Saarland University

Improving Android App Security and Privacy with Developers

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von
Duc Cuong Nguyen

Saarbrücken, 2020

Tag des Kolloquiums:	08.07.2021
Dekan:	Prof. Dr. Thomas Schuster

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Christian Rossow
Berichterstattende:	Prof. Dr. Michael Backes
	Prof. Gang Wang, PhD.
	Dr. Katharina Krombholz
Akademischer Mitarbeiter:	Dr. Robert Künnemann

Abstract

Existing research has uncovered many security vulnerabilities in Android applications (apps) caused by inexperienced, and unmotivated developers. Especially, the lack of tool support makes it hard for developers to avoid common security and privacy problems in Android apps. As a result, this leads to apps with security vulnerability that exposes end users to a multitude of attacks.

This thesis presents a line of work that studies and supports Android developers in writing more secure code. We first studied to which extent tool support can help developers in creating more secure applications. To this end, we developed and evaluated an Android Studio extension that identifies common security problems of Android apps, and provides developers suggestions to more secure alternatives. Subsequently, we focused on the issue of outdated third-party libraries in apps which also is the root cause for a variety of security vulnerabilities. Therefore, we analyzed all popular 3rd party libraries in the Android ecosystem, and provided developers feedback and guidance in the form of tool support in their development environment to fix such security problems. In the second part of this thesis, we empirically studied and measured the impact of user reviews on app security and privacy evolution. Thus, we built a review classifier to identify security and privacy related reviews and performed regression analysis to measure their impact on the evolution of security and privacy in Android apps. Based on our results we proposed several suggestions to improve the security and privacy of Android apps by leveraging user feedbacks to create incentives for developers to improve their apps toward better versions.

Zusammenfassung

Die bisherige Forschung zeigt eine Vielzahl von Sicherheitslücken in Android-Applikationen auf, welche sich auf unerfahrene und unmotivierte Entwickler zurückführen lassen. Insbesondere ein Mangel an Unterstützung durch Tools erschwert es den Entwicklern, häufig auftretende Sicherheits- und Datenschutzprobleme in Android Apps zu vermeiden. Als Folge führt dies zu Apps mit Sicherheitsschwachstellen, die Benutzer einer Vielzahl von Angriffen aussetzen.

Diese Dissertation präsentiert eine Reihe von Forschungsarbeiten, die Android-Entwickler bei der Entwicklung von sichereren Apps untersucht und unterstützt. In einem ersten Schritt untersuchten wir, inwieweit die Tool-Unterstützung Entwicklern beim Schreiben von sicherem Code helfen kann. Zu diesem Zweck entwickelten und evaluierten wir eine Android Studio-Erweiterung, die gängige Sicherheitsprobleme von Android-Apps identifiziert und Entwicklern Vorschläge für sicherere Alternativen bietet. Daran anknüpfend, konzentrierten wir uns auf das Problem veralteter Bibliotheken von Drittanbietern in Apps, die ebenfalls häufig die Ursache von Sicherheitslücken sein können. Hierzu analysierten wir alle gängigen 3rd-Party-Bibliotheken im Android-Ökosystem und gaben den Entwicklern Feedback und Anleitung in Form von Tool-Unterstützung in ihrer Entwicklungsumgebung, um solche Sicherheitsprobleme zu beheben. Im zweiten Teil dieser Dissertation untersuchten wir empirisch die Auswirkungen von Benutzer-Reviews im Android Appstore auf die Entwicklung der Sicherheit und des Datenschutzes von Apps. Zu diesem Zweck entwickelten wir einen Review-Klassifikator, welcher in der Lage ist sicherheits- und datenschutzbezogene Reviews zu identifizieren. Nachfolgend untersuchten wir den Einfluss solcher Reviews auf die Entwicklung der Sicherheit und des Datenschutzes in Android-Apps mithilfe einer Regressionsanalyse. Basierend auf unseren Ergebnissen präsentieren wir verschiedene Vorschläge zur Verbesserung der Sicherheit und des Datenschutzes von Android-Apps, welche die Reviews der Benutzer zur Schaffung von Anreizen für Entwickler nutzen.

Background of this Dissertation

This dissertation is based on the papers mentioned in the following. I contributed to all papers as the main author.

The initial work - FixDroid [P1] was based on the author's master thesis proposed, and advised by Sascha Fahl. In the master thesis, a set of common security pitfalls in Android was extracted from existing research and from the Android official documentation. A prototype version of FixDroid was then developed to help developers avoid such security problems. FixDroid was then tested in a pilot study to get first hands-on feedback on its feasibility and on how Android developers would used it. During his PhD, the author had redesigned and reimplemented major parts of FixDroid since a significant part of the internal API of Android Studio that FixDroid depended on had changed. This work then was evaluated with participants (N=39) in a new study to evaluate FixDroid's effects on the code delivered by developers. The author was responsible for major parts of the design, implementation, developer study, and paper writing with the feedback from Sascha Fahl, Yasemin Acar, and Michael Backes. Sascha Fahl, Charles Weir were involved in the general writing. All authors performed reviews of the paper.

The initial idea of Up2Dep [P2] was proposed by Erik Derr and Sven Bugiel. The author joint the discussion of Up2Dep with Erik and Sven on the topic, which eventually shaped the idea of Up2Dep toward tool support. The author further extended the initial idea to bring in more security focus by integrating cryptographic analysis into Up2Dep. The author was then responsible for the major parts of the design including the developer study, implementation, evaluation, and paper writing with the feedback of Sven Bugiel, Erik Derr. Sven Bugiel and Erik were involved in the general writing of the paper. All authors performed reviews of the paper.

The author had the initial idea and motivation for measuring the impact of user reviews on app security & privacy evolution [P3]. The author was further responsible for major parts of the design, implementation, evaluation, and paper writing with the feedback from Erik Derr, Sven Bugiel, and Michael Backes. Sven Bugiel contributed with the idea of using regression analysis to evaluate the impact of security & privacy related reviews on app security & privacy evolution, the author was then responsible for realizing this idea. Erik Derr provided his support in using his static analysis tools (*LibScout* and *Axplorer*) to analyze Android apps. Sven Bugiel and Erik Derr were involved in the general writing of the paper. Michael Backes provided feedback on the general direction of the project. In general, all authors performed reviews of the paper.

Author's Papers for this Thesis

[P1] Nguyen, D. C., Wermke, D., Acar, Y., Backes, M., Weir, C., and Fahl, S. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. ACM, 2017.

[P2] Nguyen, D. C., Derr, E., Backes, M., and Bugiel, S. Up2Dep: Android Tool Support to Fix Insecure Code Dependencies. In: *Proceedings of the 36th Annual Computer Security Applications Conference*. ACSAC '20. Association for Computing Machinery, 2020.

[P3] Nguyen, D. C., Derr, E., Backes, M., and Bugiel, S. Short Text, Large Effect: Measuring the Impact of User Reviews on Android App Security & Privacy. In: *2019 IEEE Symposium on Security and Privacy (SP'19)*. 2019.

Further Publications of the Author

[S1] Nguyen, T. T., Nguyen, D. C., Schilling, M., Wang, G., and Backes, M. Measuring User Perception for Detecting Unexpected Access to Sensitive Resource in Mobile Apps. In: *Proceedings of the 16th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '21. Association for Computing Machinery, 2021.

[S2] Stransky, C., Acar, Y., Nguyen, D. C., Wermke, D., Kim, D., Redmiles, E. M., Backes, M., Garfinkel, S. L., Mazurek, M. L., and Fahl, S. Lessons Learned from Using an Online Platform to Conduct Large-Scale, Online Controlled Security Experiments with Software Developers. In: *10th USENIX Workshop on Cyber Security Experimentation and Test, CSET 2017, Vancouver, BC, Canada, August 14, 2017*. 2017.

Acknowledgments

First and foremost I would like to thank my advisor Michael Backes for giving me the opportunity to pursue my research area in the Information Security & Cryptography Group/CISPA Helmholtz Center for Information Security. Michael has always been supportive to me. He gave me freedom to do research in the areas of my interests which always motivates me to do the work I love. I am honored to be one of his students. I am also thankful to have had Sascha Fahl as my master thesis advisor which initially sparked my interest to pursue the research career path that eventually led me to my PhD. Sascha also further advised me during the first months of my PhD. Without Sascha initial support, I could not imagine I would have spent years of doing research.

I am also very much thankful to Sven Bugiel for giving me helpful advice and guiding me in different research projects. Sven has been really understanding and supportive. During my PhD, I have learned a lot from Sven. Besides, I would like to thank all my closest colleagues, collaborators, and co-authors, without them this dissertation would not have been possible: Ben Stock, Erik Derr, Michael Schilling, Charles Weir, Yasemin Acar. I have gained valuable knowledge in many different aspects of doing research from every one of them.

I would like to thank Michael Schilling for spending his time proof reading this thesis and providing me valuable feedback.

My special thanks to Sanam Lyastani, Michaela Neumayr, and Michael Schilling for being the best officemates ever. I really love the time we spend together. We are not just colleagues, but also friends. I am thankful to have Michaela and Michael to *practice* my German skills and to discuss many interesting topics during our Mensa time. Further, thanks to our beloved team assistant Bettina Balthasar who helps us with many bureaucratic tasks and who has always been so nice and supportive to us. Besides, I really appreciate the help of the CISPA staff during my stay, especially of Olga Kill, Julia Schwarz, Stephanie Feyahn, Sebastian Weisgerber, and Deborrah Kolz.

Moreover, I like to express my gratitude towards Katharina Krombholz and Gang Wang, who agreed to be examiners of this thesis.

Finally, huge thanks to my lovely wife — Trinh, who shares my journey and has been supportive to all my decisions. Most importantly, I am deeply grateful to my parents for their ultimate support and unconditional love. Without them I would not have been able to experience all the wonderful journeys.

Contents

List of Figures

List of Tables

List of Code Listings

1

Introduction

Smart phones have become an essential part of our daily life. They make our life easier and more convenient. Thereby, smart phones make use of sensors (camera, GPS, microphone, etc.), and other information of the users to personalize user experience. One of the primary factors that contribute to the success of smart phones is third-party applications (apps). Particularly, third-party apps have diversified the potential of smart phones which ultimately serves many aspects of user's needs. To distribute apps to end users, market store was introduced that allows users to browse and download for apps that suit their need. Google's Play Store - the most popular market store for the Android ecosystem currently has 2.96 million apps as of June 2020 [116]. The ubiquity of smart phones and their ability to access user personal information attract an ever increasing number of malicious third-party entities.

Existing research has shown a variety of attacks that abuse end user's personal information [67, 159, 124, 104]. While many works have been proposed to mitigate and deal with malicious third-party applications, they often target post-development phase where apps might have been released and already available to end users [40, 159, 162, 67]. The number of benign apps that have security & privacy problems *unintentionally* created by benign developers still increases. Existing work highlighted two main reasons for this problem namely developers' inability to write secure code [6, 37] and the lack of motivation to adhere to security & privacy best practices [3].

Particularly, developers can be inexperienced, and might not possess enough security knowledge to write secure code [59, 60, 41, 51, 56]. Besides, existing code is often reused by developers in the form of third-party libraries and when a library contains bugs or security & privacy issues, those flaws could be amplified when the affected versions of such a library are integrated in different applications [19]. Even when security fixes are available in newer versions of the affected libraries, their adoption by developers progresses very slowly as this does not seem to be an easy task mainly due to the lack of tool support [49].

Furthermore, developers might not even be motivated enough to think about the security & privacy of the end users as security is only their secondary concern [3, 115]. While both research community and the industry can create advanced techniques to mitigate security vulnerabilities, and to help developers avoid such vulnerabilities, such solutions would not be effective if developers do not pay attention to app security & privacy i.e., security is a secondary concern. As a result, this leads to apps that are vulnerable to multitude levels of attacks affecting millions of end users.

This thesis aimed to examine this topic as comprehensively as possible, we thereby explored both major types of potentially influential factors to app developers, their ability to write secure & privacy preserving code as well as their motivation to do so in the first place. In particular, we set to investigate on how we can *help* and *motivate* developers to write more secure and privacy preserving code. In the first part, our goal was to empirically study to which extent tool support helps developers write more secure code. To this end, we targeted the development phase of apps, and built an Android Studio extension that identifies common security problems of Android projects,

and provides developers suggestions to more secure alternatives. In the second part, we looked at the lack of tool support that contributes to the security & privacy problems of Android apps caused by outdated (insecure) third-party libraries. Specifically, we aimed to further extend our tool support to help developers tackle such problems. To this end, we built a second Android Studio extension to analyze Android third-party libraries and provide developers information regarding the updatability and security of third-party libraries included in developer's projects. In the third part of this thesis, we then turned to other aspect that influences the security & privacy of apps namely developer's motivation. Particularly, we focused on end user reviews and set to examine whether end users pay attention to app security & privacy related issues, whether they communicate with app developers on such matters, and finally what is the corresponding effect on app's security & privacy. Thus, we set to measure the impact of user reviews on app security & privacy evolution. We applied machine learning techniques to build a review classifier to identify security & privacy related reviews and performed several regression analysis to measure their impact on app security & privacy evolution. Based on our results, we could then identify insights to improve the security & privacy of Android apps by leveraging user feedbacks to create motivation for developers to increase the security & privacy of their apps.

Summary of contributions

In the following, we summarize the major contributions of this dissertation:

FixDroid FixDroid is the first Android Studio extension that identifies common security problems in Android projects and proposes more secure alternative (Chapter 3). When developers writes code, FixDroid performs static code analysis on developer's project to find common security problems and provides developers suggestions to more secure alternatives. In our study with professional and hobby app developers, we showed that code delivered with the support of FixDroid contain significantly less security problems.

Up2Dep Up2Dep is a first implemented solution that supports app developers in their task to avoid outdated, insecure libraries (Chapter 4). Up2Dep analyzes third-party libraries to provide developers with information about the changes that they may need to perform when updating a library, based on the public API changes between the library versions. Using the collected information about library APIs and their usages on a given project, Up2Dep provides developers feedback on the updatability of outdated library versions. Up2Dep also maintains a database of publicly disclosed vulnerabilities and cryptographic API misuse of libraries, and alerts developers if a vulnerable library version was included in their apps. To evaluate how Up2Dep could support developers in fixing insecure code dependencies, we publicly released Up2Dep and tested invited developers in their daily task. Up2Dep has delivered quick-fixes that could mitigate dependencies with security problems in *real* projects. Up2Dep was further perceived by those developers as being helpful.

The impact of user reviews on app security & privacy This work presents the first study on the relationship between end-user reviews and security- & privacy-related changes in apps (Chapter 5). Using supervised learning techniques, we built a classifier to detect security and privacy related reviews (SPR). Using static code analysis, we then could identify the changes between those user reviewed app versions and their immediate successor versions as security & privacy relevant when later app versions behave more privacy friendly. Finally, we built a statistical regression model that takes different factors into account that could affect the update of an app to measure the impact of user reviews on app security & privacy evolution. Our results showed that user reviews in fact lead to privacy improvements of apps.

Outline The remaining of this dissertation is structured as follows. In Chapter 2 we provide common technical background information on the Android platform and its main components. We then present FixDroid in Chapter 3 and Up2Dep in Chapter 4. The impact of user reviews on app security & privacy is presented in Chapter 5. We then conclude this dissertation in Chapter 6.

2

Technical Background on Android

This chapter introduces background information on the Android platform and its fundamental components.

2.1 Android Software Stack

Android is a software system based on a modified version of the Linnux kernel, and is designed for smart devices (e.g., smart phones, tablets, wearable devices). Android was initially developed by Android Inc., which was then bought by Google in 2005 and later made it debut in 2008 with Android 1.0. Since then Android has had multiple versions ranging from 1.0 to 10.1. Each version is specified with an API level to manage compatibility among these versions. Android is a software stack consisting of four software layer (see Figure 2.1).

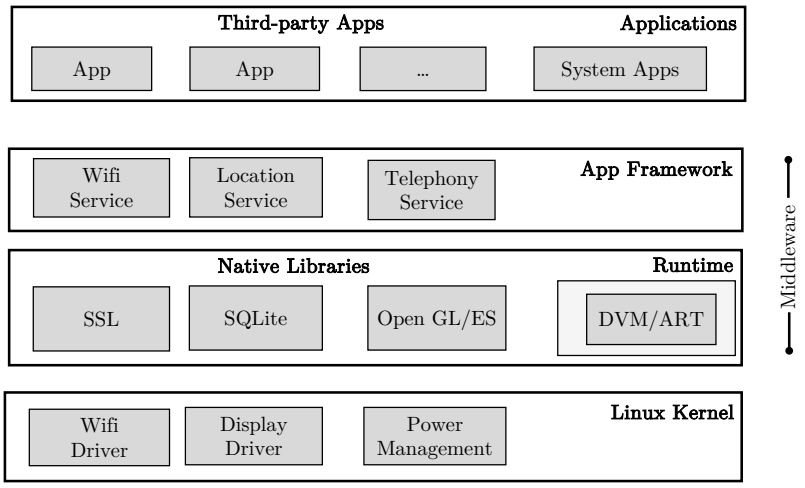

Figure 2.1: The Android software stack. Figure adopted by (27).

Linux Kernel Linux Kernel is responsible for basic operating system services and enables accesses of the operating system to low-level hardware. *Linux Kernel* in Android has been modified to accommodate the resource-constraint of mobile devices.

Android Middleware Android Middleware consists of *native libraries*, *Android runtime* featuring Dalvik Virtual Machine (DVM), and the *application framework*. DVM is a register-based virtual machine executing Android applications. Unlike traditional computers, everything in mobile is limited, DVM had been therefore optimized for

resource constrained mobile devices. DVM was later replaced by Android Runtime (ART) starting from Android version 5.0. ART performs code compilation a head-of-time to optimize for performance. Additionally, in Android 7.0 just-in-time complier was introduced to improve compilation performance.

Applicaton Framework Applicaton Framework provides a variety of API for Android applications to access system resources such as location service, wifi service, settings. These APIs are used to develop Android apps including both pre-installed and third-party apps.

2.2 Android Apps

Android apps are usually written in Java, and since 2017 in Kotlin programming languages [72], then are compiled to *dex* bytecode for execution inside DVM or ART. Besides, developers can also develop apps in C and C++ languages using the *Native Development Kit*. Similar to other platforms, many third-party libraries are developed such that developers can re-use existing code in the form of (third-party) libraries which eases the development task. Libraries are included in the form of bundled Java bytecode (.jar file) or as Android Archive Library (.aar file). With Android Archive Library, library developers can include resource files besides the code files. During the building process, app code and library code files are merged together.

App Structure Every Android app must have an *AndroidManifest.xml* file that declares all essential components of the app. An *AndroidManifest.xml* file also contains information about permissions, minimum API level, third-party libraries, and hardware features that the app requires to run.The following components are crucial:

- *Activity*: An activity is a foreground task that implements user interface and serves as they entry point for a user into an app.
- *Service*: In contrast to *Activity*, a service runs in the background and performs long-running tasks. It does not have user interface.
- *BroadcastReceiver*: A broadcast receiver is used to register for system and application events.
- *ContentProvider*: A content provider is used to manage access to data of the app itself, or stored by other apps. It also allows data sharing among apps.

