_of_attributes >=
                    veryHighNumberOfMethodsAndAttributes
RETURN
    distinct(c.app_key) as app_key,
    count(distinct c) as number_of_smells
```

A.3.15. Sibling Duplication

```
MATCH
    (firstClass:Class)-[:EXTENDS*]-> (parent:Class)
            <-[:EXTENDS*]-(secondClass:Class),
    (firstClass:Class)-[:DUPLICATES]->(secondClass:Class)
WHERE
    firstClass.data_string contains d.fragment or
    secondClass.data_string contains d.fragment
RETURN
    distinct(firstClass.app_key) as app_key,
    count(distinct d) as number_of_smells
```

A.3.16. Internal Duplication

```
MATCH
    (firstClass:Class)-[r:DUPLICATES]->(secondClass:Class),
    (module:Module)-[:MODULE_OWNS_CLASS]->(firstClass),
    (module:Module)-[:MODULE_OWNS_CLASS]->(secondClass)
WHERE
   firstClass.data_string contains r.fragment or
   secondClass.data_string contains r.fragment
RETURN
   distinct(firstClass.app_key) as app_key,
   count(distinct r) as number_of_smells
```

A.3.17. External Duplication

```
MATCH
    (firstClass:Class)-[:DUPLICATES]->(secondClass:Class),
    (module:Module)-[:MODULE_OWNS_CLASS]->(firstClass),
    (secondModule:Module)-[:MODULE_OWNS_CLASS]->(secondClass)
WHERE
    id(module) <> id(secondModule) and
    firstClass.data_string contains d.fragment or
    secondClass.data_string contains d.fragment
RETURN
    distinct(firstClass.app_key) as app_key,
    count(distinct d) as number_of_smells
```

A.3.18. Divergent Change/Schizophrenic Class

```
MATCH
    (c:Class)-[:CLASS_OWNS_METHOD]->(m:Method)-[r:CALLS]
                                ->(other_method:Method)
WITH
    c,
    m,
    COUNT(r) as number_of_called_methods
WHERE
    number_of_called_methods > veryHighNumberOfCalledMethods
RETURN
    distinct(m.app_key) as app_key,
    count(distinct m) as number_of_smells
```

A.3.19. Long parameter list

```
MATCH
    (c:Class)-[:CLASS_OWNS_METHOD]->(m:Method)-
```

```
            [r:METHOD_OWNS_ARGUMENT]->(a:Argument)
WITH
    c,
    m,
    count(a) as argument_count
WHERE argument_count > veryHighNumberOfParameters
RETURN
    distinct(m.app_key) as app_key,
    count(distinct m) as number_of_smells
```

A.3.20. Feature envy

```
MATCH
    (class:Class)-[:CLASS_OWNS_METHOD]->(m:Method)-[:USES]->
    (v:Variable)<-[:CLASS_OWNS_VARIABLE]-(other_class:Class)
 WHERE
  class <> other_class
WITH
    class,
    m,
    count(distinct v) as variable_count,
    collect(distinct v.name) as names,
    collect(distinct other_class.name) as class_names,
    count(distinct other_class) as class_count
MATCH
    (class)-[:CLASS_OWNS_METHOD]->(m)-[:USES]->
        (v:Variable)<-[:CLASS_OWNS_VARIABLE]-(class)
WITH
   class,
   m,
   variable_count,
   class_names,
   names,
    count(distinct v) as local_variable_count,
    collect(distinct v.name) as local_names,
  class_count
WHERE
    local_variable_count + variable_count > 0
WITH
class,
  m,
   variable_count,
    class_names,
   names,
    local_variable_count,
     local_names,
  class_count,
    local_variable_count*
        1.0/(local_variable_count+variable_count) as locality
WHERE
    variable_count > fewAccessToForeignVariables and
    locality < localityFraction and
   class_count <= fewAccessToForeignClasses
RETURN
    distinct(class.app_key) as app_key,
    count(distinct m) as number_of_smells
```

A.3.21. Data clumps (class variables)

```
MATCH
   (app:App)-[:APP_OWNS_MODULE]->(module:Module)-
        [:MODULE_OWNS_CLASS]->(class:Class)-
            [:CLASS_OWNS_VARIABLE]->(variable:Variable)
MATCH
   (app)-[:APP_OWNS_MODULE]->(module:Module)-
        [:MODULE_OWNS_CLASS]->(other_class:Class)-
            [:CLASS_OWNS_VARIABLE]->(other_variable:Variable)
WHERE
  class <> other_class and
  variable.type = other_variable.type and
  variable.name = other_variable.name
WITH
   app,
   class,
   other_class,
   variable
   order by variable.name
WITH
   app,
   class,
   other_class,
   collect(distinct variable.name) as variable_names,
   count(DISTINCT variable) as variable_count
WITH
    app,
    class,
    other_class,
    variable_names,
    variable_count
    order by id(class)
WITH
    app,
    collect(distinct class.name) as class_names,
    variable_names,
    variable_count
WHERE
    variable_count >= highNumberOfRepeatingVariables
RETURN
   distinct(app.app_key) as app_key,
   count(distinct class_names) as number_of_smells
```

A.3.22. Data clumps (function arguments)

```
MATCH
    (app:App)-[:APP_OWNS_MODULE]->(module:Module)-
    [:MODULE_OWNS_CLASS]->(class:Class)-[:CLASS_OWNS_METHOD]
    ->(method:Method)-[:METHOD_OWNS_ARGUMENT]
        ->(argument:Argument)
MATCH
    (app)-[:APP_OWNS_MODULE]->(module:Module)-
    [:MODULE_OWNS_CLASS]>(other_class:Class)-
    [:CLASS_OWNS_METHOD]->(other_method:Method)-
    [:METHOD_OWNS_ARGUMENT]->(other_argument:Argument)
WHERE
   method <> other_method and
   argument.name = other_argument.name and
   argument.type = other_argument.type
WITH
    app,
    class,
    other_class,
    method,
    other_method,
    argument
    order by other_method.name
 WITH
    app,
    class,
    other_class,
    method,
    other_method,
    argument
    order by argument.name
WITH
   collect(argument.name) as argument_names,
   count(argument.name) as argument_count,
   method,
   other_method,
   app,
   class
WHERE
    argument_count >= highNumberOfRepeatingArguments
WITH
    collect(other_method.name) + method.name as method_names,
    collect(id(other_method)) + id(method) as method_ids,
    count(distinct other_method) as method_count,
    method,
    app,
    argument_names,
    argument_count,
    class
WITH
   collect(class.name) as class_names,
   method_names,
   app,
   argument_names,
   argument_count,
   method_ids,
   method_count
MATCH
     (app)-[:APP_OWNS_MODULE]->(:Module)-[:MODULE_OWNS_CLASS]
        ->(class:Class)-[:CLASS_OWNS_METHOD]->(method:Method)
        -[:METHOD_OWNS_ARGUMENT]->(argument:Argument)
WHERE
    id(method) in method_ids and
    argument.name in argument_names
WITH
    argument,
    app,
    method,
    argument_names,
    argument_count,
    class order by argument.name
WITH
    collect(distinct argument.name) as new_argument_names,
    app,
    method,
    argument_names,
    argument_count,
    class
WITH
  collect(method.name) as new_method_names,
  collect(class.name) as class_names,
  new_argument_names,
  app,
  argument_names,
  argument_count
RETURN
    distinct(app.app_key) as app_key,
    count(distinct class_names) as number_of_smells
```

A.3.23. Speculative generality (interfaces)