App Permission Android introduces *Permission* system to govern accesses to sensitive resources (e.g., Location, Storage, SMS ...) and to separate privileges among apps. *Install-time* permission was introduced where users have to either grant all permissions and install the app or deny all permissions and stop the installation process. Since Android 6.0, *run-time* permission was introduced to and the users can grant or deny permissions during app's run-time.

2.3 App Development Tool

At the early stage, Google provided the Android Developer Tools (ATD) in the form of a third-party plugin for Eclipse integrated development environment (IDE) so that it could be used to develop Android applications. In December 2014, Google released Android Studio as a standalone and the official supported tool for Android app development [14]. Android Studio is built based on JetBrain's IntelliJ IDEA software and designed for Android development. Plugins play an important role in extending Android Studio's feature. The following are some of the essential features that are built in the form of plugins for Android Studio:

- Manage project's dependencies and building process: *Gradle*
- Catching performance and usability issues, syntax highlighting, code inspection: *Lint*
- Managing code version control: *GitIntegration*
- Unit testing: *Junit*

Android Studio Plugin Development Android Studio is based on Jetbrain's IntelliJ IDEA. Therefore, to develop an Android Studio plugin one needs to create an IntelliJ IDEA plugin that targets Android Studio. The IntelliJ platform provides tools designed for static code analysis, i.e., inspections that allow developers to check for potential problems in the source code. Examples of such inspections are finding probable bugs, dead code, performance issues; improving code structure and quality; and examining coding practices and guidelines. Code inspection in Android Studio leverages the program structure interface (PSI) to analyze source code files of a project. PSI is responsible for parsing files and creating syntactic as well as semantic code models. This allows the IDE to efficiently perform static code analysis on a project's source code such as identifying code inconsistency, probable bugs, and specification violations. There are two main program structure interfaces in IntelliJ IDEA namely *PsiFile* and *PsiElement*. *PsiFile* represents the content of a code file as a hierarchy of elements (so-called *PsiTree*). Each specific programming language can extend the *PsiFile* base class to have its own representation, such as *PsiJavaFile* for Java language, *GroovyFileBase* for Groovy language, or *KtFile* for Kotlin language. *PsiElements* are used to explore the internal structure of a project's source code by the IntelliJ platform. Specifically, *PsiElements* are used to perform code inspection and quick-fixes on IntelliJ IDEA/Android Studio projects. When a quick-fix is applied, *PsiElements* are updated, removed from, or additionally added to an existing *PsiFile*. To analyze developer's code, one can extend the *InpsectionProfileEntry* class to build a *PsiElementVisitor* that traverses through all *PsiElements* belonging to a *PsiFile*. Each *PsiElement* corresponds to a keyword, a variable, or an operation in a particular language. To apply a quick-fix, e.g., replacing an insecure code snippet with a more secure one. In such a case, a new *PsiElement* representing a more secure code snippet is created and replaces the existing *PsiElement* that represents the insecure code snippet.

2.4 Market Store

Third-party applications can be distributed from centralized software distribution channels namely market stores or from external sources such as SDCard. To distribute apps via market stores such as Google Play, third-party developers need to submit their apps to Google Play. The apps must adhere to Google's policy, and will go through automated vetting systems such as *Google Play Protect* [69] for quality control, censorship, and security protection. These systems scan apps for malicious intention. They execute apps in a simulated environment to detect hidden malicious behavior. Once an app has been vetted, it will be available on the store and end users can then search, download, and install it.

User reviews App users can share their experience — in the form of review, with a rating score — with other users on application stores such that the other users can decide whether or not to install an app. Application store further allows end users to provide direct feedback to developers on the apps that they are using. Developers in turn can use this channel to respond (reply) to user reviews. A reply can have maximum 350 characters limit.

3

Supporting Android Developers in Writing Secure Code

3.1 Motivation

Despite security advice in the official documentation and an extensive body of security research about vulnerabilities and exploits, many developers still fail to write secure Android applications. Frequently, Android developers fail to adhere to security best practices, leaving applications vulnerable to a multitude of attacks. This work points out the advantage of a low-time-cost tool both to teach better secure coding and to improve app security. Using the FixDroid™ IDE plug-in, we show that professional and hobby app developers can work with and learn from an in-environment tool without it impacting their normal work; and by performing studies with both students and professional developers, we identify key UI requirements and demonstrate that code delivered with such a tool by developers previously inexperienced in security contains significantly less security problems. Perfecting and adding such tools to the Android development environment is an essential step in getting both security and privacy for the next generation of apps.

3.2 Problem Description

The introduction of Android to the mobile operating system market led to the development of new paradigms and open standards on mobile systems. Today, Google's operating system is among the most used mobile operating systems with the largest installed base of any operating system. A major contributor to this success is the Google Play market with its free and paid apps for any and all circumstances, from ordering food to playing card games. The market currently allows Android users to install over 2.9 million apps from third-party developers and, when installed, run the apps on their mobile system. The benefits of a large Android app environment thus come with a number of security and privacy related risks for a user, especially due to errors by app developers. Therefore, it is especially important to secure third-party apps, by encouraging and enabling third-party developers to write secure code.

Many available mobile apps have poorly implemented privacy and security mechanisms, possibly resulting from developers who are inexperienced, distracted, or overwhelmed [6]. Risk-factors leading to insecure code include general inexperience of developers, a sole focus on code functionality while ignoring security implications, and careless adopting of code parts from unverified online information sources [6]. Even worse, some developers just copy and paste code they find when searching for a solution to their security related issues [94]. Even in the absence of these security-neglecting actions by developers, benign failure to write privacy preserving or secure code can lead to applications that leave user data vulnerable to leaks and attacks. Developers have been found to risk users' privacy and security by requesting more permissions than actually needed [125, 128], by not using TLS [59, 60], by failing to use cryptographic APIs correctly [52], by using dangerous options for Inter-Component Communication [40], and by failing to store sensitive information in private areas [57].

Although the Android environment provides users with a number of tools and policies to counter security problems and manage privacy risks, the issues above prove that these are not sufficient to prevent insecure Android apps. We propose that supporting App developers in a developer-friendly and compelling manner in making choices will result in improved security and privacy for the app users. Teaching a developer about secure coding practices will not only help the developer, but will also result in increased security and privacy for every user that runs apps by that developer.

3.3 Contributions

To support Android developers in writing secure code, we developed the FixDroid tool. As plugin for the officially supported Integrated Development Environment (IDE) of Android, Android Studio, FixDroid highlights security and privacy related code problems, provides an explanation to developers, and suggests 'quick fix' options. Similar to a spell-checker in a modern word-processor, FixDroid highlights code snippets that impact the security or privacy of the app. FixDroid builds upon the concept of Android Lint, a tool included in the official Android Software Development Kit (SDK), but avoids certain limitations and improves the support for developers.

To evaluate the usability and acceptance of a FixDroid prototype, we performed a pilot study with 9 developers. With knowledge from this pilot study, we improved FixDroid and developed study platform integrated inside FixDroid that allows conducting Android developers study remotely and fully automatic. On this platform, we then conducted a remote developer study with Android developers and students (N=39) to evaluate the security benefits of FixDroid.

The study proved the effectiveness of this approach by reducing the number of security errors in resulting code. It also validated the approach of using remote IDE telemetry as a means for evaluating developer behavior and showed the importance of having very clear visible indicators for security errors.

The main contributions of this work are:

- Proving the effectiveness of an interactive IDE-based security review tool in improving the security and privacy aspects of code written by third party developers,
- Identifying new UI requirements for such a tool based on feedback from developers,
- Delivering evaluations of the effectiveness of such a tool with both experienced professional developers and less experienced student developers, and
- Validating the use of telemetry in an IDE to determine programmer behavior.

3.4 Technical Description and Approach

3.4.1 Android Application Development

The mobile operating system Android includes the Google Play market with access to over 2.9 million user-developed apps. In the early days of Android app development, developers either relied on the Eclipse IDE with the Android Development Tools (ADT) plugin or the NetBeans IDE with plugin for writing apps. In December 2014 Google released the Android Studio IDE based on JetBrains' IntelliJ IDEA, which functions together with the SDK as officially supported Android IDE. Features of Android Studio include a Gradle-based build system, and an Android Device emulator for testing apps.

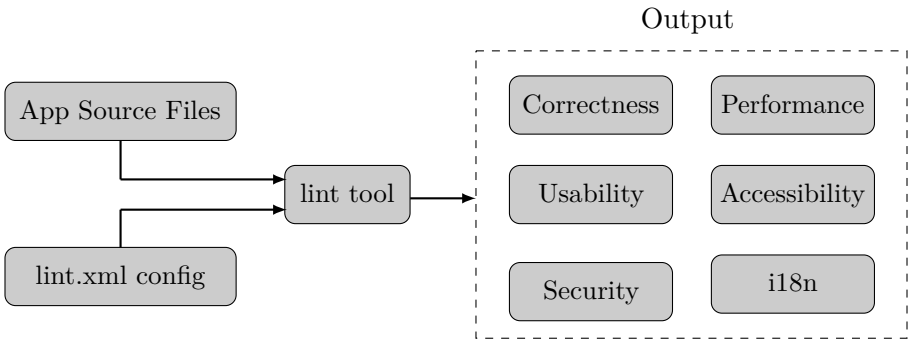

Figure 3.1: Code scanning work flow with Lint tool. Figure adopted by (70).

3.4.1.1 Android Lint Tool

In addition to functional checks, the Android SDK includes the Android Lint code scanning tool to detect problems with the structural quality of code. The lint tool takes a configuration file and the source files of an app, performs static code analysis, and highlights over 200 problems[1] in the categories of correctness, security, performance, usability, accessibility, and internationalization, (cf. Figure 3.1). Examples of the problems lint highlights include missing permissions for requested APIs, using a mock location provider in production, and initializing a random number generator with a fixed seed.

3.4.1.2 Lint Shortcomings

Though Lint is a very useful tool that helps developers to improve code quality in general and some aspects of software security in particular, its current implementation does not support the app developer optimally. The following sections identify a couple of drawbacks that limit its effectiveness, and suggest actionable changes for each to make Lint security more effective.

[1] https://sites.google.com/a/android.com/tools/tips/lint-checks

```
Cipher cipher = Cipher.getInstance("AES");
```

Figure 3.2: A vague highlighted code.

Limited User Interface Lint security uses 'vague highlighting' for detected security issues (e.g. ECB mode for cryptography) - cf. Figure 3.2. This way of highlighting insecure code snippets has two drawbacks:

- Lint uses the same highlighting for all kinds of warnings, i.e. non-security related bad code smells are highlighted in the same way as security related bad code smells.
- Using the same highlighting for all sorts of coding problems may lead to habituation and even to overlooking the highlighting entirely.

> Proposed Action:
> To attract the developer's attention, the user interface should consider insights from previous usable security and privacy research [136, 138, 54].

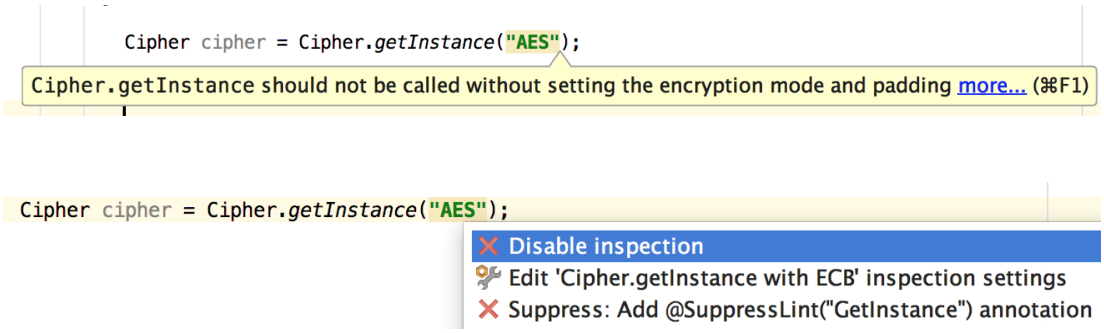

Figure 3.3: Lint does not provide help in term of quick-fixes for security bad practices .

No Way Out While Lint highlights security problems and even provides textual information in the form of tool tips (cf. Figure 3.3), it does not guide developers through the process of turning insecure code into secure code. Although this is not possible in all cases (in particular in cases that spread insecurities across many different methods, classes and packages), in many cases developers could be instructed to apply secure coding practices. Examples might be not using an empty TrustManager implementation, or replacing an insecure mode of operation such as ECB for symmetric cryptography.

> Proposed Action:
> Provide easy-to-use code snippets to turn insecure code into secure code in as many cases as possible [64].

Limited Data Flow Analysis Lint has a lightweight data flow analysis to detect programming issues [146, 91]. It is able to detect obvious security issues such as using ECB for symmetric encryption or a HostNameVerifier that returns true (cf. Figure 3.4).

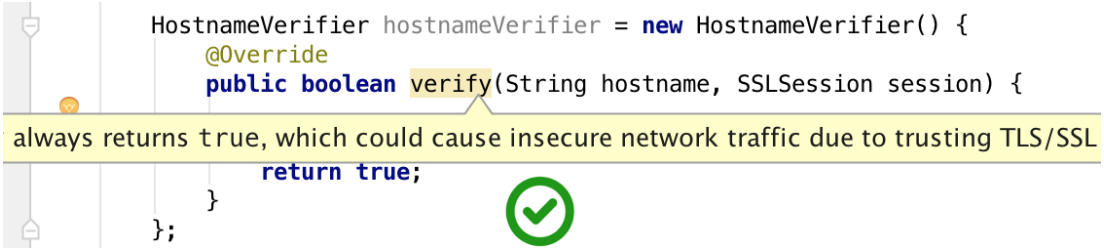

Figure 3.4: Android Lint is able to detect an insecure HostNameVerifier that returns true.

However, due to the lack of comprehensive data flow analysis, Lint does not detect more complex instances of the above problem (cf. Figure 3.5).

```
final boolean isTesting = true;

HostnameVerifier hostnameVerifier = new HostnameVerifier() {
    @Override
    public boolean verify(String hostname, SSLSession session) {
        return isTesting;
    }
};
```

Figure 3.5: Android Lint fails to detect a simple insecure HostNameVerifier.

While we are aware that comprehensive data flow analysis is an ongoing branch of research in the field of static code analysis [122, 84], we feel that covering some more complex cases is crucial to provide a good user experience, since it confuses users to detect one instance of a problem but not another, more complex one.

Proposed Action: Improve data flow analysis to cover more complex cases

3.4.2 Approach

Given these limitations in Android Lint, we believe the tool has only limited impact in helping developers to improve app security. We hypothesized that an enhanced version might achieve better security results. To test this hypothesis, we implemented a further plug-in for Android Studio, tailored towards teaching developers about app security. We call this tool 'FixDroid'.

FixDroid addresses the Lint tool's limitations, and adds functionality to learn about developer behavior, while supporting developers in making security related decisions.

FixDroid aims to give its users unobtrusive feedback about the privacy and security impact of the code, as they write it. FixDroid scans a developers' code for 'pitfalls': constructs with less-than-ideal privacy and security. Additionally, FixDroid detects whenever a developer pastes a code snippet and attempts to match it against an online database of known insecure code snippets (from StackOverflow). FixDroid is available as an Intellij IDEA plugin for Android Studio. It had had more than 500 downloads by August 2017[2].

3.4.3 Addressed Pitfalls

FixDroid currently covers 13 security pitfalls taken from the Android Official documentation and from the existing research described in Section 3.10. It indicates these problems on the appropriate lines of code, using a 'security indicator' to catch developers' attention. For some of those pitfalls, FixDroid offers quick-fixes; when a quick-fix is not available, FixDroid provide a warning message that describes the pitfall. The list of addressed security pitfalls is in Table 3.1.

Pitfall	**Security Tooltip**	**Quick-fix**
Insecure `Cipher.getInstance`	You appear to be using `Cipher.getInstance` with the insecure default ECB Mode. To improve security, a different encryption mode with padding e.g. AES and CBC should be used.	AES/CBC/PKCS5Padding
Non-random Initial Vector for `Cipher.init`	You appear to use a constant Initial Vector. To secure the encrypted data against hacking attacks, the IV should be randomly generated and passed or stored along with the encrypted data.	
Constant key for encryption	You are using a constant key for encryption. To avoid an extraction the hard-coded key of the hard-coded key by reverse-engineering, a dynamically generated should be used, preferably from a server.	
Less than 1000 iterations for PBE	You are using less than 1000 iterations for PBE. It is recommended to use at least 1000 iterations to increase the difficulty of reversing the hash.	Use 1000 iterations
ECB mode for encryption	You appear to be using the insecure ECB mode for encryption. It is recommended to use a more secure mode like AES/CBC/PKCS5Padding.	AES/CBC/PKCS5Padding

[2]https://plugins.jetbrains.com/plugin/9497-fixdroid

Improper `HostNameVerifier`	You appear to be using an improper `HostNameVerifier`. This allows an attacker to impersonate the host. It is recommended to use default `HostNameVerifier` or, better still, SSL pinning.	
`SecureRandom` with static seed	You are using a static seed, which allows an attacker to predict the random numbers generated. It is recommended to use the default constructor of `SecureRandom`.	Remove static seed
HTTP over HTTPS	You are using an insecure HTTP connection. An attacker may intercept and view all the traffic, or replace the server completely. It is recommended to use HTTPS.	HTTPS upgrade
WebView HTTP over HTTPS	You are using an insecure HTTP connection. An attacker may intercept and view all the traffic, or replace the server completely. It is recommended to use HTTPS.	HTTPS upgrade
`WebView` Loading local HTML file	You are loading HTML content directly from the file system. A virus or rogue app running on the device might replace this with other code. It is recommended to load JavaScript only from secure areas.	
Custom certificate	This is a connection to a server with a self-signed/untrusted certificate. If you believe this server should be trusted, it is recommended to use SSL pinning.	SSL pinning
Loading code from public places	You are loading code from the publicly accessible location. This code can be infected from contact with a virus or rogue app running on the device. It is recommended to load code only from secure sources.	
SQL Injection	You are using a query that is vulnerable to SQL injection. An attacker can enter text that is interpreted as SQL commands, allowing access to the whole database. It is recommended to use a parameterized query.	Placeholder string

Table 3.1: Security tooltips and corresponding quick-fixes displayed by FixDroid.

3.4.4 How FixDroid Works

FixDroid leverages the inspecting mechanism in IntelliJ IDEA[3]. By default, FixDroid analyzes all open files of Java and Xml source code. It highlights all security bad practices as the developer writes code, using both Intellij's default highlighting and more visible 'security indicators' on the insecure code's line numbers. Furthermore, Intellij also supports developers running FixDroid inspection in bulk mode where all source files will be inspected: thus the developer can choose to inspect an entire project, or any scope within it.

When the developer moves the mouse over the highlighted code or over the security indicator, the corresponding warning message will be displayed. The developer can enable the available quick-fix by using the default short-cut of Android Studio or by simply clicking on the security indicator (cf. Figure 3.6).

3.4.5 Example of Use

Figures 3.6 through 3.8 show an Insecure Network Connection example. Here FixDroid observes that developers are writing code to connect to a given URL with the HTTP protocol – which is insecure. FixDroid finds a quick-fix using the same URL but replacing HTTP by HTTPS. Given this option is available, developers are informed by highlighting the insecure code and marking the corresponding code lines as insecure with a security warning icon. When developers move their mouse over the highlighted code or the warning icon, a corresponding message is shown, telling them what the problem is and how to resolve it. Developers can fix the insecure code by clicking on the warning icon or by using the built-in shortcut of Android Studio (cf. Figure 3.7). When a quick-fix is applied (cf. Figure 3.8), the previous warning message and security indicator disappear.

Figure 3.6: FixDroid detects an insecure code snippet.

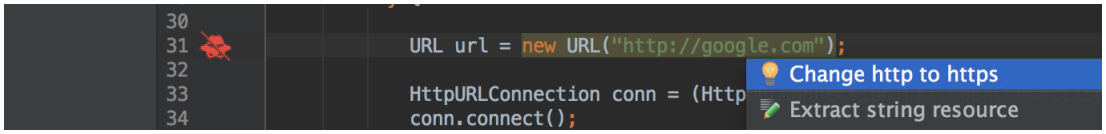

Figure 3.7: FixDroid suggests a quick fix.

[3] https://www.jetbrains.com/help/idea/running-inspections.html

```
30
31          URL url = new URL("https://google.com");
32
33          HttpURLConnection conn = (HttpURLConnection)url.openConnection();
34          conn.connect();
```

Figure 3.8: HTTPS Upgrade quick-fix has been applied.

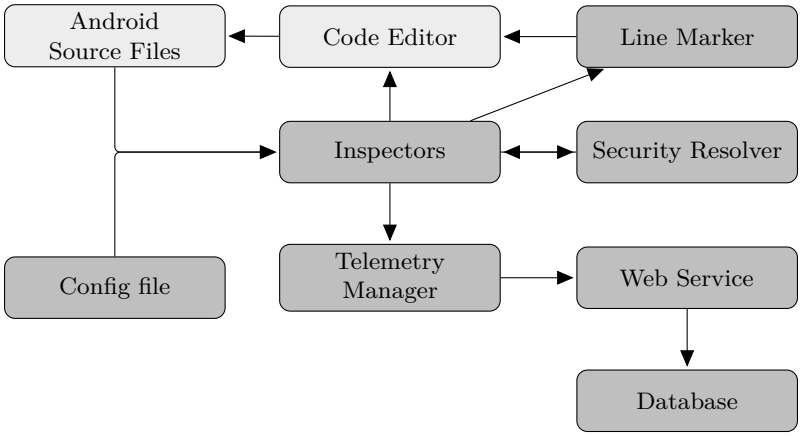

Figure 3.9: FixDroid's Architecture

3.4.6 FixDroid Implementation

The different components of FixDroid are illustrated in Figure 3.9. Inspectors are the center components that watch developer's code. Whenever the developer finishes writing a line of code, a method or a class implementation, the appropriate Inspector invokes Security Resolver to check if that given code snippet is secure or not. If the code snippet is insecure, the Inspector forwards the information to Telemetry Manager. At the same time the Inspector also informs developers via Code Editor by highlighting the insecure code snippet as well as marking the insecure code with a security indicator.

When the developer invokes a quick-fix, this invokes Line Marker to make the code change.

Web Service supports FixDroid's communication with our back-end database. Config File contains the mapping of which Inspector is responsible for which security pitfall.

3.4.7 Static code analysis

FixDroid leverages the IntelliJ IDEA static analysis techniques to performs static code analysis at method, class and project levels. Hence, FixDroid can statistically resolve variables that are computed from different code locations. This eliminates mistakes similar to the example of HostNameVerifier (c.f Section 3.4.1.2).

3.5 Pilot Study

In the developer review sections and the pilot study, we have the same 3 programming tasks: network connection, SQL query, and data encryption. They will be described in details in section 3.6.2

3.5.1 Initial User Interface Evaluation

First, we conducted three developer review sessions to gain a first insight into how developers might use FixDroid in real world situations. The reviews were conducted with three Android developers within the lead author's organization, CISPA. These three developers were asked to solve three programming tasks with FixDroid installed on their Android Studio. To closely observe the developers' interaction with FixDroid, a researcher sat beside them while they were solving the tasks. Developers expressed their feelings and expectations during the study.

From the reviews, we observed:

- All the observations of programmer behavior could be automated by the tool, so we did not need to invite future participants into our lab to watch them solve programming tasks. This could help avoid the biases that lab studies often face [75, 81].
- Highlighting insecure code is not enough. None of the three developers noticed the highlighted code.

Therefore as a next step we redesigned the field study to be conducted automatically and remotely. We invited later participants to join in our study online, by installing FixDroid over the web. We added functionality to gather and observe developers' interaction and send anonymous details to FixDroid's server. We also added an additional security indicator (cf. Figure 3.6 and 3.7) to inform developers of insecure practices.

3.5.2 Remote Pilot Study

We conducted a second pilot study with 9 participants, recruited from our industry contacts. All were experienced professional developers; Table 3.2 shows the participant demographics. A researcher stayed online (e.g., using Skype) to provide participants instruction and help when needed.

Age		
Mean = 26.11	Median = 26	Standard Deviation = 1.36
Professional Android Experience		
Yes = 5		No = 4

Table 3.2: Pilot study participant demographics.

All participants reported that they noticed the security indicator from FixDroid while only 3 participants noticed the highlighted code. This indicates the effectiveness of the

security indicator in informing developers about their insecure code snippet.

In this study, 6 out of 9 participants used quick-fixes provided by FixDroid. At this time FixDroid only provided participants with quick-fixes for the SQLite and Connection tasks; only one participant managed a secure solution for the Encryption task. That 8 out of 9 participants produced insecure code for the Encryption task although all of them had read FixDroid's warning messages, suggests that the cryptographic APIs in Android are particularly difficult for developers to use, even when they are aware of the security implications of their code. With that in mind, we decided to include a quick-fix for the Encryption task in our final study.

3.5.3 Online Developer Study Platform

After the online pilot study, we learned that, having a suitable infrastructure would allow us to conduct online developer study to evaluate the effect of FixDroid completely automatically. All study instructions and interaction between participant and executive researchers can be automated. This would eliminate the bias of in-lab study and enable us to conduct study with as many participants simultaneously as we would like. Therefore, to realize such infrastructure, we developed the following generic setup which can be easily integrated with any Android Studio plugin:

- A web service component that allows researchers to set up developer studies. Particularly, researchers can provide app skeletons for the programming tasks, instructions for participants, and a corresponding survey for the study.
- A client based (Android studio) component that retrieves from the web service information about the study, app skeleton, and setup the study, display instruction accordingly. More importantly, this component can have opt-in features that allows collect telemetric data in-the-wild on developer behavior, and send to the web service for later analysis.

In the context of this work, we leveraged such an infrastructure to evaluate the impact of FixDroid on the security of code provided by developers. In the following we describe how we apply the aforementioned infrastructure to conduct a remote study with developers in an automated fashion e.g., no executive researchers needed to moderate the study.

Study Guidelines We used the web service to ship a sample 'study project' and instructions to help developers learn how to use it. The sample project challenges developers to avoid many of the possible pitfalls, specifically those related to secure network connections, SQL injection, and encryption.

Telemetry We aimed to gain a better understanding of how developers interact with FixDroid. Therefore, we extended the client based component of our infrastructure to collect the following information:

- Source code: when participants completed a programming task in the study

project, we sent their completed implementation to our server[4].

- Security bad practice events, including the time, the type of bad practice (pitfall), and whether the code in question was copied/pasted. When FixDroid detects a pasted insecure code snippet, it asks the programmer for the source of that snippet, preferably as a URL
- Security good practice events, code that avoids a pitfall, to help measure if participants' security programming skills improve from using FixDroid.
- Security tooltip events record whether a particular warning message has been read by developers, how long developers spend to read it.
- Quick-fix events, indicating whether an offered quick-fix was used by developers, when it was used, and whether the developer used the default shortcut of Android Studio or clicked on security indicator. When the programmer applies a suggested quick-fix, FixDroid asks how useful they found the quick-fix, on a five point Likert scale.

3.6 User Study

3.6.1 Study Design

For our main study, we wanted to evaluate the effectiveness of the FixDroid approach with professional Android developers. We therefore recruited Android developers who submitted apps to the Google Play store.

Our hypothesis H_1 was that developers using FixDroid would deliver more secure Android code; the corresponding null hypothesis H_0 that they would not. We therefore divided participants into two groups: developers in one group had all the functionality of FixDroid (FixDroid enabled); developers in the other did not have FixDroid fully enabled (no warning messages or quick-fixes). Both groups had the Lint tool enabled. To balance the group sizes the FixDroid server assigned participants based on the number of valid participants so far received in each group.

Each participant carried out the 'study project'. Our analysis only considers participants who completed writing code for at least 1 code snippet and filled out our exit survey.

3.6.2 Study Tasks

In the 'study project', participants were provided a skeleton Android application and asked to solve three different security related programming tasks. Each participant received the same three programming tasks, although the task ordering was randomized. For each task, a corresponding unit test was provided that participants could run to check if their solution is functional. Note that the test cases do not check if the solution is secure.

[4]We only sent the source code that belongs to our study. If participants used FixDroid in other projects, the corresponding source code was not collected.

The following sections describe the three tasks. While these do not encompass the entire space of security relevant problems encountered by Android developers, previous studies [4, 6] have found that even simple problems similar or identical to those used in the study lead to problems in production code.

Network Connection This provides a code snippet to be completed to establish a connection to a server. Participants were given the domain and query path of the URL, then requested to add a protocol to make this URL valid and establish the connection and get a return code status (i.e 200, 400 or 500). The goal of this task was to check if participants used a secure connection (HTTPS) to connect to the given host. The host supported both HTTP and HTTPS protocols.

A corresponding test case was provided, which passes when the method connect returns the value of 200.

Data Encryption In this task, participants were asked to encrypt a given plain-text. Participants were expected to encrypt a string and return an array of bytes. The goal of this task is to see how knowledgeable developers are about cryptographic APIs.

A corresponding test case was provided, which passes when the returned array of `encryption` method is not null and has a length greater than 16.

SQLite Query In this task, participants were asked to build a SQL query to retrieve the age of a given user name. In the skeleton app, we have already created a SQLite database with a predefined table named "users". Table "users" has 4 columns: id, name, age, password. The goal of this task is to see if participants are aware of SQL injection attacks.

A corresponding test case was provided, which passes when this test case returns the correct age of a predefined user which has been inserted into the attached SQLite database.

3.6.3 Participant Journey

After each participant finished installing FixDroid and its dependencies[5], the installer requested a restart of their Android Studio to commission the newly installed plugin (FixDroid). FixDroid then offered the participant to join our Android Research Study with a reminder that all of the listed dependencies must be installed in advance.

If the participant decided to join our study, FixDroid then provided instructions on how to navigate between tasks, test cases and how to reset a solution back to the original version of the code; then invited them to click start to get the first task. Every time the participant built the Android application, the solution was sent to FixDroid's web service for later analysis.

[5]SDK Platform Android 7, Android SDK build-tools

Since the second group did not have FixDroid enabled for the study, when a participant in this group had filled in our exit survey, FixDroid then enabled its full functionality. It also opened a file containing example insecure code, allowing the participant to see how the full-functionality FixDroid worked by examining the warning message or applying the provided quick-fix. Thus each participant in the second group received the benefit of the FixDroid tool for later use without a biasing effect on our study results.

3.6.4 Exit Survey

While a participant was still completing the tasks, FixDroid showed an "Open Survey" button, encouraging participation in our the survey. When a participant had written code that passed all of the three test cases, FixDroid prompted explicitly for participation in the survey. FixDroid also asked for survey participation when a participant closed or quitted Android Studio without completing all the tasks.

For the group with FixDroid enabled the survey included questions about the participant's interaction with FixDroid together with demographic questions. For the group without FixDroid, the survey asked only demographic questions. Appendix Section A.1 has details of the questions.

3.6.5 Evaluating Participant Solutions

We invoked each sample of code submitted by our participants, built into a suitable framework. For each task, we also manually evaluated their security and functional correctness, creating a score to reflect each outcome based on the properties of each task. Two researchers were assigned to score the participants' solutions, with a third coder providing a casting vote in cases of disagreement. The scores were assigned as follows:

Functionality For each programming task, a participant received score 1 for their functionality if the code passed the test, otherwise 0 is given.

Security Only functional solutions received a security score. Depending on the task, we evaluated different security considerations, coding each as secure (1) or not (0) as follows.

3.6.6 Security Evaluation

For the Connection task, if a participant used `https` protocol for their connection, their solution was considered secure; `http` was considered insecure.

```
1 String query = "select age from users where name = ?";
2 Cursor cursor = database.rawQuery(query,new
3 //or
4 Cursor cursor2 = database.query("users",new
    String[]{"age"},"name=?",new String[]{userName}, null,
    null, null);
```

Listing 3.1: Parameterized query string

For the SQLite task, if a participant used question mark `?` as string placeholder and put `userName` as parameter of their `rawQuery` method call; or if they used the `query` method specifying the column's name, table's name, and arguments as parameters, the solutions was considered secure (see Listing 3.1 for example). However, if a participant concatenated the variable `userName` to their query string, the solutions was coded as insecure.

For the Encryption task, we captured how different parameters affect a solution's security as described in Table 3.3; that table only lists options that were found in one or more participants' solutions.

Parameter	**Secure**	**Insecure**
Cipher/Mode	AES/CBC [152] AES/GCM [137] AES/CFB [137]	DES, AES/ECB [52] Blowfish [151]
Initialization Vector	provider generated	static [52] bad derivation[52]
Key	provider generated	static [52] bad derivation [52]
Password Based Encryption	$\geqslant$ 1000 iterations [93] $\geqslant$ 64-bit salt [93] non-static salt [93]	$<$ 1000 iterations [93] $<$ 64-bit salt[93] static salt [52]

Table 3.3: Encryption security parameters

3.6.7 Recruitment

To maintain the validity of our study, we wanted only to recruit experienced Android developers. Therefore, we extracted developers' emails from the Google Play Store, since any such developer will have completed at least one working app. We sent emails in batches, asking Google Play developers to participate in a study on how to support Android developers in writing code. We did not mention security in the recruitment email. However, this approach did not scale well since participants wanted to know what FixDroid does before installing it as an Android Studio plugin. We received a number of emails asking for this information. We also provided participants the option to stop receiving invitation emails from us.

To encourage more participants, we added more details to our later emails, specifying that FixDroid helps developers write more secure code with possible quick-fixes. After this change we had a higher response rate. In all, we sent invitation emails to 210,854 developers and got 16 participants who volunteered to participate in our study and finished both writing code and filling in our exit survey.

We also recruited students to join in our study. As compensation they received 25 Euros either in cash or as an Amazon voucher. We sent invitation emails to five universities in Germany.

Our email linked to the FixDroid plugin in the Android Studio plugins repository, allowing participants to download FixDroid directly from their Android Studio.

To verify students' Android programming experience we gave them a set of 5 Android programming related questions to answer. We only invited students who answered at least 3 questions correctly. 65 students participated in our pre-study quiz; 59 were invited. We stopped recruiting when we had 24 students, due to budget constraints.

3.6.8 Ethical Concerns

All the telemetry data was gathered pseudonymously, with personally identifiable information removed before sending it to the server. All data was sent securely to FixDroid's web service using HTTPS. Our study was approved by our institution's ethics review board.

We have concerns about researchers sending emails to large numbers of developers, and are working with StackOverflow to deliver an opt-in list for developers interested in working with academic researchers.

3.7 Study Results

3.7.1 Participants

In total, 409 participants downloaded FixDroid. Table 3.4 shows how many from each group completed the exit survey. This includes some participants who did not write any code.

Mode	Started	Completed
Full functionality	45	22
Only telemetry	70	35

Table 3.4: Number of participants who started and completed surveys

Table 3.5 shows how many completed 1, 2, or 3 tasks, and how many dropped out before completing any tasks. This includes all participants, including those who didn't complete a task or the survey.

Mode	0 Task	1 Task	2 Tasks	3 Tasks
Full functionality	33	9	3	19
Only telemetry	55	5	3	19

Table 3.5: Number of participants who completed tasks

	Full functionality	Only Telemetry
Age		
Mean	25.32	27.45
Median	25.00	25.00
Standard Deviation	4.60	5.88
Information Security Background		
Yes	7	12
No	6	14
Apps Submitted		
Mean	4.89	6.95
Median	3.00	3.00
Standard Deviation	4.42	7.05

Table 3.6: Participant Background

Our participants in the developers group were aged between 21 and 47 (see Table 3.6) while participants in students groups were aged between 19 and 30.

Country	Count	Country	Count
Moldova	1	Poland	1
UK	1	Colombia	1
Germany	3	India	3
Vietnam	2	Turkey	1
Czech Republic	1	Greece	1
Kenya	1		

Table 3.7: Country of origin of developers

All the students come from Germany; the developers included participants from nearly all over the world (Table 3.7).

Almost all participants have been programming in Android for at least 6 months (see Figure 3.10). The exception was two students who had only taken Android programming related courses.

3.7.2 Findings from Participants' Experience

This section explores our findings from the exit survey, telemetry features and experience sampling. This analysis considers only participants who completed both at least one task and the exit survey.

Sources of Information Figure 3.11 shows participants' descriptions of where they looked for coding support. The results are consistent with Acar et al.'s earlier research [6], suggesting that these developers are typical in their use of development resources.