```
MATCH
    (class:Class)
WHERE NOT
    ()-[:IMPLEMENTS|EXTENDS]->(class) and
    class.is_interface = true
RETURN
    distinct(class.app_key) as app_key,
    count(distinct class) as number_of_smells
```

A.3.24. Speculative generality (methods)

```
MATCH
    (class)-[:CLASS_OWNS_METHOD]->(m:Method)-
    [:METHOD_OWNS_ARGUMENT]->(p:Argument) -[:IS_OF_TYPE]
        ->(other_class:Class)
WHERE
    NOT (m)-[:CALLS|USES]->()
     <-[:CLASS_OWNS_VARIABLE|CLASS_OWNS_METHOD]-
            (other_class) AND
    NOT m.data_string contains (\"=\" + p.name) AND
    NOT m.data_string contains (\"= \" + p.name) AND
    NOT m.data_string contains (\":\" + p.name) AND
    NOT m.data_string contains (\": \" + p.name) AND
    NOT m.data_string contains (\"(\" + p.name + \")\") and
    NOT m.data_string contains (\"(\" + p.name + \",\") and
    NOT m.data_string contains (\"(\" + p.name + \" ,\") and
    NOT m.data_string contains (\", \" + p.name + \")\") and
    NOT m.data_string contains (\", \" + p.name + \",\") and
    class.is_interface = false
RETURN
    distinct(class.app_key) as app_key,
    count(distinct m) as number_of_smells
```

A.3.25. Middle man

```
MATCH
    (class:Class)-[:CLASS_OWNS_METHOD]->(method:Method)-
    [:USES|CALLS]->(ref)
        <-[:CLASS_OWNS_VARIABLE|CLASS_OWNS_METHOD]-
            (other_class:Class)
WHERE
    class <> other_class and
    method.number_of_instructions <
        lowNumberOfInstructionsMethod
WITH
    class,
    method,
    collect(ref.name) as referenced_names,
    collect(other_class.name) as class_names
WITH
    collect(method.name) as method_names,
    collect(referenced_names) as references,
    collect(class_names) as classes,
    collect(method.number_of_instructions) as
    numbers_of_instructions,
    class,
    count(method) as method_count,
    count(method)*1.0/class.number_of_methods as method_ratio
WHERE
    method_ratio > delegationToAllMethodsRatioHalf
RETURN
    distinct(class.app_key) as app_key,
    count(class) as number_of_smells
```

A.3.26. Parallel inheritance hierarchies

```
MATCH
```

```
    (parent:Class)<-[:MODULE_OWNS_CLASS]-(:Module)
        <-[:APP_OWNS_MODULE]-(app:App)
MATCH
   (app)-[:APP_OWNS_MODULE]->(:Module)-[:MODULE_OWNS_CLASS]
        ->(other_parent:Class)
WHERE
   parent <> other_parent
MATCH
    path = (class:Class)-[:EXTENDS*]->(parent)
MATCH
   other_path = (other_class:Class)-[:EXTENDS*]
        ->(other_parent)
WHERE
    length(path) = length(other_path) and
    length(path) > 0 and
    class.name starts with
        substring(other_class.name, 0, \(self.prefixLength))
    and parent.name starts with
        substring(other_parent.name, 0, \(self.prefixLength))
WITH
  collect(distinct [n in nodes(path) | n.name ]) as first,
  collect(distinct [n in nodes(other_path) | n.name]) as second,
  parent,
  other_parent
WITH
  REDUCE(output = [], r IN first | output + r) as first_names,
  REDUCE(output = [], r IN second | output + r) AS second_names,
  parent,
  other_parent
UNWIND
  first_names as first_name
WITH
   collect(distinct first_name) as first_names,
   second_names,
   parent,
   other_parent
UNWIND
    second_names as second_name
WITH
    collect(distinct second_name) as second_names,
    first_names,
    parent,
    other_parent
```

```
WHERE
    size(first_names) >= minimumNumberOfClassesInHierarcy and
    size(second_names) >= minimumNumberOfClassesInHierarcy
RETURN
   distinct(parent.app_key) as app_key,
   count(parent)/2 as number_of_smells
```

A.3.27. Inappropriate intimacy

```
MATCH
    (app:App)-[:APP_OWNS_MODULE]->(module:Module)
        -[:MODULE_OWNS_CLASS]->(class:Class)-
        [:CLASS_OWNS_METHOD]->(method:Method)
MATCH
   (app:App)-[:APP_OWNS_MODULE]->(other_module:Module)-
    [:MODULE_OWNS_CLASS]->other_class:Class)-
        [:CLASS_OWNS_METHOD]->(other_method:Method)
WHERE
    class <> other_class
MATCH
    path = (method)-[r:CALLS]-(other_method)
WITH
    count(distinct r) as number_of_calls,
    collect(distinct method.name) as method_names,
    collect(distinct other_method.name) as other_method_names,
    class,
    other_class
WHERE
   number_of_calls > highNumberOfCallsBetweenClasses
RETURN
   distinct(class.app_key) as app_key,
   count(class)/2 as number_of_smells
```

A.3.28. Brain method

```
MATCH
    (class:Class)-[:CLASS_OWNS_METHOD]->(method:Method)
WHERE
   class.number_of_instructions >
        highNumberOfInstructionsForClass and
   method.cyclomatic_complexity >=
            highCyclomaticComplexity and
   method.max_nesting_depth >= severalMaximalNestingDepth
MATCH
```

```
  (method)-[:USES]->(variable:Variable)
WITH
    class,
    method,
    count(distinct variable) as number_of_variables,
    collect(distinct variable.name) as variable_names
WHERE
    number_of_variables > manyAccessedVariables
RETURN
   distinct(class.app_key) as app_key,
   count(distinct method) as number_of_smells
```

A.3.29. God class