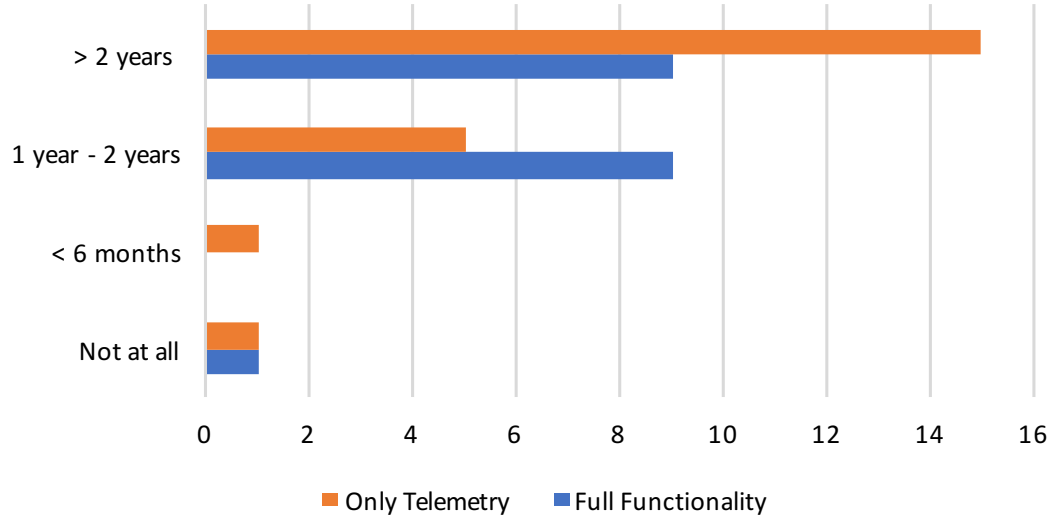

Figure 3.10: Android programming experience

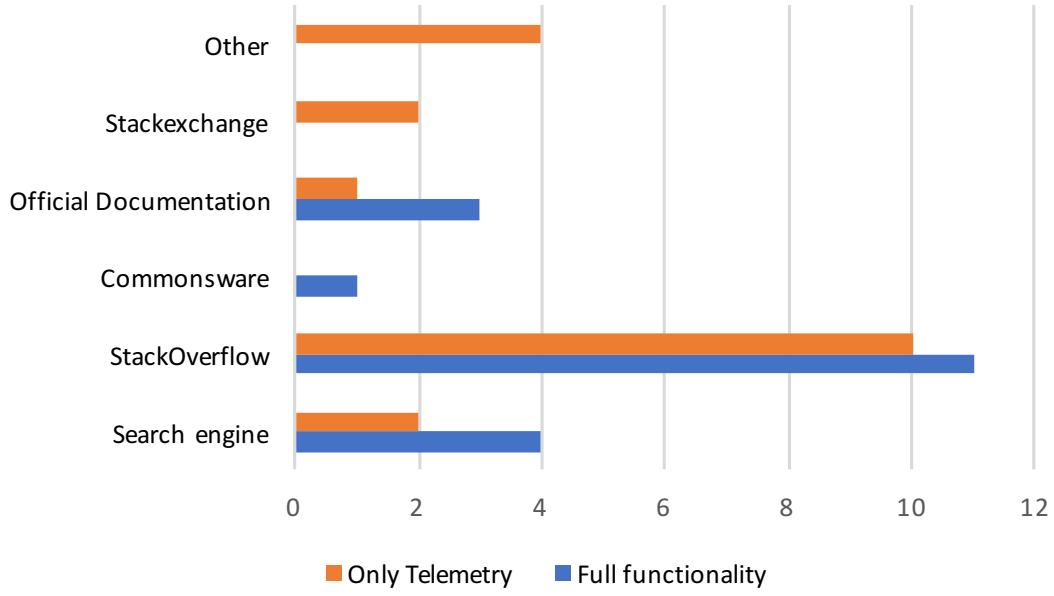

Figure 3.11: Where do you usually look for security related coding questions?

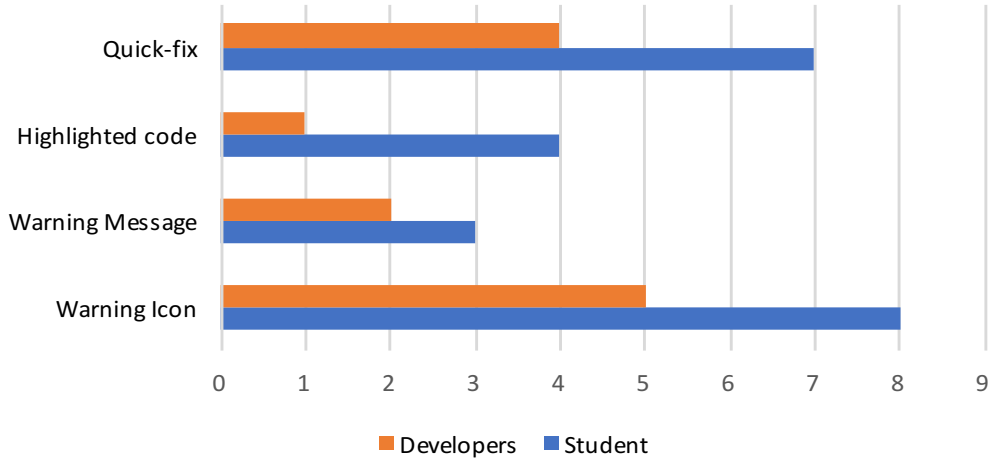

Figure 3.12: FixDroid features reported by participants

Perceived use of FixDroid features Moving on to consider survey information from the participants using FixDroid, Figure 3.12 shows which features of FixDroid users believed they used. Consistently the students used more features; and the warning icon and quick-fix were used most. This is consistent with the telemetry recorded by FixDroid (see Section 3.7.2)

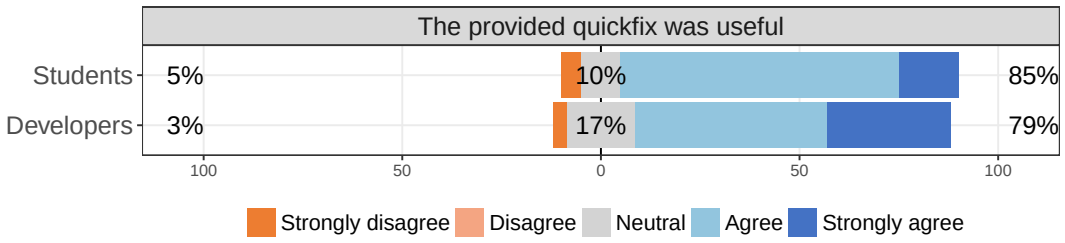

Figure 3.13: Reported value of each quickfix applied

Perceived value of quick fixes Every time participants used a quick-fix offered by FixDroid, they were immediately presented with a question asking for the usefulness of the provided quick-fix, using a 5-point Likert scale ranging from "Strongly Disagree" to "Strongly Agree". Figure 3.13 summarises the responses; a majority of quick-fix users agreed or strongly agreed that their provided quick-fixes were useful.

On the exit survey, nearly two third of the participants (63.15%) in the FixDroid group reported that they used at least one provided quick-fix; they all reported that the provided quick-fix was useful (n=19). Interestingly only half of participants in this group reported having used IDE-provided quick-fixes prior to our study, although quick-fixes are generally available for non-security related issues.

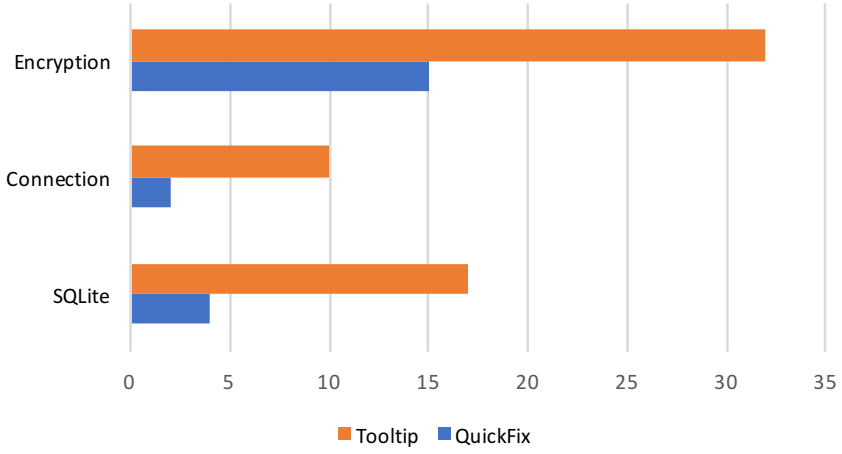

Figure 3.14: Actual use of FixDroid features

Actual Use of FixDroid Features Figure 3.14 shows the actual use of FixDroid's features during each of the tasks, as measured by FixDroid's telemetry functionality. The Encryption task generated significantly more activities than the other two, suggesting that that this is particularly difficult for developers.

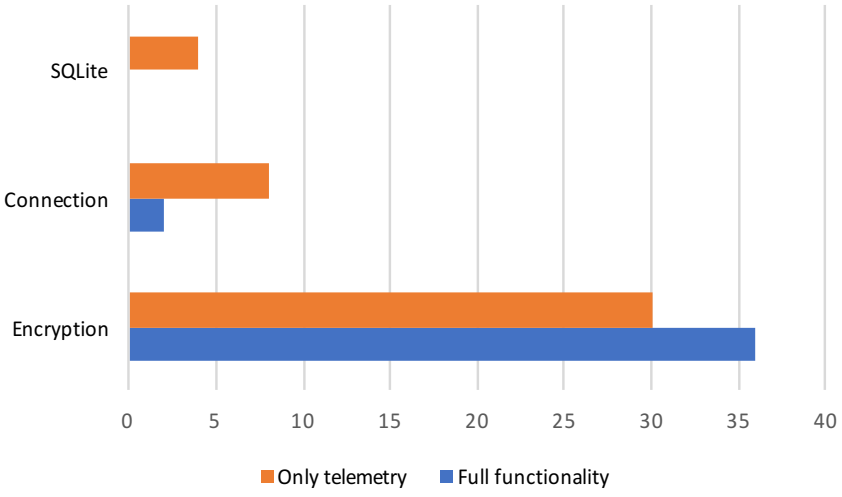

Figure 3.15: Number of copied and pasted insecure code events

Use of copy/paste Figure 3.15 shows the developer use of copy/paste during each of the tasks. Again, the Encryption task generated significantly than the other two, suggesting that it required much more online research on the part of developers.

Whenever participants copied and pasted an insecure code snippet to their solution, FixDroid asked them to provide us the URL of the website from which they copied

the code. The most common source was StackOverflow; this supports the reported information sources in our exit survey. We manually examined the links, and found that, for encryption related questions, every link contained at least one insecure code snippet.

Other participants reported that they copied code from other projects or from their notepad.

Factor	Description	Baseline
Required factors		
Mode	FixDroid or Default	Default (only telemetry feature)
Task	One of the three tasks described in Section 3.6.2	Connection
Optional factors		
Group	Developer or student	Developer
Experienced	True if participant has submitted more than 5 apps or has more than 3 years of experience otherwise False. Self-reported.	False
Security background	True or False, self reported	False

Table 3.8: Factors used in regression analysis

3.7.3 Regression Model

The following sections show the results of applying a regression model to analyze the results in detail.

Table 3.8 shows the factors analyzed. Since we are only interested in binary outcomes (e.g., secure vs. insecure), we used logistic regression. When we considered results on a per-task rather than a per-participant basis, we used a mixed model that adds a random intercept to account for multiple tasks from the same participant.

For the regression analysis, we considered a set of candidate models and selected the model with the lowest Akaike Information Criterion (AIC) [30]. The included factors are described in Table 3.8. We considered candidate models consisting of the required factors *mode* and *task* , the participant random intercept, plus every possible combination of the optional variables.

We report the outcome of our regressions in Table 3.9. Each row measures the change in the analyzed outcome related to changing from the *baseline* value for a given factor to a different value for that factor (such as changing from not having access to FixDroid functionality to having FixDroid activated). Logistic regressions produce an odds ratio (O.R.) that measures the change in likelihood of the targeted outcome; baseline factors by construction have O.R.=1. In each row, we also give a 95% confidence interval (C.I.) and a p-value indicating statistical significance.

Factor	O.R.	C.I.	p.value
FixDroid	19.55	[2.42, 157.84]	0.005*
Encryption	0.37	[0.13, 1.06]	0.065
SQLite	0.99	[0.34, 2.88]	0.993

Table 3.9: Mixed logistic regression on factors contributing to task security

For the regression, we set using normal Android Studio (only with FixDroid telemetry features) as the baseline. In addition, we used the connection task as the baseline, as this seems like a likely task for developers to encounter in real life and has been done before by Acar et al. [6]. All baseline values are given in Table 3.8.

3.7.4 Functional Correctness Results

We observed no statistically significant difference in the number of functional solutions between each task, and between groups (cf. Figure 3.16). Developers and students with FixDroid's support do not perform significantly better compared to participants with only the telemetry features, in terms of functional correctness.

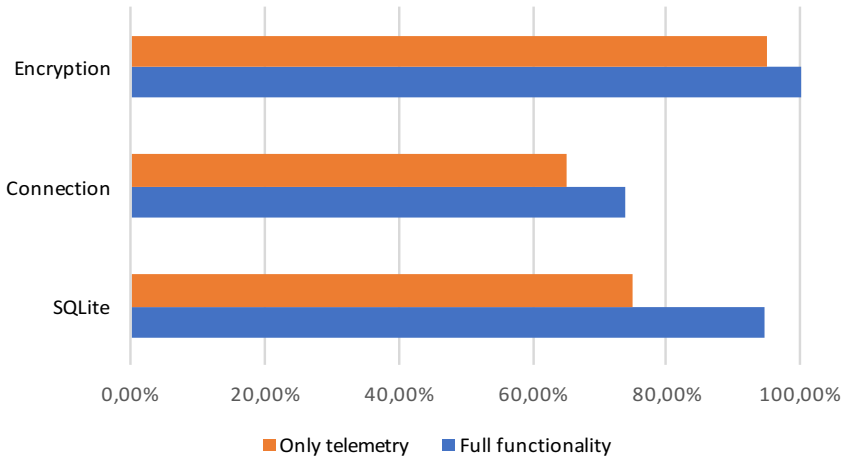

Figure 3.16: Funtionality results for each task

3.7.5 Security Correctness Results

Figure 3.17 shows the proportion of developers and students achieving a secure solution in each of the tasks. We were pleased to see a dramatic difference in security success, with more than twice the proportion achieving a secure solution in the Encryption and Connection tasks. Interestingly there was little difference in security success for the SQLite task.

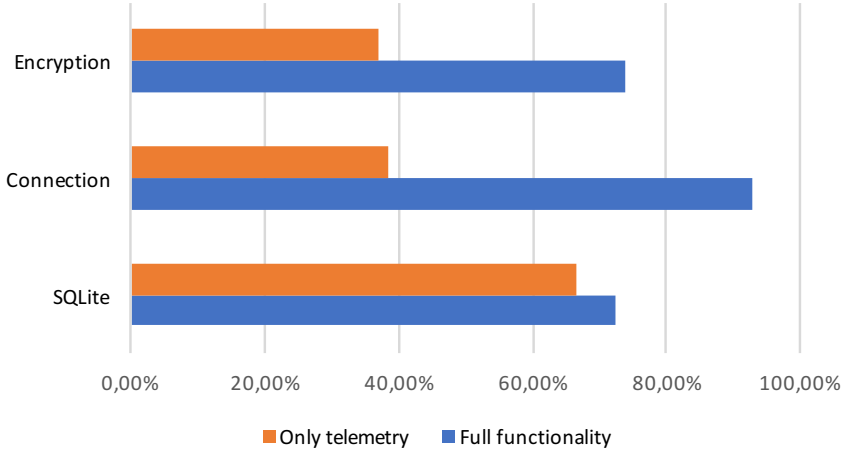

Figure 3.17: Security results for each task

Table 3.9 shows the analysis for these figures, considering factors contributing to task security, performed on functionally correct tasks. Statistically significant values are indicated with *.

Since we used the group of participants with no security quick-fixes or warning messages available as our baseline, our hypothesis H1 (that FixDroid has an impact on security) is represented by the factor 'FixDroid' in the analysis. We can therefore reject the null hypothesis, H0, that FixDroid has no impact, with the p value of 0.005.

3.8 Limitations

We can identify the following limitations to our sampling process. First, since the participants are in effect self-selecting, we have an opt-in bias. There is the possibility that these results won't extend to the full set of programmers. We also have a relatively small sample set, though we have addressed that using appropriate statistical methods.

As we wanted to have more participants, we had to compensate students to join our study while professional developers were not paid. However, we observed only one significant difference: professional developers were more likely to drop out.

There is an issue with acquiescence bias or the Hawthorne effect, especially in the reported value of quick-fixes in section 3.7.2: participants are more likely to report liking a new system.

As we describe in Section 3.6.7, we had to briefly mention what FixDroid does in order to recruit more professional developers. This could possibly introduce bias into our study. It is not an easy task to convince developers to install a third party plugin without telling them what it does.

Though we believe it is important, we did not focus on improving data flow analysis

except for leveraging the existing features of IntelliJ IDEA previously ignored by Lint (see Section 3.4.7).

3.9 Future Work

We have identified several areas for further work:

- FixDroid provides a good platform for the analysis of programmer coding behaviour with regards to security. Clearly there is scope for exercises covering further kinds of defect and using FixDroid to analyse how developers address them.
- Improved data flow analysis will help developers detect more kinds of defect; work on this will improve the usefulness of FixDroid app still more.
- FixDroid continuously measures security good practices and security bad practices. This has the potential to support a future longitudinal study to see if people actually use FixDroid and get better at writing secure code over time.
- Further, and more ambitiously, FixDroid telemetry feedback has the potential to support machine learning to start to identify software security 'smells' and provide more sophisticated analysis of developer code.

3.10 Related Work

We found related work in two key areas: investigations of security issues in Android development, and studies of tools that support developers in writing code.

3.10.1 Security Issues in Android Development

Several research teams have used analysis tools to investigate Android app security. With CryptoLint, a lightweight static analysis tool, Egele et al. [52] showed that 88% of Android applications using cryptographic APIs include at least one mistake. Balebako et al. [21] found that developers can make these mistakes due to lack of security knowledge;Georgiev et al. [66] also identified bad API implementations as a cause. Fahl et al. [59] implemented MalloDroid, a static code analysis tool that detects potential vulnerabilities against SSL Man-In-The-Middle (MITM) attacks in Android and iOS applications, and found that many developers accept insecure practices (such as SSL certificate validation that accepts all certificates) to achieve functional code.

Poeplau et al. [123] investigated dynamic code loading in Android applications, using a static code analysis tool. Their results revealed that many applications load additional code in insecure ways.

The integration of web content into mobile apps also exposes Android applications to multiple types of attacks [114, 108]. Wang et al. [154] studied the cross origin risks inherent in mobile applications and found that lack of origin-based protection enables many types of cross-origin attacks. Luo et al. [106] also demonstrated different attacks on benign Android and iOS applications that misuse webview customization.

Felt et al. [125] investigated app permissions, and identified several reasons why developers tend to request more permissions than their apps actually need, including insufficient API documentation, confusing permission names, copy and paste code snippets, and testing artifacts. In another investigation concerning 'permission re-delegation', Felt et al. [128] concluded that not all developers are security experts and they are not motivated enough to prevent permission re-delegation because the consequences do not affect their applications directly.

Acar et al. [6] examined the impact of the information sources used by Android developers on their security related decisions, and found that developers often use informal sources such as StackOverflow, resulting in functional code but often also vulnerabilities in their apps.

3.10.2 Tools that Support Developers

Kim et al. [94] ethnographically studied copy and paste (C&P) programming practices in object oriented programming languages by observing programmers using an instrumented Eclipse IDE, and proposed a set of tools to reduce software maintenance problems incurred by C&P and support the intents of common C&P situations.