```
MATCH
    (class:Class)-[:CLASS_OWNS_METHOD]->(method:Method)
MATCH
    (class)-[:CLASS_OWNS_METHOD]->(other_method:Method)
WHERE
    method <> other_method
WITH
    count(DISTINCT [method, other_method]) as pair_count,
    class
MATCH
    (class)-[:CLASS_OWNS_METHOD]->(method:Method)
MATCH
    (class)-[:CLASS_OWNS_METHOD]->(other_method:Method)
MATCH
    (class)-[:CLASS_OWNS_VARIABLE]->(variable:Variable)
WHERE
    method <> other_method and
    (method)-[:USES]->(variable)<-[:USES]-(other_method)
WITH
    class,
    pair_count,
    method,
    other_method,
    collect(distinct variable.name) as variable_names,
    count(distinct variable) as variable_count
WHERE
    variable_count >= 1
WITH
    class,
    pair_count,
    count(distinct [method, other_method])
                    as connected_method_count
WITH
    class,
    connected_method_count*0.1/pair_count as class_cohesion,
    connected_method_count,
    pair_count
WHERE
    class_cohesion < tightClassCohesionFraction and
    class.class_complexity >= veryHighWeightedMethodCount
OPTIONAL MATCH
    (class)-[:CLASS_OWNS_METHOD]->(m:Method)-[:USES]
    ->(variable:Variable)<-[:CLASS_OWNS_VARIABLE]-
            (other_class:Class)
WHERE
    class <> other_class
WITH
    class,
    class_cohesion,
    connected_method_count,
    pair_count,
    count(distinct variable) as foreign_variable_count
WHERE
    foreign_variable_count >= fewAccessToForeignData
RETURN
    distinct(class.app_key) as app_key,
    count(distinct class) as number_of_smells
```

A.3.30. SAP Breaker

```
MATCH
    (app:App)-[:APP_OWNS_MODULE]->(module:Module)-
        [:MODULE_OWNS_CLASS]->(class:Class)
MATCH
(app:App)-[:APP_OWNS_MODULE]->(other_module:Module)-
        [:MODULE_OWNS_CLASS]->(other_class:Class)
WHERE
(other_class)-[:CLASS_OWNS_METHOD]->()-[:USES|:CALLS]->()
        <-[:CLASS_OWNS_METHOD|:CLASS_OWNS_VARIABLE]-(class) and
class <> other_class
WITH
  count(distinct other_class) as number_of_dependant_classes,
  class,
  app
WITH
    class,
    number_of_dependant_classes as efferent_coupling_number,
    app
MATCH
  (app)-[:APP_OWNS_MODULE]->(module:Module)-
        [:MODULE_OWNS_CLASS]->(other_class:Class)
WHERE
   (class)-[:CLASS_OWNS_METHOD]->()-[:USES|:CALLS]->()
    <-[:CLASS_OWNS_METHOD|:CLASS_OWNS_VARIABLE]-(other_class)
    and class <> other_class
WITH
    count(distinct other_class) as afferent_coupling_number,
    class,
    efferent_coupling_number
WITH
    efferent_coupling_number*1.0/(efferent_coupling_number +
                afferent_coupling_number) as instability_number,
    class,
    afferent_coupling_number,
    efferent_coupling_number
OPTIONAL MATCH
    (class)-[:CLASS_OWNS_METHOD]->(method:Method)
WHERE
  method.is_abstract
WITH
    count(distinct method)/class.number_of_methods
        as abstractness_number,
    instability_number,
    afferent_coupling_number,
    efferent_coupling_number,
    class
WITH
   1 - (abstractness_number + instability_number)^2
        as difference_from_main,
   instability_number,
   abstractness_number,
   class
WHERE
    difference_from_main < - allowedDistanceFromMain or
    difference_from_main > allowedDistanceFromMain
```

```
RETURN
    distinct(class.app_key) as app_key,
    count(distinct class) as number_of_smells
```

A.3.31. SAP Breaker for Modules

```
MATCH
    (app:App)-[:APP_OWNS_MODULE]->(module:Module)-
        [:MODULE_OWNS_CLASS]->(class:Class)
MATCH
    (app:App)-[:APP_OWNS_MODULE]->(other_module:Module)-
        [:MODULE_OWNS_CLASS]->(other_class:Class)
WHERE
    (other_class)-[:CLASS_OWNS_METHOD]->()-[:USES|:CALLS]->()
    <-[:CLASS_OWNS_METHOD|:CLASS_OWNS_VARIABLE]-(class) and
    module <> other_module
WITH
    count(distinct other_class) as number_of_dependant_classes,
    module
WITH
    module,
    number_of_dependant_classes as efferent_coupling_number

MATCH
    (module:Module)-[:MODULE_OWNS_CLASS]->(class:Class)
MATCH
    (other_module:Module)-[:MODULE_OWNS_CLASS]
            ->(other_class:Class)
WHERE
    (class)-[:CLASS_OWNS_METHOD]->()-[:USES|:CALLS]->()
    <-[:CLASS_OWNS_METHOD|:CLASS_OWNS_VARIABLE]-(other_class)
    and module <> other_module
WITH
   count(distinct other_class) as afferent_coupling_number,
   module,
   efferent_coupling_number
WITH
    efferent_coupling_number*1.0/(efferent_coupling_number +
        afferent_coupling_number) as instability_number,
    afferent_coupling_number,
    efferent_coupling_number,
    module
OPTIONAL MATCH
```

```
  (module)-[:MODULE_OWNS_CLASS]->(class:Class)
WHERE
    class.is_interface
WITH
    count(distinct class)/module.number_of_classes
                                  as abstractness_number,
    instability_number,
    afferent_coupling_number,
    efferent_coupling_number,
    module
WITH
    1 - (abstractness_number + instability_number)^2
        as difference_from_main,
    instability_number,
    abstractness_number,
    module
WHERE
    difference_from_main < - allowedDistanceFromMain or
    difference_from_main > allowedDistanceFromMain
RETURN
    distinct(module.app_key) as app_key,
    count(distinct module) as number_of_smells
```

A.3.32. Unstable dependencies

```
MATCH
    (class:Class)
MATCH
    (other_class:Class)
WHERE
    (other_class)-[:CLASS_OWNS_METHOD]->()-[:USES|:CALLS]->()
    <-[:CLASS_OWNS_METHOD|:CLASS_OWNS_VARIABLE]-(class)
    and class <> other_class
WITH
    count(distinct other_class) as number_of_dependant_classes,
    class
WITH
    class,
    number_of_dependant_classes as efferent_coupling_number

MATCH
    (class:Class)
MATCH
```

```
    (other_class:Class)
WHERE
    (class)-[:CLASS_OWNS_METHOD]->()-[:USES|:CALLS]->()
    <-[:CLASS_OWNS_METHOD|:CLASS_OWNS_VARIABLE]-(other_class)
    and class <> other_class
WITH
    count(distinct other_class) as afferent_coupling_number,
    class,
    efferent_coupling_number
WITH
    efferent_coupling_number*1.0/(efferent_coupling_number
        + afferent_coupling_number) as instability_number,
    class,
    afferent_coupling_number,
    efferent_coupling_number

MATCH
    (comparison_class:Class)
WHERE
    (comparison_class)-[:CLASS_OWNS_METHOD]->(:Method)-
    [:USES|:CALLS]->()
    <-[:CLASS_OWNS_METHOD|:CLASS_OWNS_VARIABLE]-(class)
    and comparison_class <> class

MATCH
    (other_class:Class)
WHERE
    (other_class)-[:CLASS_OWNS_METHOD]->()-[:USES|:CALLS]->()
    <-[:CLASS_OWNS_METHOD|:CLASS_OWNS_VARIABLE]-
        (comparison_class)
    and comparison_class <> other_class
WITH
    count(distinct other_class) as number_of_dependant_classes2,
    comparison_class,
    class,
    instability_number
WITH
    comparison_class,
    number_of_dependant_classes2 as efferent_coupling_number2,
    class,
    instability_number

MATCH
```

```
    (comparison_class:Class)
MATCH
    (other_class:Class)
WHERE
    (comparison_class)-[:CLASS_OWNS_METHOD]->()-[:USES|:CALLS]
    ->()<-[:CLASS_OWNS_METHOD|:CLASS_OWNS_VARIABLE]-
        (other_class)
    and comparison_class <> other_class
WITH
    count(distinct other_class) as afferent_coupling_number2,
    comparison_class,
    efferent_coupling_number2,
    class,
    instability_number
WITH
   efferent_coupling_number2*1.0/(efferent_coupling_number2
         + afferent_coupling_number2) as
   instability_number2,
   comparison_class,
   afferent_coupling_number2,
   efferent_coupling_number2,
   class,
   instability_number

WHERE
    instability_number2 < instability_number
RETURN
    comparison_class.app_key as app_key,
    count(distinct comparison_class) as number_of_smells
```

A.3.33. Primitive obsession

```
MATCH
   (class:Class)-[:CLASS_OWNS_VARIABLE]->(variable:Variable)
        <-[use:USES]-(method:Method)
WHERE
  not (variable)-[:IS_OF_TYPE]->()
WITH
   collect(distinct method.name) as uses,
   count(distinct use) as use_count,
   variable,
   class
WHERE
```

```
  use_count > veryHighPrimitiveVariableUse
RETURN
    distinct(class.app_key) as app_key,
    count(distinct variable) as number_of_smells
```

A.3.34. Alternative classes with different interfaces

```
MATCH
    (app:App)-[:APP_OWNS_MODULE]->(module:Module)-
    [:MODULE_OWNS_CLASS]->(class:Class)-[:CLASS_OWNS_METHOD]
    ->(method:Method)-[:METHOD_OWNS_ARGUMENT]
    ->(argument:Argument)
MATCH
    (app:App)-[:APP_OWNS_MODULE]->(module:Module)-
    [:MODULE_OWNS_CLASS]->(other_class:Class)-
    [:CLASS_OWNS_METHOD]->(other_method:Method)-
    [:METHOD_OWNS_ARGUMENT]->(other_argument:Argument)
WHERE
    not (class)-[:IMPLEMENTS|:EXTENDS]->()
    <-[:IMPLEMENTS|:EXTENDS]-(other_class) and
    not (class)-[:IMPLEMENTS|:EXTENDS]-(other_class) and
    class.app_key = other_class.app_key and
    class <> other_class and
    method.number_of_parameters =
        other_method.number_of_parameters and
    method.number_of_parameters >= minimumNumberOfParameters
    and argument.type = other_argument.type and
    method.return_type = other_method.return_type
WITH
    class,
    other_class,
    method,
    other_method,
    argument
    order by argument.type
WITH
    collect(distinct argument) as arguments,
    count(distinct argument) as number_of_arguments,
    method,
    other_method,
    class,
    other_class
WHERE
```

```
  number_of_arguments = method.number_of_parameters
WITH
  [argument in arguments | argument.type] as arguments,
  number_of_arguments,
  method,
  other_method,
  class,
  other_class
  order by method.name
WITH
   collect(distinct class.name +"."+method.name) as method_names,
   count(distinct method)
        as method_count,
   class,
   other_class,
   collect(number_of_arguments) as number_of_arguments,
   collect(distinct arguments) as types
WHERE
    method_count >= \(minimumCommonMethodCount)
WITH
   collect(distinct method_names) as method_names,
   collect(distinct class.name) as class_names,
   types,
   class.app_key as app_key,
   count(distinct class.name) as class_count
WHERE
  class_count >= 2
RETURN
    distinct app_key,
    count(distinct class_names) as number_of_smells
```

A.3.35. Missing template method

```
MATCH
    (app:App)-[:APP_OWNS_MODULE]->(module:Module)-
    [:MODULE_OWNS_CLASS]->(class:Class)-[:CLASS_OWNS_METHOD]
    ->(method:Method)-[:USES|:CALLS]->(common)<-[:USES|:CALLS]-
    (other_method)<-[:CLASS_OWNS_METHOD]-(other_class:Class)
WHERE
    method <> other_method
WITH
    collect(distinct common) as commons,
    count(distinct common) as common_count,
    class, other_class, method, other_method
WHERE
     common_count >= minimalCommonMethodAndVariableCount
WITH
    [common in commons | class.name+"."+common.name]
        as common_names,
   class,
   other_class,
   method,
   other_method,
   common_count
WITH
    collect(class.name) as class_names,
    collect(class.name + "." + method.name) as method_names,
    count(distinct method) as method_count,
    class.app_key as app_key,
    common_names, common_count
WHERE
    method_count >= \(self.minimalMethodCount)
RETURN
    distinct(app_key),
    count(distinct common_names) as number_of_smells
```

A.3.36. Swiss army knife

```
MATCH
    (cl:Class)
WHERE
    cl.is_interface AND
    cl.number_of_methods > veryHighNumberOfMethods
RETURN
    distinct(cl.app_key) as app_key,
    count(distinct cl) as number_of_smells
```

A.3.37. Complex class