Several research teams have developed tools that support secure coding, typically focusing on finding application vulnerabilities after the program has been written. This results in these tools finding vulnerabilities at the end of the development cycle [142, 105]. Furthermore, though valuable, these tools all have one thing in common: developers need to have certain levels of security expertise to use them. Chin et al. [39] proposed platform-level, API-level, and design-level solutions to help developers and platform designers build secure applications and systems. They also developed ComDroid [42] to detect and warn developers of exploitable inter-application communication errors. However, ComDroid works only on compiled code and can thus not help developers while they are writing source code. Jovanovic at el. developed Pixy to help developers avoid cross-site scripting vulnerabilities in PHP scripts [92]. Pixy uses flow-sensitive, inter-procedural and context-sensitive data flow analysis to discover vulnerable points in a web application, but provides no IDE-based feedback to developers. Recently, Tabassum at el. conducted a study comparing the effect of secure programming tool support (ESIDE) versus teaching assistants [107]. The results showed that ESIDE provided more insights to students about the security flaws. Tyler et al. [147] examined how developers understand the support of an interactive static analysis tool using a plugin for Eclipse that helps web developers detect and mitigate security vulnerabilities as they write code.

None of this work, however, has investigated providing feedback on code security to Android developers as they write their code. This work aims to fill this gap.

3.11 Conclusion

This work explored the possibility of supporting Android developers to write secure code. Section 3.4.1 showed the limitations of the existing Android Lint tool, and suggests

improvements. These improvements motivated the creation of a new IDE plug-in, FixDroid, as described in Section 3.4.

A series of studies evaluated this approach, as described in Section 3.6, and analyzed in Section 3.7. Early studies discovered the importance of a more visible signal of security issues than the existing Lint indicators. The later studies also validated the approach of using telematics in an IDE to determine programmer behavior.

Finally, the studies conclusively proved the effectiveness of such a tool in improving the security of code produced.

These findings suggest that it will significantly improve the security of developed apps if future Android IDEs contain functionality similar to the FixDroid tool, with a clear indication of security errors and offers of security 'quick-fixes'.

We conclude that, to improve app security, it's vital that future versions of the Android development environments incorporate similar features.

4

Android Tool Support to Fix Insecure Code Dependencies

4.1 Motivation

Third-party libraries, especially outdated versions, can introduce and multiply security & privacy related issues to Android applications. While prior work has shown the need for tool support for developers to avoid libraries with security problems, no such a solution has yet been brought forward to Android. It is unclear how such a solution would work and which challenges need to be solved in realizing it.

In this work, we want to make a step forward in this direction. We propose Up2Dep, an Android Studio extension that supports Android developers in keeping project dependencies up-to-date and in avoiding insecure libraries. To evaluate the technical feasibility of Up2Dep, we publicly released Up2Dep and tested it with Android developers (N=56) in their daily tasks. Up2Dep has delivered quick-fixes that mitigate 108 outdated dependencies and 8 outdated dependencies with security problems in 34 *real* projects. It was perceived by those developers as being helpful. Our results also highlight technical challenges in realizing such support, for which we provide solutions and new insights.

Our results emphasize the urgent need for designated tool support to detect and update insecure outdated third-party libraries in Android apps. We believe that Up2Dep has provided a valuable step forward to improving the security of the Android ecosystem and encouraging results for tool support with a tangible impact as app developers have an easy means to fix their outdated and insecure dependencies.

4.2 Problem Description

Software developers commonly re-use existing code, in particular in the form of third-party libraries. Third-party libraries are software components that are bundled in a form that can be distributed to developers through different channels, such as central repositories. However, those libraries come at a cost: if they contain bugs or security and privacy issues, those flaws could be amplified by being integrated in different applications that use the affected library versions. Prior works [26, 56, 49, 113] showed that such libraries could expose user sensitive information to third-party applications, or be a contributing factor for cryptographic API misuse in applications. More concerning, even when privacy & security related fixes were available in newer versions of affected libraries, their adoption by developers progressed very slowly [19]. Existing work has proposed different solutions to overcome the problem of outdated third-party libraries. Ogawa et al. [117] proposed using an external service to split app code and third-party library code from Android application package (APK) files, and then replace the (vulnerable) outdated libraries with their fixed versions. This might improve the situation but requires user actions to re-install the updated APK. Market stores may play the central role to roll out updates for libraries to end users but third-party libraries are tightly integrated into their host app which makes it virtually impossible to precisely separate app code and library code [19], and to pinpoint the exact version of a library that an app

is using. App developers (*not* market stores, *not* end users) are in the perfect position to fix this problem in today's ecosystem as app code and library code are separated in their development environment. They are aware of the exact library version they are using and can upgrade their dependencies. However, developers do not update their app's dependencies [49] because the outdated libraries are still working; developers fear incompatibilities between library versions, specifically semantic versioning has been found unreliable and has failed developers; developers are unaware of the updates; and updates take too much effort. This raises an important issue: providing security updates does not solve the problems at all if developers do not migrate their app's dependencies to the security fixes.

Unfortunately, existing solutions to support developers in keeping their project's dependencies up-to-date are ineffective. Android Studio itself includes *Lint* tool [102] to inform developers about updates of third-party dependencies in Android projects. However, *Lint* only provides developers information on whether there are newer versions of the included libraries, but it does not alert developers about security vulnerabilities of the libraries and it is limited to a list of only two libraries that are classified as privacy risks and without further information about the risks. The lack of information on whether the current version is secure and on whether the newer version is compatible with the existing code of the app makes developers afraid of migrating their project dependencies to newer versions, and, more importantly, it accustoms developers to stay with the current (although outdated and more likely vulnerable) versions as long as the app is still working.

With established tools being unreliable and not universally adopted (semantic versioning) [49] or providing insufficient support (*Lint*), this leaves a gap between developers' expressed wishes for support and the status quo. At the same time, there are still open questions on the technical feasibility of tool support for developers to keep project's dependencies up-to-date, which challenges come along the way, especially how developers would receive (i.e., apply) such a tool support, and most importantly its (security) impact on real projects. As long as these questions are not yet answered, we will still see security & privacy problems in apps that are attributed to outdated, insecure third-party libraries.

4.3 Contribution

To make advances in filling the aforementioned gap with appropriate tool support for app developers, we focus in this work on the following research questions:

- *"Would it be technically feasible to support developers in keeping their project's dependencies up-to-date?"*
- and, more importantly, *"Could such a tool support have a tangible impact on the security and privacy of Android apps?"*.

To try to answer these questions, we developed an Android Studio extension called Up2Dep to help Android app developers in upgrading their library dependencies and in

avoiding vulnerable library versions. Up2Dep analyzes third-party libraries to provide developers with information about the changes that they may need to perform when updating a library, based on the public API changes between the library versions.

Using the collected information about library APIs and their usages on a given project, Up2Dep provides developers feedback on the updatability of outdated library versions (i.e., we base our updatability information only on the code itself and not on unreliable external sources like semantic version of libraries).

Up2Dep also maintains a database of publicly disclosed vulnerabilities and cryptographic API misuse of libraries, and alerts developers if a vulnerable library version is included in their apps.

Our solution is the first implemented solution to support app developers in their task to avoid outdated, critical dependencies, and an important step to gather first-hand feedback from developers about solutions that so far have only been recommended in the literature.

We tested Up2Dep with developers to see how Up2Dep can support them in their daily programming tasks. To measure the impact of Up2Dep, we implemented telemetric features inside Up2Dep that gather anonymized information on how developers interact with Up2Dep and that allow developers to provide feedback in-situ on whether the suggested quick-fixes worked as they expected and what they think about such support from Up2Dep. Our telemetric data shows that among 56 developers, 30 have applied quick-fixes suggested by Up2Dep on 34 real projects, totaling 116 quick-fixes (8 insecure library versions, 108 outdated library versions). The results from the 60 in-situ feedbacks we received from 22 developers confirm that 80.0% of the proposed fixes worked and Up2Dep's support was useful, while only four cases of the proposed fixes did not work. Upon further investigating the feedback, we discovered that 13.51% libraries in our dataset have hidden security related problems as the problems reside in the transitive dependencies of those libraries, and are not shown to the developers. We believe, this is a new and important finding because if this problem is not solved, many app developers would continue using insecure libraries without being aware of it. This is detrimental for the security of the Android ecosystem as end users of such apps will eventually be exposed to a variety of attacks. Therefore, we subsequently developed a solution to tackle this problem by analyzing and alerting developers of libraries that have (hidden) transitive dependencies with security problems. Further, our study results show that having tool support on the compatibility of the updates really helps developers more willing to keep their project's dependencies up-to-date. Lastly, we further evaluated developers' Up2Dep experience in an online survey where 23 developers shared with us their opinion. Up2Dep received a SUS score [28] of 76.20, which indicates that Up2Dep was considered good by developers in terms of usability.

In summary, we make the following tangible contributions:

- We significantly extended *LibScout*'s original library dataset (by the factor of 7.5x, totaling 1,878 libraries with their complete version history) and analyze those libraries (37,402 library versions) to discover cryptographic API misuse.

To support future research, we make both Up2Dep and this dataset publicly available [1].

- We built an Android Studio extension called Up2Dep to warn developers about vulnerable library versions including both publicly disclosed vulnerabilities and cryptographic API misuse. Up2Dep helps developers upgrade their project's dependencies, taking into account the library API compatibility.
- We evaluated the technical feasibility of Up2Dep with Android developers (N=56) in-the-wild and gather anonymized usage information with our telemetric features. Our results show that Up2Dep has helped developers in fixing their project dependencies (n=108) and in avoiding dependencies with security problems (n=8). The majority (80.0%) of suggested fixes (from developer's feedback) worked and developers found them useful, while only four instances of the proposed fixes did not work as developers expected.
- We discovered that 13.51% of the libraries (233 out of 1,725) have hidden security problems by including (insecure) dependencies which is normally not visible to developers. We have subsequently developed a solution to tackle this problem.
- Our results show that developers indeed are in favor of such support and are willing to use it in their projects. Thus, this work makes a call for action to include such an IDE-provided support for app developers to avoid insecure code dependencies already during app development and for the research community to further investigate how library updatability can be further improved (e.g., detecting non-code, breaking changes between library versions).

4.4 Technical Description and Approach

4.4.1 Gradle Build Tool in Android Studio

Android Studio uses Gradle Build Tool [80] as an Android Studio plugin to automate and to manage the app build process. The Gradle build system eases the task of including internal and/or external libraries to app builds as dependencies. In our work, we do not take into account local binary dependencies, e.g., jar files that developers manually download and import into their projects because the majority of third-party libraries are included in Android projects via central repositories. Besides, for local module dependencies and local binary dependencies, the exact version information is not available, one can only profile the binary files and provide approximate matches which would add up another factor of uncertainty.

Listing 4.1 shows examples of how developers can declare their project's external dependencies in Android Studio. On line 3, components of a dependency's information are colon-separated, *group_id:artifact_id:version*, while on line 5, they are declared as key-values. From this information, when developers choose to sync their project's dependencies, *Gradle* will sync such dependencies from the default repository (e.g.,

[1] https://github.com/ngcuongst/up2dep

```
1 ext.supportVersion = 25.3.1
2 dependencies {
3  implementation 'com.example:magic:1.2.1'
4   //or
5  implementation group: 'com.example', name: 'magic', version: '1.2.2'
6  //dependencies use variable as version string
7  implementation 'com.android.support: support-v4:$supportVersion'
8  implementation 'com.android.support: appcompbat-v7:$supportVersion'
9 }
```

Listing 4.1: Declaring external dependencies in Android projects.

JCenter or Maven) or the ones declared in the *gradle.settings* file of the app project. Besides, developers can also declare version strings as a variable (line 1) and use this variable for the external dependency's version (lines 7,8). This helps developers avoiding repeatedly specifying (and updating) version strings for multiple libraries from the same group (e.g., *com.android.support*) that use the same version string.

4.4.2 Up2Dep Design

In this work, we propose Up2Dep, an Android Studio extension that facilitates the task of keeping Android project dependencies up-to-date, and help developers avoid insecure library versions. We focus on the Android Studio IDE as this is the tool officially supported by Google and a previous survey [49] has shown that most Android developers use it to develop apps. We abstain from performing automatic (updates) patching in the background because this is too much control over developer's source code. Further, it is not possible (with absolute reliability) to guarantee that the patching is free of unintended side effects. Additionally, developers should be informed and in control of the changes on their projects.

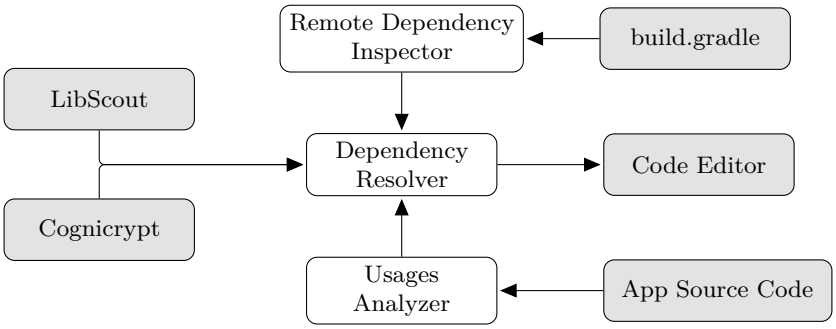

Figure 4.1: Up2Dep's architecture. Gray boxes are external components

Up2Dep analyzes the developer's code and provides developers information about the compatibility of the dependency's update. In case an update to the latest version is incompatible, developers are provided with two options: either they can update to the latest *compatible* version without having to adjust their app's code; or they can update to the latest version and Up2Dep provides them with information about which library APIs

have changed and recommends changes to their existing app code. Additionally, Up2Dep leverages information about publicly disclosed vulnerabilities of libraries and detected cryptographic misuse in Android third-party libraries to warn developers against using insecure versions of dependencies. Figure 4.1 illustrates how the different components of Up2Dep interact with each other. Using *LibScout* [101] and *Cognicrypt* [96], we feed Up2Dep the pre-analysis results consisting of API dependency analysis (from *LibScout*) and cryptographic API misuse analysis (from *Cognicrypt*). These pre-analyses results are bundled into offline databases. This allows Up2Dep to provide developers real time feedback as it does not need to repeatedly analyze all version history of the third-party libraries that developers include into their applications, which might incur unnecessary performance overhead. More importantly, Up2Dep does not need to send developer's code or all library information to its server as this would potentially threaten the privacy of developers and their code. After developers open a project in their Android Studio, *Usages Analyzer* will read the *Android Source Code* to analyze it for usages of the included third-party libraries. Whenever developers open a gradle build file (i.e., where dependencies are specified, see Section 4.4.1), *Remote Dependency Inspector* will run its inspection to check for outdated library versions. Finally, *Dependency Resolver* takes the results of *Dependency Inspector* and *Usages Analyzer* to compare them against the pre-analysis results to gather the following information:

- Are there newer versions of the included library?
- To which extent can the included library be upgraded, e.g, is there any incompatibility, where is the incompatibility, how can the app code be adjusted?
- Does the included library contain any security vulnerabilities, and does the developer's code happen to use this potentially insecure code?

We will now describe each component of Up2Dep in details.

4.4.2.1 Analysis Tools

As mentioned above, Up2Dep collects information about third-party libraries using existing analysis tools.

LibScout We provide developers information about the API of third-party libraries that they include in their apps. In particular, we notify developers if they can upgrade a library to the latest version or if the newer version would be API-incompatible with the existing app code. Hence, we need to analyze the library history to find out if any of the used library APIs have changed in newer versions of the library. When such changes occur, we provide developers with further information on how they can adapt their existing code so that it will be compatible to the newer version of the library. To this end, we leverage the open-source tool *LibScout* to produce API information for each library version in our dataset.

Library API database: The last version (2.3.2) of *LibScout* contains a dataset of around 250 libraries. In this work, we build on and extend the library database of *LibScout*. In particular, a library on a third-party repository would usually come with a descriptive file, e.g., *pom.xml*, and we analyze those files to discover transitive dependencies of the

libraries in the *LibScout* database. A transitive dependency is another library on which the library included by the app developer depends. For instance, the *Facebook Login Android Sdk* library version 4.40.0 declares three transitive dependencies in its *pom.xml* file: *Android AppCompat Library V7, Facebook Core Android SDK, Facebook Common Android SDK*. To obtain a list of popular third-party libraries that developers commonly include in their projects, we crawl the F-Droid repository [58] to extract libraries included in open source apps. In the end, we have a dataset of 1,878 libraries with their version history. We also extend *LibScout*'s list of publicly disclosed vulnerabilities of third-party libraries. As of July 2019, our list contains 10 libraries with a total of 97 vulnerable library versions.

Determining API Compatibility: To determine the API compatibility between any consecutive library versions, we use the API diff algorithm of *LibScout* that operates on two sets of public APIs api_{old} and api_{new}, where api_{old} is the API set of the immediate predecessor version of api_{new}. An API is presented by its signature that consists of package and class name as well as the list of argument and return type, e.g. example.com.ClassB.foobar()java.lang.String. If $\text{api}_{\text{old}} = \text{api}_{\text{new}}$ two versions are consider compatible. If $\text{api}_{\text{old}} \nsubseteq api_{\text{new}}$, the newer version has added new APIs but did not remove or change any existing ones. This is also considered as compatible (backwards). Whenever api_{old} includes APIs that are not included in api_{new}, a type analysis is conducted to check for compatible counterparts in api_{new}. Compatible changes also include generalization of argument types, e.g., an argument with type String is replaced by its super type Object. Generalization on return types is normally not compatible and depends on the actual app code that uses the return value. If any of the api_{old} is not found in the set of api_{new}, we consider 2 versions incompatible.

CogniCrypt We employ the static analysis component of *Cognicrypt*, namely *CogniCrypt_*SAST, to discover insecure uses of cryptographic APIs within the libraries in our dataset. *CogniCrypt_*SAST takes rules written in the *CRYSL* language, which define best-practice for secure use of cryptographic APIs, and analyzes Java applications to find any potential violations of the predefined rules. We choose *Cognicrypt* instead of other tools, such as [53, 34, 139], because *Cognicrypt* and `CRYSL` are publicly available and provide the flexibility in defining cryptographic rules while other tools mostly provide hard-coded rules, which are not easy to extend. Besides, *Cognicrypt* provides more comprehensive rules that result in three times more identified cryptographic violations in comparison to previous work [53], and the analysis finishes on average in under three minutes per application. More importantly, *Cognicrypt* leverages serveral extensions [98, 18] of the program analysis framework Soot [150], which performs intra- and inter-procedural static analysis that gives *Cognicrypt* and `CRYSL` a high precision (88.95%) and recall (93.1%).