```
MATCH
    (cl:Class) WHERE cl.class_complexity >
        veryHighClassComplexity
RETURN
    distinct(cl.app_key) as app_key,
    count(distinct cl) as number_of_smells
```

A.3.38. Ignoring low memory warning

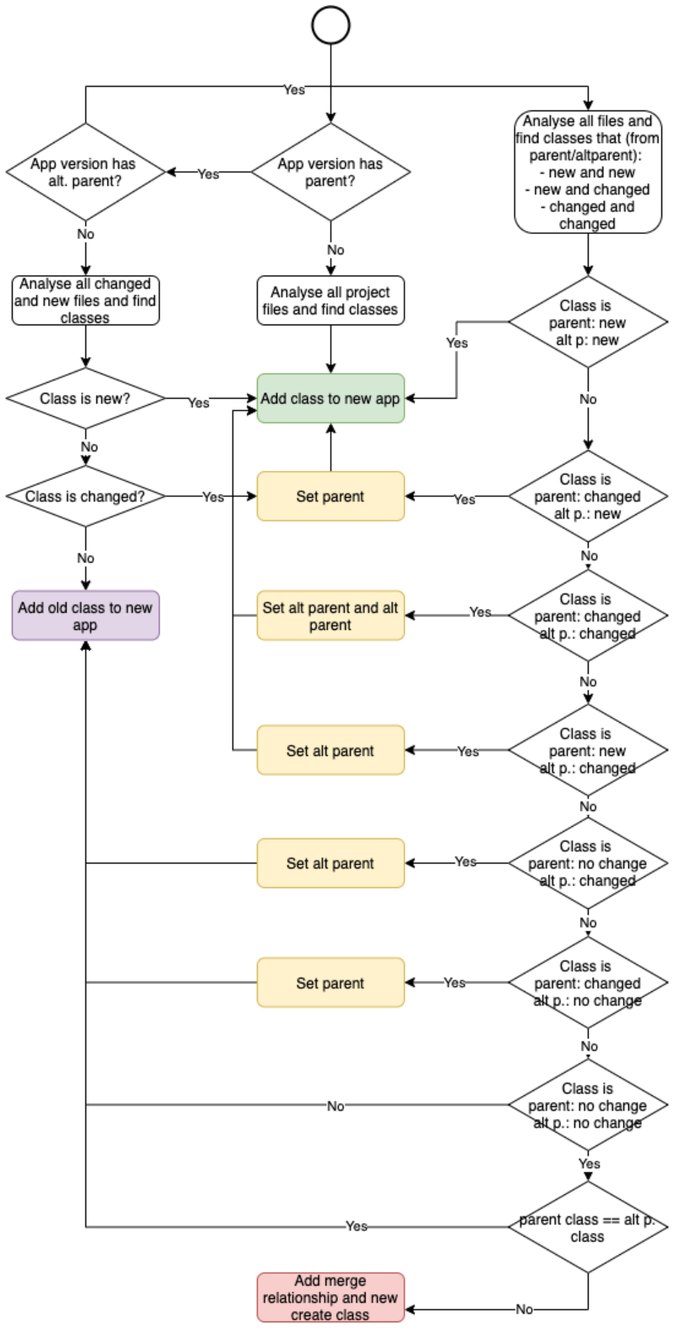
```
MATCH
    (class:Class)
WHERE
    class.name contains ’ViewController’ and
    not (class)-[:CLASS_OWNS_METHOD]
            ->(:Method{name:’didReceiveMemoryWarning()’})
RETURN
    distinct(class.app_key) as app_key,
    count(distinct class) as number_of_smells
```

A.3.39. Massive view controller