Cognicrypt's rule set [97] includes 23 rules covering Java classes involving cryptographic key handling as well as digital signing. All rules are available on Github [45]. Beside the these rules, we have also written an additional rule for *http* (to check whether a library uses *http* instead of *https* to communicate with a server)

Cryptographic API misuse dataset: We apply *Cognicrypt* to our dataset consisting of

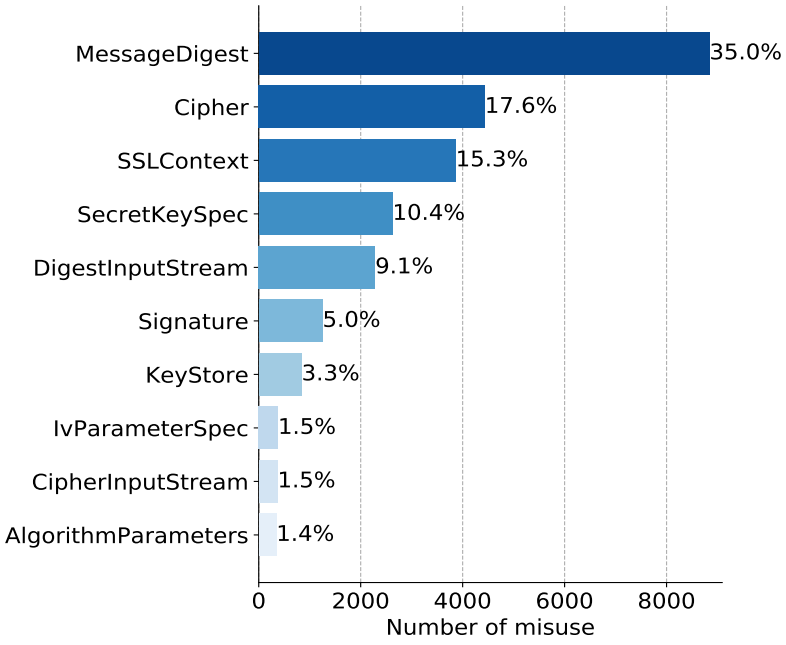

Figure 4.2: Top 10 cryptographic API misuses by Java classes in our library dataset.

1,878 libraries. We are able to analyze 1,725 (91.9%) libraries. It took *Cognicrypt* more than 3 hours to analyze the remaining 153 libraries and we terminated *Cognicrypt* when processing a library exceeded 3 hours[2]. Among the 1,725 libraries, 238 (13.80%) contain at least one cryptographic API misuse, and 70 of those affected libraries (29.41%) have fixed/removed the cryptographic misuse in their later versions. This means that, developers could easily avoid such (vulnerable) cryptographic API misuse by upgrading their project's dependencies to the latest version. Figure 4.2 lists the distribution of the cryptographic API misuse of the libraries in our dataset. The list is headed by *MessageDigest* (35.0% of the top 10 misuses). One of the reasons why MessageDigest has a significantly higher number of misuse is that to use it securely, developers (suggested by the Java Cryptography Architecture Standard) must apply a sequence of method calls, e.g, *MessageDigest.getInstance(algorithmName)* followed by *MessageDigest.update(input)*, followed by *MessageDigest.digest()*, etc., combined with minimum required length for the offset of the *update* method. This does not seem trivial to follow. In general, for Java classes such as *MessageDigest*, *SSLContext*, and *Cipher*, developers need to specify an algorithm or a protocol to work with and library developers often use an algorithm or mode of encryption that is considered insecure, such as ECB mode for encryption, or MD5 or SHA-1 for hashing. This puts these classes of misuse among the most common cryptographic API misuses in third-party libraries. Further, we have found 20 cases where the libraries (spanning across 93 library versions) use *http* to communicate with remote servers.

[2]Such libraries are overly complex and mainly serve traditional Java applications, not intended for Android apps. Analyzing one library version already takes hours.

4.4.2.2 Remote Dependency Inspector

Android Studio is built on Jetbrain's IntelliJ IDEA software. However, the major challenge is the implementation of an Android Studio extension for Up2Dep as it is not well supported and very few documentation is available. To learn how the internal system of Android Studio works, we have to manually read Android Studio source code and examine its APIs (e.g., dynamically run and test them) as well as use reflection to access its internal (private) API to enable the crucial functionality of Up2Dep. To effectively inspect an Android project's dependency, we need to implement a custom code inspection. With the gradle build system, Android developers need to declare their project's or module's dependencies (libraries) in a gradle build file (see Listing 4.1). This file is written in the Groovy language. This means we need to write an inspection that is able to analyze Groovy code. IntelliJ IDEA provides an abstract class called *GroovyElementVisitor* that offers plugin authors the options to analyze varieties of Groovy code fragments. For every *GroovyCodeBlock*, Up2Dep looks for a *dependencies* tag and iterates over all declared dependencies to extract *group_id*, *artifact_id*, and *version* string of each dependency (see Section 4.4.1). Up2Dep then checks if the current dependency is available in our dataset (i.e., it checks if we have pre-analyzed this dependency and if the information about its APIs is available in our database). In case the dependency is available in our dataset, Up2Dep gathers all information about the current version up to the latest version, including information on whether a version has security vulnerabilities. The reason we do this is to not only detect the latest version, but also the latest compatible version in case an incompatibility with the app code occurs while helping developers avoid versions with known security vulnerabilities. At this point, Up2Dep knows if a dependency is outdated and which version is the latest one.

Database maintenance: To allow continuous maintenance of Up2Dep's database we set a crawler up to run periodically to get new versions of the libraries in our database and subsequently apply *Cognicrypt* to analyze them for cryptographic API misuse, and *LibScout* to identify API compatibility between library versions. The updated database is retrieved automatically inside Android Studio to timingly provide developers with updatability and security information about their included third-party libraries. For publicly disclosed vulnerabilities, we update our database manually.

4.4.2.3 Usages Analyzer

As we want to provide developers with information regarding a dependency's compatibility and the use of libraries with potentially insecure usages of cryptographic APIs, we need to analyze the developers' code. We built a code dependency analyzer that traverses through all Java and Kotlin files. When developers open a project in Android Studio, and the indexing process of Android Studio has completed, Up2Dep starts to analyze the project's dependencies. We decided to wait for the indexing process to be done before analyzing code dependencies, because it significantly speeds up the analysis process as code files (including resources) have been transformed into a preferable representation, namely *PsiTree*, that allows faster processing. Each file corresponds to

a *PsiTree*, and *PsiTree*s can depend on each other and can contain sub-*PsiTree*s. For every file (*PsiTree*), Up2Dep extracts its dependent *PsiTree*, and resolves the *PsiTree* to find out if it is associated with an external (foreign) code file. In case of a foreign *PsiTree*, Up2Dep checks with the *ProjectFileIndex* class (provided by IntelliJ/Android Studio) to examine if the corresponding code file is in library classes or library source. As the *ProjectFileIndex* class contains information about all included libraries, Up2Dep can resolve a library class or a library source to find its library information (e.g., library name and version). Once the resolving process is completed, Up2Dep records any usages of the library, e.g., method call (including constructor), and saves them for later references. At the end of the process, Up2Dep has a complete dependency tree of source code files (Java and Kotlin) and their corresponding used libraries with details on which library methods the app is using. More specifically, the result of *Usages Analyzer* is a mapping of multiple pairs: code file (Java or Kotlin) and corresponding used library including API usages.

4.4.2.4 Dependency Resolver

The results from *Remote Dependency Inspector* and *Usages Analyzer* are fed to *Dependency Resolver*. For each included library, *Dependency Resolver* checks the library's usages in the app code as reported by *Usages Analyzer*. At this point, *Dependency Resolver* has information on which APIs of the currently included libraries are used in the developer's code. If *Dependency Resolver* finds that any of the used APIs of an outdated library is no longer available in the library' latest version, it picks the library version that is newer than the current version but contains all the used APIs (newer compatible version). Using the information of publicly disclosed vulnerabilities of third-party libraries, *Dependency Resolver* checks if the currently included library version has a known security vulnerability. Additionally, *Dependency Resolver* looks up each used library API to detect if the API leads to cryptographic API misuse in the library, and the details of the misuse. From all those information, *Dependency Resolver* gives developers the following warnings and potential fixes in their *build.gradle* files. The dependency

- is outdated and can be updated to the latest version.
- is outdated and cannot be updated to the latest version.
- is outdated and has a known security vulnerability.
- potentially uses a cryptographic API insecurely.

We notify developers about the security and outdatedness of their project dependencies in the *build.gradle* file as this is the location where developers would manage their project dependencies. Besides, we also leverage the IDE functionality to allow developers to use Up2Dep in batch mode to analyze the whole project and see the analysis results in a separate window. In the following, we describe how Up2Dep notifies developers about the above declared problems.

Outdated version can be updated to the latest version: In this case, all the used APIs of the outdated library are also available in its latest version. *Dependency Resolver* suggests developers to update to the latest version as it will be compatible to the

Figure 4.3: UP2DEP warns against an outdated library.

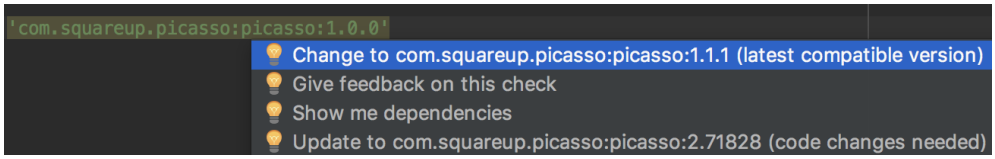

Figure 4.4: UP2DEP provides different options to update an outdated library version.

developer's code (see Figure 4.3). Developers can apply the suggested fix by using the default short-cut of Android Studio (e.g., ⌘+⏎ on Mac computers) or clicking on the default bulb icon to apply the recommended fix. When this quick-fix is applied, the outdated version string of the library declared in the *build.gradle* will be replaced by the latest version. *Outdated version cannot be updated to the latest version:* When not

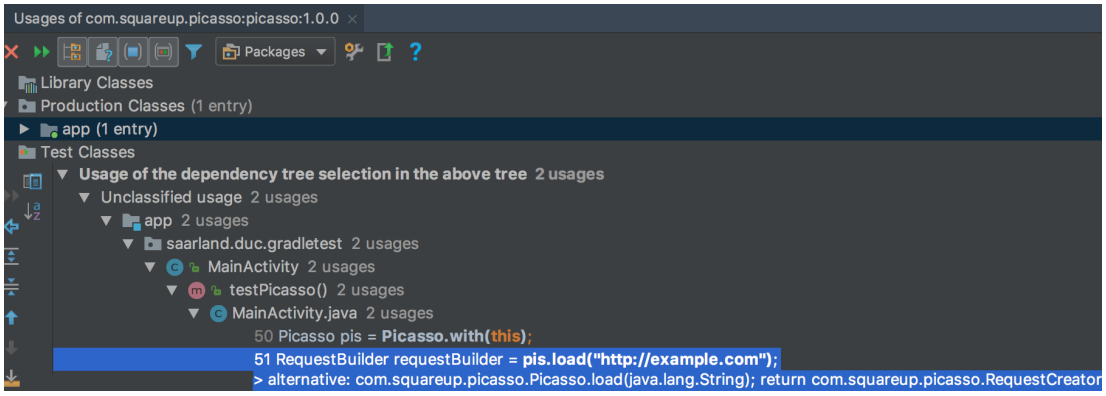

Figure 4.5: UP2DEP shows how developers can migrate their project dependencies to the latest version when incompatibility between library versions occurs, i.e., the return type of method *load* has changed from *RequestBuilder* to *RequestCreator*.

all used APIs of an outdated library are available in the latest version (e.g., because the library developer removed or changed methods), *Dependency Resolver* suggests developers to update to a newer but compatible version. This means the newer version would not require changes to the app code to adapt to the library's API changes. Similar to the previous fix, developers can apply it by using the default short-cut or clicking on the default bulb icon. When no compatible version is available and developers still want to update the outdated library to the latest but incompatible version, they are provided the option *Show Dependencies* (see Figure 4.4). The purpose of the *Show Dependencies* fix is to give developers feedback on how and where they can migrate their project's dependencies to the latest versions (see Figure 4.5).

Outdated library version with known security vulnerability: When the included library

contains a known security vulnerability, *Dependency Resolver* alerts developers with an error (in red color) with details on the vulnerability. Developers can further check the vulnerability in the attached link to our Up2Dep project website (see Figure 4.6). Since a known security vulnerability can be a serious problem for the host app or end-user, we use a red warning instead of a normal warning (in yellow color) to notify developers. In this case, developers can upgrade to the latest version that contains the security patches. When the latest library version is also vulnerable, developers are recommended to consider not using this library.

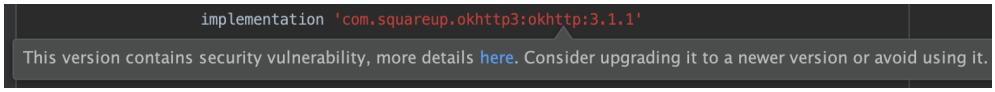

Figure 4.6: Up2Dep warns against using an insecure library version (with publicly disclosed vulnerability).

Use of insecure cryptographic APIs: Similar to known security vulnerabilities, if any used library method happens to insecurely use a cryptographic API, *Dependency Resolver* warns developers in form of errors against using this API (see Figure 4.7). In this case, Up2Dep suggests to developers to update to the latest version if the used APIs in the latest version do not contain cryptographic API misuse. If the latest version still has that problem, developers can use the *Show dependencies* option to examine the location and necessity of the used library method and decide whether or not they can remove the used method call, or switch to another library.

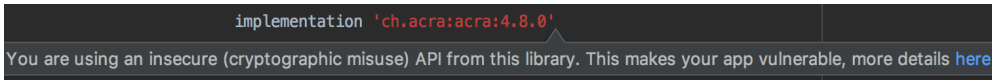

Figure 4.7: Up2Dep warns against re-using a cryptographic API misuse in a library.

4.5 Evaluation Design

Our goal is to find out if it is technically feasible for Up2Dep to support developers in keeping their project dependencies up-to-date and in avoiding library versions with security problems, e.g., how many outdated (including insecure) libraries Up2Dep has helped developers migrate to the latest versions and which security vulnerabilities it has fixed for developers. We further examine developers' Up2Dep experience in an online survey (see Figure 4.8). Different aspects of Up2Dep in interacting with developers — studying developers behavior upon learning about the security of an included library, how security warning messages can be customized, how can we keep the balance between developers being annoyed and being informed, how developer's mental model evolves — are not in the scope of this work, and left for future work. In the following, we first describe how we enable developers to evaluate Up2Dep in-the-wild. We then report how we advertised Up2Dep and delivered it to Android developers for evaluation.

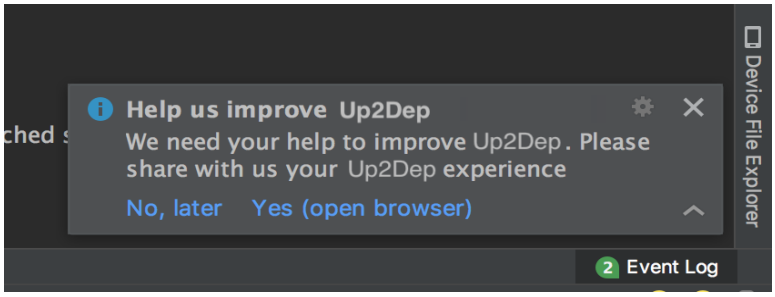

Figure 4.8: Invitation to our online survey inside Android Studio.

To enable developers to evaluate Up2Dep while they are using it we leveraged the remote study platform of FixDroid to setup and conduct our evaluation. In particular, we included telemetric features that record whether a suggested quick-fix has been applied. We provided developers the *Feedback in context* (see Figure 4.9) option[3] where they could send us feedback on whether the suggested fix works as expected, if they needed more information on any particular warning, or on other issues they encountered. In Up2Dep's instruction, we *strongly encouraged* developers to provide us feedback so that we can make Up2Dep better in the future, this was where they can help us to help them, i.e., making a free-to-use tool better for them. Developers were also provided the option to opt-out of our telemetric data collection in Up2Dep's settings. Before developers downloaded Up2Dep we clearly informed developers on which information we gather about their usage (on our project's website and in Android Studio plugin repository description). Our goal in this step was to make sure they are well informed before they decide to install our plugin.

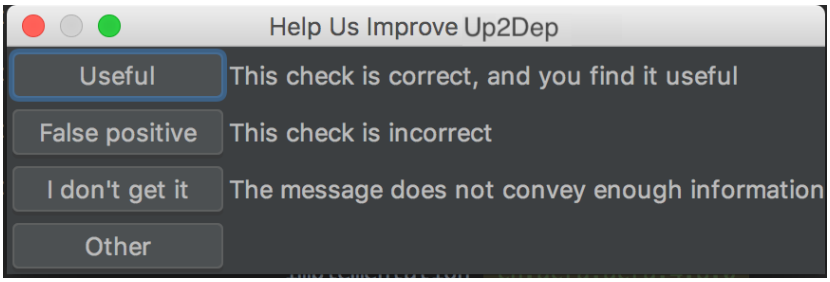

Figure 4.9: In context feedback dialog.

4.5.1 Recruitment

After we advertised Up2Dep's prototype at an Android developer conference, we used Twitter and email as communication channels to keep in contact with developers and to recruit further developers. After we released Up2Dep with complete features, we advertised our tool on different Android developer forums, Android developers groups on *Facebook*, and in a related lecture at our institution to invite experienced students,

[3]This feature was adopted from Lint tool.

who are working on real (non-study-related) Android projects[4], into using Up2Dep. Finally, we sent an invitation email to an Android development team, with which we already had contact before, to ask the team to try out Up2Dep.

We abstained from sending emails to the contact information harvested from Google Play apps, as done in prior studies [5, 49, 77], since those studies had an extremely low response rate and such mass emails may be considered as harmful/spamming behavior that would create a negative view from developers toward studies conducted by researchers.

4.5.2 Ethical Concerns

This study has been approved by our institution's ethics review board. All telemetric information is gathered anonymously—we do not know who the developer is—and we do not collect the developer's code. Furthermore, we clearly explain on our website which information we gather and provide developers the option to opt-out of our telemetry data collection at any time. Finally, all data is sent to our server over a secured connection.

4.6 Results

In this section, we present our evaluation results, which provide the answers to our research questions (RQ) stated in Section 4.3. This covers both telemetric data of developers who filled out our exit survey as well as of developers who are using Up2Dep but did not answer our survey. Our evaluation has lasted for 81 days, the results we report in the following are from within this duration. All data related to Up2Dep tutorial was excluded from our results[5]. Finally, we briefly compare Up2Dep with *LibScout* and *Cognicrypt*.

4.6.1 RQ1: Would it be technically feasible to support developers in keeping their project's dependencies up-to-date?

From the telemetric data and answers to our online survey, we can see that developers have made use of Up2Dep to keep their project dependencies up-to-date. In particular, Up2Dep helped developers upgrade their project's dependencies (N=116) to the latest version in 34 *real* projects. We describe the data as well as the feedback developers have provided in details in the following.

[4]Projects that are not related to their university studies/courses

[5]When Up2Dep recorded telemetry data, it computed a hash value of the project's name. If developers used Up2Dep in a project that has the same hash value with one of our exemplary projects, such data was excluded from our results.

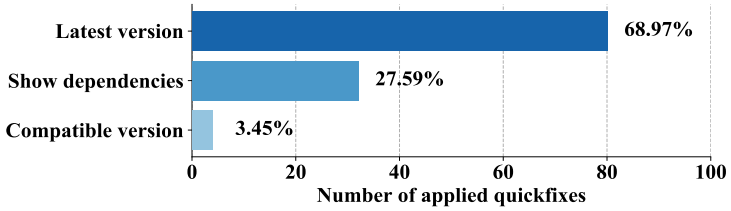

Figure 4.10: Number of applied quickfixes per type.

Telemetric Results As we included telemetric features in Up2Dep, we are also able to gather telemetric data from developers who did not participate in our survey. Of 56 developers who are using Up2Dep, 30 (53.57%) have applied quickfixes (N=116) provided by Up2Dep to update their project's dependencies—i.e., updated an outdated third party library to the latest/newer compatible version or examined a library's API dependencies (34 projects).

Figure 4.10 shows the number of applied quickfixes per type. We can see that the majority of applied quickfixes are *Update to the latest version*. Besides, 27.59% of quick-fixes belong to *Show dependencies* meaning that developers have checked the API usages of the corresponding dependency. However, since we do not collect developer's code, we do not know whether manual code change were performed to update the corresponding dependency.

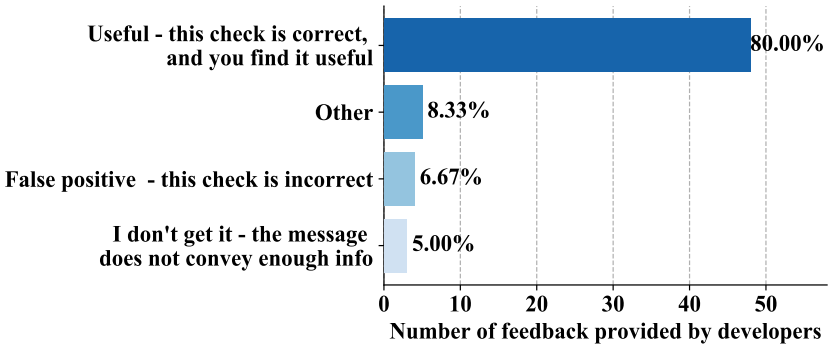

Figure 4.11: Feedback given by developers in context. Developers can give feedback to multiple quickfixes.

Among all 30 developers who have applied suggested quick-fixes in their projects, 22 of them (73.33%) have provided us feedback through the feedback dialog (Figure 4.9). On average developers have spent 19 minutes working with Up2Dep before giving us the first feedback. The results from the 60 in-situ feedbacks we received from 22 developers confirm that 80.0% of the proposed fixes worked, and developers found the warning/quickfix *useful* while only 4 proposed fixes did not work as expected. Figure 4.11 lists all feedback provided by developers. We also observe that 5.0% of the feedback

indicates that the developer did not understand the warning message (*I don't get it*). We manually examined the corresponding third-party libraries and found that their warnings were about cryptographic API misuse. This suggests that we need to make the warning message more developer-friendly, e.g., make it easier to understand (similar to other domains such as browser security warnings [8, 61]). As each feedback came together with the dependency for which developers had given feedback, we manually investigated the feedback that belongs to *False positive* and *Other*. We noticed that transitive dependencies might be the reason for such feedback. When a third-party library A depends itself on library B in version v_1 and developers use library B version v_2 in their app code, this means this project has now two versions ($v_1 \neq v_2$) of the library B. This might break the app due to unresolved dependencies. We found transitive dependencies' problems in: *org.jsoup:jsoup:0.22* and *com.jakewharton:butterknife:7.0.1* (found in the *False positive* feedbacks). Both of these dependencies have transitive dependencies that app code itself makes use of. Up2Dep suggests developers to update them to the latest versions. Although the latest versions provide all APIs that the apps are currently using, but they no longer contain the exact transitive dependencies (version) that the apps are using, this in the end breaks the functionality of apps. Since we do not collect developer's code, we cannot evaluate which API of a library developers are using in their project. We decided to further study this problem on open source Android projects. We collected libraries (*org.jsoup:jsoup:0.22* and *com.jakewharton:butterknife:7.0.1*) that are found in the *False positive* feedbacks, and found projects on F-Droid repository that have such dependencies. We further investigate the problems of transitive dependencies and report our finding in Section 4.6.2.2.

4.6.1.1 Online Survey Results

Demographic data: Of 56 developers, 23 have shared their Up2Dep experience with us in our online survey. Developers have spent on average 48 minutes working with Up2Dep before joining our survey (see Table 4.1 for details). Around half of the developers have less than one year of Android programming experience, while the other half has at least two years of experience. In particular, 11 developers developed more than 2 Android apps, while only 3 participants have not yet published any apps. About two-thirds of the developers have a security background, most of them are male, and their age ranges from 18 to 30 years. Among 23 developers, 9 of them are students (2 with at least 2 years of programming experience, 7 with less than 1 year of programming experience) who got to know Up2Dep after we advertised it in a related lecture at our institution[6].

[6]We did not distinguish academic training from programming experience.

Table 4.1: Participant demographics of online survey.

Age	18-30	22
	No answer	1
Gender	Male	21
	No answer	2
Based	Europe	13
	Asia	9
	Other	1
Programming Experience (years)	<1	10
	2	5
	3	4
	>3	4
Apps Developed	>2	12
	2	6
	1	2
	0	3
IT-Security Background	Yes	17
	No	6

Usability score: To assess the usability of Up2Dep in our survey, we used the SUS (System Usability Scale) [28]. A system with a SUS score of above 68 would be rated as above average. Up2Dep achieves a SUS score of 76.20, which is considered good in terms of usability [22] (see Figure A.1 in Appendix).

Useful features: Of Up2Dep's features, the *Compatible version check* was named most often (see Figure 4.12). This supports the results of a previous study [49] that showed that developers abstain from updating their project's dependencies due to (fear of) incompatible updates.

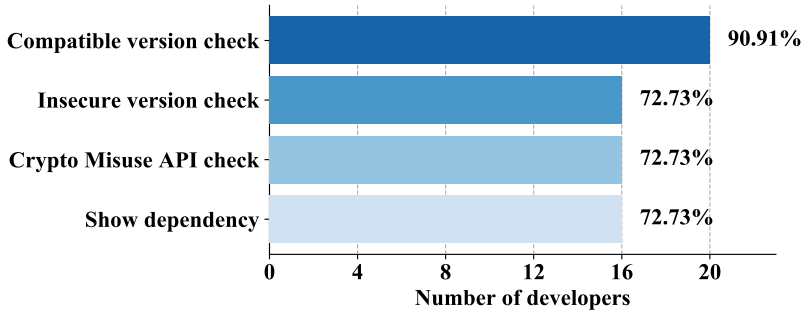

Figure 4.12: Features of Up2Dep that developers find useful. Developers can choose multiple features.

4.6.2 RQ2: Could such a tool support have a tangible impact on the security and privacy of Android apps?

4.6.2.1 Fixed Security Problems

We observe that there are 4 instances of the *okhttp3* v3.0.0 library in developers' projects, which contains a known security vulnerability. *okhttp* v3.0.0 allows man-in-the-middle attackers to bypass certificate pinning by sending a certificate chain with a certificate from a non-pinned trusted CA and the pinned certificate. Zhang et al. showed that nearly 10% of the most popular apps on Google Play store still used such an insecure version for more than 1 year after the fixed version had been released [160]. In our study, those library versions were updated by developers with the support of Up2Dep to the latest, fixed version. Furthermore, there are 3 instances of an outdated version of the *Glide* library where developers used hash API without calling the complete sequence of function (see Section 4.4.2.1). Finally, one instance of *okhttp3* v3.11.0 that misused a cryptographic API, and the developer in our study happened to re-use the correspond API of the library. This issue has been fixed in their latest, misuse-free version of the library. All in all, 6.89% of the outdated dependencies that Up2Dep has helped developers to migrate to their latest versions (8 out of 116) had security problems. Since we do not collect information of the developers' projects (i.e., this may make developers skeptical to try Up2Dep), we therefore do not have information on the projects patched by Up2Dep. However, regardless of the project details, we consider this number non-negligible given the easy means that developers can employ to fix them. Therefore, by fixing projects containing these insecure library versions, Up2Dep directly benefits the security and privacy of Android apps.

4.6.2.2 Security Problems of Transitive Dependencies

From the feedback related to the *False positive* category, we learned that for a small number (2) of cases, the problem of transitive dependency would prevent developers from keeping their project's dependencies up-to-date because of incompatibility. However, the current dependency management system of *Gradle* makes it hard for developers to be informed about what are the transitive dependencies of the manually declared dependencies as it automatically downloads sub-dependencies of a given dependency without developers easily noticing it. Developers can check the log console to see what sub-dependencies are downloaded together with the current dependency, yet this is only available in the log console with hundreds of log events. The problem becomes more serious if a transitive dependency has (well known) security problems. Those are totally hidden from developers because they are usually automatically downloaded following the main dependency unless developers specifically exclude them [79]. Thus, even if developers would vet a dependency manually, insecure sub-dependencies that are automatically, non-obviously pulled in when installing the dependency can undermine the app's security again. This highlights the need for tooling support, as such Up2Dep.

Transitive dependency analysis: Given the crucial information regarding security problems of transitive dependencies, we developed an additional feature that thoroughly checks all transitive dependencies of all declared dependencies to: (1) analyze com-

patibility when suggesting developers to update the declared dependencies, and more importantly, (2) to check and notify developers if any transitive dependencies contain security problems. When Up2Dep detects a declared dependency in a *build.gradle* file of a project, it checks all transitive dependencies (all sub levels) of the current dependency and queries security related information of these transitive dependencies. If any transitive dependency contains security problems, developers are notified similar to how security problems of the main dependency are communicated (see Section 4.4.2.4).

Analysis Results: Our results reveal that there are 1,209 library versions (belonging to 112 unique dependencies) that have security problems. These dependencies are currently (transitively) used by 9,787 library versions (233 unique libraries) in our data-set. Especially, among 1,209 transitive dependency versions with security problems, 16 contain a publicly disclosed vulnerability. This means even if developers are aware of such libraries with security problems they have no way to find out if their projects are including such insecure dependencies as they are not visible to developers. The latest version of Up2Dep now informs developers about such security problems of both the main dependency and transitive dependencies so that developers can also avoid insecure transitively included library (versions).

4.6.3 Comparison with Existing Work

In our work, we significantly increased the database of *LibScout* by a factor of 7.5x. Furthermore, our database covers the top 100 most popular libraries on Maven repository [149] which was not considered by *LibScout*. Most importantly, we provided an effortless synchronization (end-to-end) process that automatically scans for new libraries (versions), analyzes for cryptographic API misuse, then the information on security and updatability of new libraries (versions) are delivered to developers right in their development environment without them having to use extra tools.

Besides, as we extended the rule set of *Cognicrypt* to include the check for use of *http* protocol, we have found 20 libraries (8.4% of all identified insecure libraries), spanning across 93 versions using such insecure protocol. With the original rule-set of *Cognicrypt* we would have missed the insecure network connection in these libraries.

4.7 Discussion

4.7.1 Threats to Validity and Future Work

Our work leverages *LibScout* and *Cognicrypt* and inherits their limitations. For *LibScout* the ability to provide suggestions for API changes relies solely on API heuristics, such as name, parameter types, or return types, which do not necessarily guarantee that the suggested API will work as expected. If the semantics or side-effects of a library method change between versions, this could break the functionality of the developer's app although the app code was compatible at the method signature level with the new library version. Further detecting semantic changes is an open problem that requires effort from different domains, especially software engineering, and is not in

the scope of our work. Yet in this work, we show that relying on API changes to derive compatibility among library versions does help developers to keep their project's dependencies up-to-date, yet it needs further improvement to cover more cases.

While *Cognicrypt* provides the flexibility to create new rules to detect cryptographic misuse, it is not free of false positives. We found cases where calls to cryptographic APIs are wrapped in custom Java utility classes by the library developer. *Cognicrypt* can not completely link the control flow graph of those custom classes to detect if a cryptographic misuse occurs in those cases. This results in *Cognicrypt* over-approximating the misuse and reporting false positives. In particular, misuse of *MessageDigest* depends on call sequences and this shortcoming of *Cognicrypt* in classifying misuse of that class when being wrapped in custom classes might be a contributing factor to the high number of misuses detected for *MessageDigest* (see Figure 4.2). However, it is not easily possible to verify such misuse using static analysis and exclude false positives from our results. Once *Cognicrypt* addresses this limitation, also Up2Dep will provide more accurate warnings to app developers.

Additionally, we currently manually look for publicly disclosed vulnerabilities, which is a tedious task. In future, this could be generally done with a central library repository, e.g., when a vulnerability of a library is disclosed, central library repositories can incorporate and mark the vulnerable versions in their database so that tools like Up2Dep or *Lint* can automatically retrieve and provide developers feedback in their IDE. However, for the cryptographic API misuse, Up2Dep's pre-analyzer component automatically crawls newer versions of third-party libraries and runs *Cognicrypt* to obtain up-to-date results.

Further, the population size of the developers in our evaluation might be perceived as small since we only have 56 developers, of which 23 shared with us their experience in our online survey, and 22 developers provided us feedback in their Android Studio. Our demographic data shows that our evaluation indeed has a population of experienced developers (e.g., 18 of them have developed at least 2 Android apps). However, developer studies [5, 77] had in the past notoriously a low number of participants as it is not easy to recruit real developers. Besides, most of them were conducted with students as proxies using handcrafted, toy projects which do not necessarily represent the day-to-day *real* situation that developers often face. In our work, on the other hand, we tried to avoid students as proxies and toy projects as much as possible and gain insights from developing real app projects (external validity).

We think the fact that we could recruit this number of developers and keep them using Up2Dep is in part due to the interest and need for such a tool by the developer community. Furthermore, with our feedback in-context option, we obtain valuable feedback from developers on whether Up2Dep works. Given the only small percentage of false positives reported (6.7%) and 80% of the suggested quick-fixes working as expected, we believe that we have delivered a novel and expedient tool, and can show the impact of such tooling support on real world situations.

Lastly, we abstained from collecting telemetric information on whether developers ignored the quick-fix, since this might be considered too intrusive. Unfortunately, this also precludes us from modeling whether a known security vulnerability or cryptographic

misuse warning is a significant predictor for applying quick-fixes and library updates in our evaluation.

4.7.2 Transitive Dependencies and App Security

While during our evaluation, we did not consider transitive dependencies, we also have seen that the problems of transitive dependency with regards to library updatability is a corner case, e.g., only 2 instances of the false positives. Also existing research [87] on the updatability of third-party libraries shows that only 1.7% of the library API could be affected by this problem (referred to as entangled dependencies). Still we see a potential threat to the security of Android apps due to transitive dependencies. We found that (known) security problems of a library could be hidden from developers when the library is included as a transitive dependency of another dependency and this transitive dependency is not communicated as obvious to app developers as needed. While the community is trying its best to find security related problems of third-party libraries, it is also important to keep developers informed on all potential risks associated with a (declared) dependency. We are to the best of our knowledge the first to study the security problems of transitive dependencies and subsequently developed a solution to tackle this problem by alerting developers when they include libraries that have transitive dependencies with security problems.

4.7.3 Impact of Fixing Insecure Dependencies

Among the 116 applied fixes, 6.89% had security vulnerabilities (4 known security vulnerabilities, 4 cryptographic API misuse). We consider these numbers non-negligible and this has tangible impact on the security and privacy of the Android apps that developers are working on. Previous work has identified the security & privacy impact of outdated third-party libraries in general and of outdated *insecure* third-party libraries in particular (see Section 4.8). By updating the insecure code dependencies to secure versions, we are removing the factors that could amplify security & privacy problems in apps and expose end users to multiple types of attacks. While market stores such as *Google Play* have been scanning apps for security & privacy problems, they are dealing with monolithic byteblobs where there is no separation between app code and library code. Hence, such solutions need exact, reliable library detection mechanisms which is a challenging task and no satisfactory solution exists yet. This becomes even more challenging when the apps' byte-code is obfuscated, something that *Google* itself is promoting to app developers [12]. Our results show that by integrating support to suggest secure code dependencies within developers' IDEs, we can eliminate many security problems that arise from including insecure third-party libraries without having to deal with monolithic apks where app code and library code have been merged together. Especially, developers do not need to learn new tools or adjust their daily work-flow to be able to use Up2Dep. Our results call for action from IDE developers to merge tools like Up2Dep into IDEs, like Android Studio, so that developers immediately and by default benefit from such support. Based on our results, the experiences in other software ecosystems [140, 68] or for native Android libs [11, 71], and the movement toward integrating security into software development life cycle namely *SevDevOps* [111,

112], we argue that this would have a tangible impact on the security & privacy of the Android ecosystem especially.

4.7.4 Fear of Incompatibility vs. Will to Update

In our evaluation, we learned that, the majority of the outdated libraries can be updated all the way to the latest version (see Figure 4.10) without having to change the app code (i.e., 68.97% quick-fixes are *update to the latest version*). Developers are afraid of updating because they fear that the new version of the libraries would break the app's existing functionality [49]. Without the information on the compatibility of the new update, developers either have to manually verify the release notes (if available) of the libraries to make sure that the functions their apps are using are still available in the update, or simply keep using the outdated versions. One developer explicitly shared such experience via email with us after trying out Up2Dep:

> "Thank you for sharing your project with me. It's really exciting, we're normally manually reviewing the change logs to see if we should update our dependencies right away or what we should test."

Compatible check was rated the most useful feature (see Figure 4.12) by developers in our study. Had Up2Dep not provided the compatibility information on the outdated dependencies, we would not think that developers would be willing to perform the updates on these 68.98% outdated dependencies (80 of 116 outdated dependencies).

4.8 Related work

We discuss related works on studying the security of third-party libraries and on tool support for developers in creating more secure apps.

Security of third-party libraries: Sonatype reported that almost 2 billion software components were downloaded per year that contain at least one security vulnerability, and that outdated software components had a three times higher rate of security issues [1]. In the Web world, Lauinger et al. [99] showed that 37% of 133k analyzed websites include at least one library with a known vulnerability, and that it takes years for web developers to upgrade the included dependencies to the latest version. On the other hand, regarding Android applications, Stevens et al. [145] investigated the user privacy in Android advertisement libraries and found that among 13 investigated ad libraries, several of them are over-privileged. Looking further into Android apps, Backes et al. [19] proposed the *LibScout* tool to detect third-party library code in Android apps, and found that 70.4% of the included libraries in their dataset are outdated. They also found that it took developers on average almost one year to migrate the app to the latest library version. Muslukhov et al. [113] proposed *BinSight*, a static program analyzer that was capable of identifying source attribution in Android applications. The authors revealed that for 90% of the apps that contain cryptographic API misuses, at least one violation originated from third party inclusions. Watanabe et al. [155] also found that 70% and 50% of vulnerabilities of free and paid apps, respectively, stemmed from software libraries, particularly from third-party libraries.