```
MATCH
    (class:Class)
WHERE
    class.name contains ’ViewController’ and
    class.number_of_methods > veryHighNumberOfMethods and
    class.number_of_attributes
            > veryHighNumberOfAttributes and
    class.number_of_instructions >
            veryHighNumberOfInstructionsClass
RETURN
    distinct(class.app_key) as app_key,
    count(distinct class) as number_of_smells
```

Appendix B. CODE ANALYSIS

B.1. GraphifyEvolution Evolution Decision Chart



App version has parent?
Yes
App version has alt. parent?
Yes
Analyse all files and find classes that (from parent/altparent):
- new and new
- new and changed
- changed and changed
No
Analyse all changed and new files and find classes
No
Analyse all project files and find classes
Class is new?
Yes
Add class to new app
No
Class is changed?
Yes
Set parent
No
Add old class to new app
Class is parent: new alt p: new
Yes
No
Class is parent: changed alt p.: new
Yes
No
Set alt parent and alt parent
Class is parent: changed alt p.: changed
Yes
No
Set alt parent
Class is parent: new alt p.: changed
Yes
No
Set alt parent
Class is parent: no change alt p.: changed
Yes
No
Set parent
Class is parent: changed alt p.: no change
Yes
No
Class is parent: no change alt p.: no change
No
Yes
parent class == alt p. class
Yes
No
Add merge relationship and new create class

Figure 42. Decision Flowchart for GraphifyEvolution

B.2. GraphifyEvolution Database Structure

B.2.1. Nodes and properties

Variable.

- *app_key* - unique key to identify application, will be added
- *code* - snippet of code where given variable is defined, currently disabled, will be made optional
- *kind* - defines if variables is instance, class or static variable
- *modifier* - private/public/internal/fileprivate/open - will be added
- *name* - name of variable
- *type* - type of variable, f.ex. String, [Int]?, App
- *usr* - unique identifier of variable inside app (provided by SourceKit)
- *start_line* - starting line of variable declaration
- *end_line* - ending line of variable declaration
- *version_number* - version number of variable, showing how many times the variable was changed

Method.

- *app_key* - unique key to identify application, will be added
- *cyclomatic_complexity* - cyclomatic complexity of method
- *code* - snippet of code where given method is defined, currently disabled, will be made optional
- *is_getter* - defines if method is a getter method, (not yet correctly implemented)
- *is_setter* - defines if method is a setter method (not yet correctly implemented)
- *is_constructor* - defines if method is a constructor
- *kind* - defines if method is instance, class or static method
- *max_nesting_depth* - maximal nesting depth of if/else/while/for/etc in a method – will be added
- *max_number_of_chaned_message_calls* - maximal number of chained message calls, for example: $test.values().findFirst().doSomething()$ is 3 chained message calls (not yet implemented)
- *modifier* - modifier of method one of the following private/public/internal/-fileprivate/open
- *name* - name of method
- *number_of_callers* - number of methods that call this method
- *number_of_declared_locals* - will be added
- *number_of_called_methods* - number of methods called from this method
- *number_of_instructions* - number of instructions in method

- $number_{o}f_{a}ccessed_{v}ariables$ - number of variables used by this method
- *number_of_parameters* - will be added
- *number_of_switch_statements* - will be added
- *type* - return type of function
- *usr* - unique identifier of method inside this application
- *start_line* - starting line of method declaration
- *end_line* - ending line of method declaration
- *version_number* - version number of method, showing how many times the variable was changed

App.

- *app_key* - unique key to identify application, optional
- *category* - app category, information taken from .json file for bulk analysis - will be added
- *developer* - app developer, information taken from .json file for bulk analysis - will be added
- *in_app_store* - specifies if app is in the app store, information taken from .json file for bulk analysis – will be added
- *language* - language of the application code - will be added
- *name* - name of application, information taken from .json file for bulk analysis
- *platform* - platform of app, currently for all swift apps set as "iOS" – will be added
- *star* - number of app repository stars, information taken from .json file for bulk analysis – will be added
- *version_number* - version number of method, showing how many times the variable was changed
- *commit* - commit hash
- *tree* - commit tree from git log
- *branch* - branch name, calculated by finding the following merge commit and extracting branch name from commit description. Sometimes incorrect, but currently only way to include names of deleted branches
- *tag* - tag of commit if it exists
- *time* - time of commit
- *author* - author of commit
- *message* - commit message
- *parent_commit* - parent commit
- *alternative_parent_commit* - commit of parent that was merged

Class.

- *app_key* - unique key to identify application – will be added
- *code* - source code of class
- *is_interface* - specifies if class is a protocol (or interface in Java) – will be added
- *name* - name of class
- *parent_name* - name of parent class - will be added
- *usr* - unique identifier inside application
- *kind* - class kind
- *path* - file path where class is located
- *version_number* - version number of class

Added through metrics queries.

- *number_of_attributes* - number of attributes in class
- *number_of_children*- number of children class has – will be added
- *number_of_comments* - number of comments – will be added
- *number_of_implemented_interfaces* - number of implemented interfaces – will be added
- *number_of_instructions* - number of instructions
- *number_of_methods* - number of methods
- *lack_of_cohesion_in_methods* - lack of cohesion in methods is calculated as lackOfCohesionInMethods = noOfMethodsWith_noVariableInCommon - noOfMethodsThat_haveVariableInCommon or 0 if previous value is negative
- *depth_of_inheritance* - number of parents a class has – will be added
- *coupling_between_object_classes* - CBO represents the number of other classes a class is coupled to. This metrics is calculated from the callgraph and it counts the reference to methods, variables or types once for each class. – will be added
- *class_complexity* - class complexity, sum of all methods cyclomatic complexities

External.

- *name* - name of external object
- *usr* - usr of external object

B.2.2. Relationships

- App
  - *APP_OWNS_CLASS* Class
- Class
  - *CLASS_OWNS_VARIABLE* Variable

    - *CLASS_OWNS_METHOD* Method
    - *IMPLEMENTS* Class
    - *EXTEND* Class
- Method
    - *USES* Variable
    - *CALLS* Method
    - *CLASS_REF* Class
    - *EXTERNAL_REF* External
- Variable
    - *IS_OF_TYPE* Class

B.2.3. Relationships and nodes added through external analysers

DuplicationAnalyser.

- Class
    - *DUPLICATES* Class

InsiderSecAnalyser.

- Vulnerability
    - *cvss* - common vulnerability score
    - *cwe* - vulnerability class
    - *line* - line on which vulnerability exists
    - *method* - vulnerable method called
    - *description* - description of vulnerability
    - *classPath* - path of vulnerable file
    - *recommendation* - recommendation for removing vulnerability
- Class $HAS_VULNERABILITY$ Vulnerability
- Method $HAS_VULNERABILITY$ Vulnerability

B.3. GraphifyEvolution Code Smell Evolution Queries

B.3.1. How did the number of long methods evolve in Tweetometer app?

As an example we chose the Tweetometer app that has 373 analysed app versions. Of these app versions 173 are affected by a total of 19 long methods. We can query all long methods from the app Tweetometer with the following query:

```
MATCH (app:App)-[:APP_OWNS_CLASS]->()
    -[:CLASS_OWNS_METHOD]->(method:Method)
WHERE
    app.name = "Tweetometer" and
    method.is_long_method = true
WITH
    app, count(distinct method) as long_methods
RETURN
    app.version_number as version, long_methods
```

B.3.2. Did methods become too long over time?

To analyse if methods were created as long methods or if they became too long over time we can query the count of changes before a method became long method with the following query:

```
MATCH (m:Method)
WHERE
    m.is_long_method = true
OPTIONAL MATCH
    p=(:Method)-[:CHANGED_TO*]->(m)
OPTIONAL MATCH
    (m2)-[:CHANGED_TO]->(m)
WHERE
    m2.is_long_method = true
WITH
    m, count(relationships(p)) as changes, m2
WHERE
    m2 is null
RETURN count(*), changes
```

B.3.3. Can we find commits that removed vulnerabilities from code?

We ran app analysis with the InsiderSecAnalyser enabled, which saved vulnerabilities into the application database as nodes and added relationships to vulnerable classes and methods. The following query finds commits that removed a vulnerability from a class:

```
MATCH
    (prev_app)-[:APP_OWNS_CLASS]->(c:Class)-[:HAS_VULNERABILITY]
                                            ->(v:Vulnerability)
MATCH
    (c)-[:CLASS_CHANGED_TO]->(c2)
        <-[:APP_OWNS_CLASS]-(app:App)<-[:CHANGED_TO]-(prev_app)
WHERE
    NOT (c2)-[:HAS_VULNERABILITY]->()
RETURN
    app.name, collect(distinct app.commit), count(v)
```

ACKNOWLEDGEMENTS

First and foremost I would like to thank my supervisor Dietmar Pfahl for the guidance and support throughout my PhD studies which made it a truly enjoyable experience. I would also like to thank my other colleagues, especially Amefon, Aleja, and Hina, for great discussions and laughs during the last years and Gunnar for the memorable conferences and industry perspective.

My biggest gratitude goes to my family, especially my partner Ottar who has supported me through all the ups and downs of the last years. Your love and support means everything to me.

The research in this thesis has been partly funded by BMK, BMAW, and the State of Upper Austria in the frame of the SCCH competence center INTEGRATE (FFG grant no. 892418) part of the FFG COMET Competence Centers for Excellent Technologies Programme, as well as by the European Regional Development Fund, grant PRG1226 of the Estonian Research Council, the IT Academy Programme for ICT Research Development, Estonian Center of Excellence in ICT research (EXCITE) and European Social Fund via IT Academy program.

SISUKOKKUVÕTE

Mobiilirakenduste kvaliteedi analüüs rõhuga hooldatavuse ja turvalisuse aspektidele

Nutitelefonidest on saanud lahutamatu osa meie igapäeva elus. Nutitelefonide populaarsuse pideva tõusuga veedetakse juba täna rohkem aega nutitelefonides kui lauaarvutis. Iga asja jaoks on olemas äpp: sõnumite saatmisekt, internetipanga külastamiseks, auto luku avamiseks. Usaldame nendele rakendustele oma isikuandmeid ja eeldame, et nad hoiavad neid andmeid turvaliselt. Kahjuks on paljud neist rakendustest ebaturvalised. Isegi kui vaadata ainult mõnda väga populaarset mobiilirakendust, nagu Facebook, TikTok ja WhatsApp, on viimastel aastatel avastatud palju turvaauke. Selliste turvaaukude raskusaste võib ulatuda teabe avalikustamisest ja konto ülevõtmisest kuni koodi kaugkäitamiseni. Arvestades kuidas me kasutame oma nutitelefone ja kanname neid kõikjal kaasas, seavad need tõsised turvavead ohtu meie kõige privaatsemad andmed. Turvalisus on tarkvara kvaliteedi üks mõõde. Teine oluline, kuid vähem nähtav aspekt on hooldatavus. Madal koodikvaliteet võib põhjustada suuri hoolduskulusid ja vähendada uute funktsionaalsuste arendamise eelarvet. Seetõttu on oluline tagada, et arendajatel oleks kvaliteetsete mobiilirakenduste loomiseks piisav tööriistatugi.

Tänu oma avatumale olemusele on olemas palju avatud lähtekoodiga Androidi rakendusi, mida saab uuringute läbiviimiseks kasutada. Androidi rakenduste kohta on läbi viidud palju uuringuid, mis analüüsivad koodi kvaliteedi erinevaid aspekte, nagu hooldatavust ja turvalisust. Samuti on uurijate poolt välja töötatud kasulikke tööriistu. Kahjuks on iOS-i rakenduste jaoks väga vähe tööriista tuge ja peaaegu puuduvad vastavad uuringud. Arvestades, et iOS on populaarsuselt teine mobiilne operatsioonisüsteem, on oluline toetada arendajaid kvaliteetsete iOS-i rakenduste loomisel nii turvalisuse kui ka hooldatavuse osas.

Selle töö eesmärk on täiustada tööriista tuge nii arendajatele kui ka teadlastele ning täita mõned uurimustöö lüngad iOS-i rakenduste hooldatavuse ja turbeanalüüside osas. Esiteks töötasime välja GraphifySwift-i, tööriista, mis tuvastab Swift-is kirjutatud projektides kasinaid koodimustreid. Seejärel rakendasime GraphifySwift-i avatud lähtekoodiga iOS-i rakendustele ning analüüsisime kasinate koodimustrite levikut ja sagedust. Lisaks kasutasime iOS-i ja Androidi rakenduste kasinate koodimustrite võrdlemiseks GraphifySwift-i ja PAPRIKA-t.

Analüüsisime iOS-i rakendustes 34 objektorienteeritud kasinat koodimustrit ja võrdlesime 19 objektorienteeritud kasina koodimustri esinemist iOS-is ja Androidis. Leidsime, et iOS-i rakendused kipuvad sisaldama rohkem väikeste ja andmeklassidega seotud kasinaid koodimustreid, samas kui Androidi rakendused sisaldavad rohkem keerukate ja suurte klassidega seotud kasinaid koodimustreid.

Tuginedes GraphifySwift-i arendamisel ja kasutamisel saadud kogemustele, otsustasime oma tööriistakomplekti võimekusi oluliselt suurendada ja arendasime välja GraphifySwift-i laiendatud versiooni GraphifyEvolution-i. GraphfiyEvolution on laiendatav tööriist, millega saab analüüsida nii projektide hetktõmmiseid kui ka projektide arengut. Kasutasime GraphifyEvolution-it esialgseks kasinate koodimustrite evolutsiooni analüüsiks.

Täiendavate väliste analüüsijate lisamiel on võimalik GraphifyEvolution-it laiendada lisaanalüüsivõimalustega. Rakendasime SwiftDependencyChekcer-it – tööriista, mis leiab iOS-i rakendustest teavet kolmandate osapoolte teekide sõltuvuste kohta ja tuvastab ebaturvalisi sõltuvusi. Rakendasime GraphifyEvolution-i jaoks SwiftDependencyChecker-il põhineva välise analüüsija ja kasutasime seda Swift-i ökosüsteemi kolmandate osapoolte teekide jaoks teekide sõltuvusvõrgu andmestiku loomiseks. See andmestik hõlmab teeke, mis on saadaval iOS-i arenduses kasutatava kolme paketihalduri kaudu: CocoaPods, Carthage ja Swift Package Manager. Kasutasime seda andmestikku Swift-i teekide sõltuvusvõrgu erinevate aspektide uurimiseks. Analüüsisime Swift-i teekide sõltuvusvõrgu üldist arengut, paketihaldurite kasutamist, tehnilist mahajäämust teekide sõltuvustes ja turvavigade levikut teekide sõltuvusvõrgus.

Leidsime, et Swift-i teekide sõltuvusvõrk kasvab nii teekide arvu kui ka teegiversioonide arvu poolest. CocoaPods on antud ökosüsteemi populaarseim paketihaldur, Carthage-i kasv on peatunud ja Swift PM muutub aja jooksul aina populaarsemaks. Teekide sõltuvuste tehniline mahajäämus kasvab. Mahajäämus on kõrgem, kui arendajad kasutavad piiravamaid sõltuvusnõude tüüpe. Viimaks leidsime, et ebaturvaliste sõltuvuste protsent on madalam kui teistes ökosüsteemides, mis on seletatav üldiselt väiksema transitiivsete sõltuvuste arvuga.

CURRICULUM VITAE

Personal data

Name:	Kristiina Rahkema
Date of Birth:	22.05.1989
e-mail:	kristiina.rahkema@ut.ee
ORCID:	0000-0001-7332-2041

Education

2019–(2023)	University of Tartu, Tartu, Estonia, PhD in Computer Science
2013–2016	University of Tartu, Tartu, Estonia, MSc in Software Engineering
2011-2012	Coventry University, Coventry, United Kingdom, BSc in Informatics
2009–2013	University of Tartu, Tartu, Estonia, BSc in Mathematics

Employment

2022–2023	N-Day research advocate intern, Exodus Intelligence, USA
2019–2023	Junior research fellow of Software Engineering, University of Tartu, Estonia
2017–2019	iOS developer, Nestri Solutions, Estonia
2013–2017	Study assistant in Software Engineering, University of Tartu, Estonia
2012–2014	Quality assurance engineer, Fortumo, Estonia
2010–2011	iOS developer, Indilo Wireless, Estonia

Scientific work

Main fields of interest:

- Mobile applications
- iOS
- Security

ELULOOKIRJELDUS

Isikuandmed

Nimi:	Kristiina Rahkema
Sünniaeg:	22.05.1989
e-mail:	kristiina.rahkema@ut.ee
ORCID:	0000-0001-7332-2041

Haridus

2019–(2023)	Tartu Ülikool, Tartu, Eesti, PhD Informaatika
2013–2016	Tartu Ülikool, Tartu, Eesti, MSc Tarkvaratehnika
2011-2012	Coventry University, Coventry, Suurbritannia, BSc Informaatika
2009–2013	Tartu Ülikool, Tartu, Estonia, BSc Matemaatika

Teenistuskäik

2022–2023	Avalike turvavigade analüüsi praktikant, Exodus Intelligence, Ameerika Ühendriigid
2019–2023	Tarkvaratehnika nooremteadur, Tartu Ülikool, Eesti
2017–2019	iOS arendaja, Nestri Solutions, Eesti
2013–2017	Tarkvaratehnika õppeassistent, Tartu Ülikool, Eesti
2012–2014	QA insener, Fortumo, Eesti
2010–2011	iOS arendaja, Indilo Wireless, Eesti

Teadustegevus

Peamised uurimisvaldkonnad:

- Mobiilirakendused
- iOS
- Turvalisus

LIST OF ORIGINAL PUBLICATIONS

Publications in the scope of the thesis

I **Rahkema, Kristiina**, and Dietmar Pfahl. "Empirical study on code smells in iOS applications." In Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems, pp. 61-65. 2020.
Lead author. The author performed the implementation and the analysis of the experiments and contributed substantially to the ideas and the writing.

II **Rahkema, Kristiina**, and Dietmar Pfahl. "Comparison of Code Smells in iOS and Android Applications." In QuASoQ@ APSEC, pp. 79-86. 2020.
Lead author. The author performed the implementation and the analysis of the experiments and contributed substantially to the ideas and the writing.

III **Rahkema, Kristiina**, and Dietmar Pfahl. "GraphifyEvolution-A Modular Approach to Analysing Source Code Histories." In 2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft), pp. 24-27. IEEE, 2021.
Lead author. The author performed the implementation and the analysis of the experiments and contributed substantially to the ideas and the writing.

IV **Rahkema, Kristiina**, and Dietmar Pfahl. "SwiftDependencyChecker: detecting vulnerable dependencies declared through CocoaPods, carthage and swift PM." In Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems, pp. 107-111. 2022.
Lead author. The author performed the implementation and the analysis of the experiments and contributed substantially to the ideas and the writing.

V **Rahkema, Kristiina**, and Dietmar Pfahl. "Dataset: dependency networks of open source libraries available through CocoaPods, Carthage and Swift PM." In Proceedings of the 19th International Conference on Mining Software Repositories, pp. 393-397. 2022.
Lead author. The author performed the implementation and the analysis of the experiments and contributed substantially to the ideas and the writing.

VI **Rahkema, Kristiina**, and Dietmar Pfahl. "Analysing the Relationship Between Dependency Definition and Updating Practice When Using Third-Party Libraries." In Product-Focused Software Process Improvement: 23rd International Conference, PROFES 2022, Jyväskylä, Finland, November 21–23, 2022, Proceedings, pp. 90-107. Cham: Springer International Publishing, 2022.
Lead author. The author performed the implementation and the analysis of the experiments and contributed substantially to the ideas and the writing.

VII **Rahkema, Kristiina**, Dietmar Pfahl and Rudolf Ramler. "Analysis of Library Dependency Networks of Package Managers Used in iOS Development." In Proceedings of the IEEE/ACM 10th International Conference on

Mobile Software Engineering and Systems, pp. 23-27. 2023.
Lead author. The author performed the implementation and the analysis of the experiments and contributed substantially to the ideas and the writing.

VIII **Rahkema, Kristiina**, and Dietmar Pfahl. "Vulnerability Propagation in Package Managers Used in iOS Development." In Proceedings of the IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems, pp. 60-69. 2023.
Lead author. The author performed the implementation and the analysis of the experiments and contributed substantially to the ideas and the writing.

74. **Meelis Kull.** Statistical enrichment analysis in algorithms for studying gene regulation. Tartu 2011, 151 p.
77. **Bingsheng Zhang.** Efficient cryptographic protocols for secure and private remote databases. Tartu 2011, 206 p.
78. **Reina Uba.** Merging business process models. Tartu 2011, 166 p.
79. **Uuno Puus.** Structural performance as a success factor in software development projects – Estonian experience. Tartu 2012, 106 p.
81. **Georg Singer.** Web search engines and complex information needs. Tartu 2012, 218 p.
83. **Dan Bogdanov.** Sharemind: programmable secure computations with practical applications. Tartu 2013, 191 p.
84. **Jevgeni Kabanov.** Towards a more productive Java EE ecosystem. Tartu 2013, 151 p.
87. **Margus Freudenthal.** Simpl: A toolkit for Domain-Specific Language development in enterprise information systems. Tartu, 2013, 151 p.
90. **Raivo Kolde.** Methods for re-using public gene expression data. Tartu, 2014, 121 p.
91. **Vladimir Šor.** Statistical Approach for Memory Leak Detection in Java Applications. Tartu, 2014, 155 p.
92. **Naved Ahmed.** Deriving Security Requirements from Business Process Models. Tartu, 2014, 171 p.
94. **Liina Kamm.** Privacy-preserving statistical analysis using secure multi-party computation. Tartu, 2015, 201 p.
100. **Abel Armas Cervantes.** Diagnosing Behavioral Differences between Business Process Models. Tartu, 2015, 193 p.
101. **Fredrik Milani.** On Sub-Processes, Process Variation and their Interplay: An Integrated Divide-and-Conquer Method for Modeling Business Processes with Variation. Tartu, 2015, 164 p.
102. **Huber Raul Flores Macario.** Service-Oriented and Evidence-aware Mobile Cloud Computing. Tartu, 2015, 163 p.
103. **Tauno Metsalu.** Statistical analysis of multivariate data in bioinformatics. Tartu, 2016, 197 p.
104. **Riivo Talviste.** Applying Secure Multi-party Computation in Practice. Tartu, 2016, 144 p.
108. **Siim Orasmaa.** Explorations of the Problem of Broad-coverage and General Domain Event Analysis: The Estonian Experience. Tartu, 2016, 186 p.
109. **Prastudy Mungkas Fauzi.** Efficient Non-interactive Zero-knowledge Protocols in the CRS Model. Tartu, 2017, 193 p.
110. **Pelle Jakovits.** Adapting Scientific Computing Algorithms to Distributed Computing Frameworks. Tartu, 2017, 168 p.
111. **Anna Leontjeva.** Using Generative Models to Combine Static and Sequential Features for Classification. Tartu, 2017, 167 p.
112. **Mozhgan Pourmoradnasseri.** Some Problems Related to Extensions of Polytopes. Tartu, 2017, 168 p.