Tool support for software developers: Prior work has proposed different approaches and tools to support developers in building more secure Android apps. Among them, many developed tools to find vulnerabilities in applications after they have been released [55, 104, 124]. This means that developers were only aware of such security mistakes at the end of or after their development cycle. Other tools [96] provided developers support while they were writing code. Krüger et al. [96] developed *Cognicrypt* to support developers in securely using crypto APIs. Rahaman et al. [132] proposed a set of analysis algorithms and a static analysis tool namely *CryptoGuard* for detecting cryptographic and SSL/TLS API misuses to help developers analyze large Java projects. Focusing on supporting Android developers in writing more privacy-friendly apps, Li et al. [100] proposed *Coconut*, an Android Studio plugin that engages developers to think about privacy during app development and to provide real-time feedback on potential privacy issues. Further, Android Studio, Google's official IDE to develop Android apps, includes *Lint* tool [102] to check for outdated third-party libraries. *Lint*, however, only informs developers about whether or not a newer version of the library is available.

None of the above solutions supports developers in keeping their project dependencies up-to-date while taking into account the effort to update the dependencies, the compatibility of the update, or the potential security vulnerabilities of the different dependencies' versions. While being the Google-provided tool for Android developers, also Android Studio has not considered all these aspects to help developers in keeping their project dependencies up-to-date, and especially to avoid insecure library versions.

4.9 Conclusion

Since security patches of libraries are often rolled out as updates, app developers (*not* market stores, *not* the end users) need to keep their project's third-party libraries up-to-date to avoid security problems of outdated libraries. In this work, we present Up2Dep, an Android Studio extension that facilitates the task of keeping an Android app project third-party libraries up-to-date while taking into account the security and the compatibility of the newer versions of such dependencies. Up2Dep suggests alternative library APIs to developers in case a newer library version is incompatible. It further helps developers in avoiding insecure libraries by alerting them to publicly disclosed vulnerabilities and cryptographic API misuse in third-party libraries. We tested Up2Dep with 56 Android developers. Up2Dep has helped developers in fixing 116 outdated third-party libraries, of which 6.89% had security vulnerabilities (4 known security vulnerabilities, 4 cryptographic API misuse). The majority (80.0%) of the suggested quick-fixes worked as expected with only 4 cases of failed quick-fixes. In further investigation, we discovered the hidden security problems of transitive dependencies of 13.51% of the libraries in our dataset. We are the first to discover the hidden problem of insecure transitive dependencies and subsequently developed the corresponding solution to tackle this problem. Our results call for action to (1) merge tool support, like Up2Dep, into developers' integrated development environments, as this would create a tangible impact on the security and privacy of the Android ecosystem when developers benefit from tool support for upgrading used third-party libraries, and (2) study developer's

behavior to best provide them the right tool support.

5

Measuring the Impact of User Reviews on Android App Security & Privacy

5.1 Motivation

Application markets streamline the end-users' task of finding and installing applications. They also form an immediate communication channel between app developers and their end-users in form of app reviews, which allow users to provide developers feedback on their apps. However, it is unclear to which extent users employ this channel to point out their security and privacy concerns about apps, about which aspects of apps users express concerns, and how developers react to such security- and privacy-related reviews.

In this work, we present the first study of the relationship between end-user reviews and security- & privacy-related changes in apps. Using natural language processing on 4.5M user reviews for the top 2,583 apps in Google Play, we identified 5,527 security and privacy relevant reviews (SPR). For each app version mentioned in the SPR, we use static code analysis to extract permission-protected features mentioned in the reviews. We successfully mapped SPRs to privacy-related changes in app updates in 60.77% of all cases. Using exploratory data analysis and regression analysis we are able to show that preceding SPR are a significant factor for predicting privacy-related app updates, indicating that user reviews in fact lead to privacy improvements of apps. Our results further show that apps that adopt runtime permissions receive a significantly higher number of SPR, showing that runtime permissions put privacy-jeopardizing actions better into users' minds. Further, we can attribute about half of all privacy-relevant app changes exclusively to third-party library code. This hints at larger problems for app developers to adhere to users' privacy expectations and markets' privacy regulations.

Our results make a call for action to make app behavior more transparent to users in order to leverage their reviews in creating incentives for developers to adhere to security and privacy best practices, while our results call at the same time for better tools to support app developers in this endeavor.

5.2 Problem Description

Application markets such as Google's Play or Apple's App Store are core components in mobile software ecosystems. They constitute centralized markets for developers to distribute their apps and for end-users to search, download, and purchase applications. Similar to online retail markets, end-user reviews are a key element to the success of app markets. Users that have used an app can write reviews—short text messages typically including a star-rating—to express their opinion about an app and help other users to choose between similar apps. At the same time, reviews can also be used as a direct feedback channel to app developers, e.g., to express feature requests or to report bugs and security issues. The app developers, in turn, can react to this feedback and reply to their users.

Although user reviews form a direct communication channel betwen users and developers,

past research on security and privacy protection has—to the best of our knowledge—not given this channel any attention. Prior research focused instead, for instance, on providing users with support in choosing less risky apps [131, 76] or on helping users making informed decisions whether or not to grant permissions to an application [118, 157, 135]. Although such support is undeniably valuable for helping users, we believe that those also form short-term solutions that do not immediately tackle the root cause of developers releasing apps that disregard privacy best practices. For apps to improve their security- and privacy-related behavior in the long-run, feedback should not only be directed to end-users but also to developers, ideally in a way that the developers have incentives and motivation to update their apps according to the security and privacy concerns of their users. User reviews would seemingly form such an immediate feedback and rating channel for security- and privacy-related user concerns. Unfortunately, the extent to which reviews can provide this kind of feedback and how developers react to such feedback have not yet been investigated.

5.3 Contributions

In this work, we study the connection between security- and privacy-related reviews (*SPR*) and security- and privacy-related app updates (*SPU*). Concretely, this includes questions like *"To which extent do SPR trigger SPU in apps?"*, *"How often do app developers react to SPR (e.g., due to the fear of follow-up reviews with low ratings and a potential financial loss)?"*, and *"What kind of SPU do app developers do in consequence of SPR?"* To answer those questions, we first build a crawler to collect the complete version histories of the top 2,583 apps (62,838 app versions) on Google Play and their corresponding 4.5M user reviews. We then use supervised learning techniques to identify 5,527 security and privacy relevant reviews. By retargeting the release dates for both the app versions and the reviews, we connect those SPR with the corresponding app version that was mentioned in the SPR. Using static code analysis, we classify the changes between those user-reviewed app versions and their immediate successor versions as SP-relevant when later app versions behave more privacy-friendly. Using recent advances in statically detecting third-party libraries [19], we are able to attribute those SPU to either app or library code changes. Using this data set, we then set out to thoroughly examine the impact of user reviews on the SPU of android applications. We build a statistical regression model that takes different factors into account that could affect the update of an app, including users' variables (e.g., ratio of SPR received, and review star rating) and app variables (e.g., permission mechanism, the ratio of replies to reviews, and app category). By applying our regression model to our entire data set of reviews and app histories, we are able to show that SPR are significant predictors of SPU in Android applications. This means the more SPR an app version receives, the more likely the subsequent version of the app will be an SPU. Additionally, our results show that of all SPU, only 17.06% could be uniquely attributed to app code while 48.81% could be uniquely attributed to (closed-source) third-party code, meaning that in most cases SPR complained about app behavior that was added to the app through inclusion of third-party code. Furthermore, through statistical testing, we confirm that app versions that use Android's run-time permission dialogs raise more suspicion from users,

expressed through a significantly higher rate of SPR for those app versions (1.46 times more than for install-time permissions).

Based on those results, we conclude that SPR indeed have a positive influence on the privacy-related development of apps and that there is a clear call for action to not only support users in making better choices but also making app behavior explicitly more transparent to users to foster higher rates of SPR that express users' privacy attitudes and create incentives for developers to adhere to privacy best practices. Developers, on the other hand, clearly need support in this task, in particular in estimating the impact of included third-party code onto their apps' privacy-critical behavior.

In summary, we make the following contributions:

- We investigate security- and privacy-relevant features in apps that can be perceived by end-users (e.g., permission requests and data accesses) and map them to permission-based functionality that can be extracted from apps.
- We build a longitudinal repository of 2,583 applications and their 4.5M user reviews. We build a classifier to identify SPR with a very good accuracy (mean AUC value of 0.93). By retargeting app release dates, we can map SPR back to their affected app versions in 88.62% of all cases.
- We statically extract permission-based features from apps mentioned in SPR and identified SPU of apps in 60.77% of all SPR. Further, 48.81% of those SPU can be attributed exclusively to (closed-source) libraries.
- We build a statistical regression model to evaluate the impact of different factors on apps' SPU, including users' variables and app variables.
- Our approach reveals that SPRs are a significant predictor of SPU of Android apps and that apps supporting runtime permissions dialogs receive 1.46 times more SPRs than apps with install time permissions.

Outline: This work is organized as follows. We give an overview of related work in 5.8 and describe our methodology in 5.4. We empirically analyze our data in 5.5 and explain our regression model to predict SPU in 5.6. We discuss our findings and draw actionable items in 5.7 and conclude in 5.9.

5.4 Technical Description and Approach

In this work, we automatically identify security- and privacy-related reviews (SPR) and map SPR to security- and privacy-related updates (SPU) of the corresponding applications. Figure 5.1 gives an overview of our methodology. We collect the dataset for our analysis with a custom built crawler, which mines Android applications and their version history as well as the apps' reviews from Google Play. After having collected the apps and their reviews, a classifier identifies SPR. Once we have the set of SPR, we establish correlations between SPR of apps and the security and privacy relevant changes within the corresponding apps' release history (S&P Mapping). In the following sections, we will describe the different steps of our methodology in details.

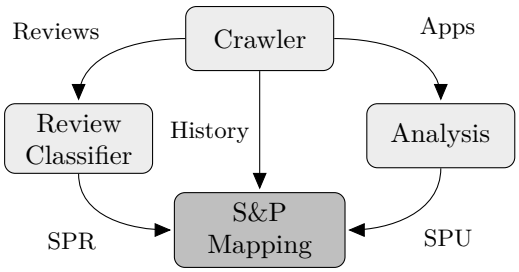

Figure 5.1: Overview of our methodology

5.4.1 App and Review Crawler

Mining user reviews The collection of the user reviews from Google Play consists of two steps: collecting the reviews' text as well as their scores, and then pre-processing the text for later classification.

We built a crawler to collect Android application reviews from Google Play. As previous studies [153, 161] have shown that only a small fraction of free applications on Play accounts for the bulk of the application downloads—a so called *superstar market*—we focus our collection of applications on those apps that are most popular among the users of Google Play. Therefore, our crawler collects all Android applications that have at least 50,000,000 downloads, which results in 2,583 distinct applications as of July 2017 when we collected our dataset. It might seem that 2,583 apps is a very small number of applications in comparison to other market studies on Android, but it has to be considered that we also crawl each app's version history and their corresponding reviews. Thus, we trade a large-scale cross-sectional study, as favored in most other studies on Play, for a longitudinal study of apps that allows us to analyze the evolution and influence of SPR on app security and privacy. Since downloading each app's version history easily amplifies the required time for data collection and analysis [19], we chose to limit our data collection, both app version histories and reviews, to apps that have at least 50,000,000 downloads. We explain the technical realization of our app collection further down.

We only crawl reviews that were written in English by selecting the Play web interface language code accordingly. Besides the review text with its rating score, we also gather developer responses (if available). Our dataset as of September 2017 contains 4,547,493 reviews. We will elaborate on how we compiled this list of reviews later on when explaining our training dataset for our review classifier (see Section 5.4.2)

Crawling app history Studying security and privacy relevant changes of applications (SPU) and their connections with user reviews requires building an app repository with historical information about apps, i.e., including all versions of a particular app, which allows analysis of an app's evolution. To this end, we adopt the approach of Backes et al. [19] that used an undocumented market API to query Google Play for older versions of an app. In the following, we will explain how we obtained the complete history of the top 2,583 apps in Play from September 2017, resulting in a repository of

62,838 distinct app versions (i.e., on average 24.33 versions per app in our collection).

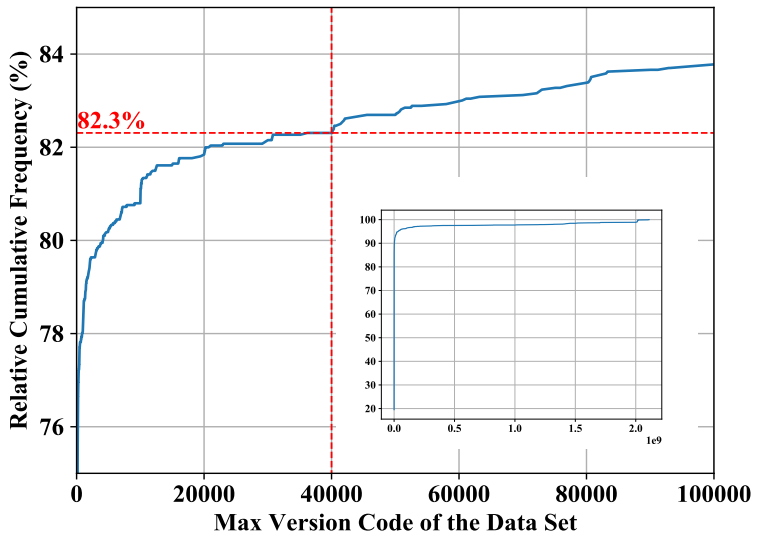

Figure 5.2: Relative CFD of maximum version codes of apps in our data set

History collection: The Play API allows us to query for app versions using the app's packagename and version code. However, there is no option to list the available versions for a given app. Thus, building the history of an app requires probing for existing version codes. The version code is an integer number that must be monotonically increased with every app update, but there is no official numbering scheme enforced. Although the majority of app developers simply increases the version code by one, related work [19] has shown that some developers use special date patterns, such as `YYYYMMDDVV` where `VV` is the revision-per-day. Since exhaustively probing for the existing version codes of each app is very time consuming, we set the threshold for version codes that we test to a maximum of 40k. This gives a coverage of 82.3% for the apps in our data set, i.e., for 2,126 out of 2,583 apps this threshold is higher than their highest version code on Play. Figure 5.2 illustrates the relative cumulative frequency distribution of the maximum version codes in our data set.

Release dates: A second major drawback of the Play API is that it is not possible to query for release/upload dates of old app versions. In order to be able to map reviews to app versions by date, we follow the approach of related work to collect missing release dates from market analysis companies, such as *appannie.com*, *apk4fun.com*, and *appbrain.com*. In total, we were able to recover the upload dates of 81.52% of all app versions in our data set (51,225 of 62,838). For a set of 957 apps we were able to retrieve the complete version history. For the remaining 1,169 apps we have an incomplete set of upload dates, for whose majority (790 of 1,169) we miss the long tail of upload dates, i.e, we could not recover dates for early versions that were published before any of the market analysis services started to collect data. Figure 5.3 shows the distribution of 790 apps for which we miss the long tail of upload dates. For about 70% of these apps,

less than 30% of the whole app history is missing.

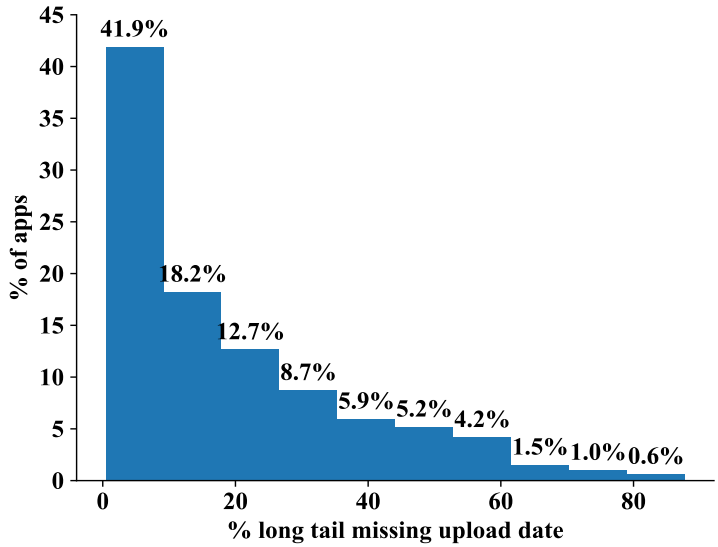

Figure 5.3: Distribution of apps missing upload dates

5.4.2 Review Classifier

A naïve way to identify SPR would be using keywords. However, this is not an easy task, since we cannot study millions of reviews to pick a representative keyword list. Besides, a review written by users can have multiple sentences. If we only use keywords to identify SPR, we may miss other information that comes from the nearby sentences that may contain interesting information but not the predefined keywords. Hence, by using machine learning techniques to learn not only the sentence with keywords but also the nearby ones we can expand our classifier's knowledge. For instance, consider the following review: *"Why do you need access to my location? Why on gods good green earth does your app need access to my location info? One star for the privacy steal."* If we would use keywords, we can only determine the first two sentences as security- or privacy-relevant. However, the last sentence is also an indicator that this app is perhaps doing something fishy. This is an important feature that we can put into a classifier without having to learn the phrase *privacy steal*. Later on, if our classifier encounters similar reviews, even without the presence of privacy-related keywords (here: location), it is still able to classify them as SPR (e.g., *"This app steals your info"*).

Training set Given the large amount of reviews and the anticipated low portion of SPR, it is not feasible to manually label a representative set of reviews while simultaneously balancing the number of SPR and non-SPR. Therefore, we first look at reviews that mention Android permissions or resources that are by default protected by an Android permission. We then manually examine some SPR to pick further keywords mentioned

in such SPR and visit the Android documentation regarding the mentioned permissions to further complement our keyword list with the information from the documentation. We strongly focus on permission-protected resources, because this is the only interaction that end-users can usually observe when they interact with the apps, e.g., install-time permission dialogs (prior to Android 6) or intercepting dialogs for runtime permission requests (Android 6 or later). It is rather uncommon to see layman users that are not security experts using extra analysis tools (e.g., *Xposed* modules [158]) to track data flows within applications for privacy violations or to detect insecure network connections of apps.

Table 5.1: Security- and privacy-relevant keywords

Permissions	**Key words**
Account	account access, account
Bluetooth	bluetooth, bluetooth devices
Calendar	read calendar, calendar, write calendar
Contact	read contacts data, write contact, contact
Location	location, track, gps
Mail	mail, voicemails
Media	picture, photo, media, files, take picture, taking picture, camera
Messages	sms, receive mms, send mms, messages, read messages, sms, read sms, send sms, mms, receive sms
Network	network, network state, wifi information, wifi, internet access, internet, network connectivity
Notification	notification, system alert window, system alert
Phone	phone call, phone number, outgoing call, manage call, phone state, call, call log, call's log, log, sip
Sensor	sensor data, sensor, fingerprint, nfc, vibrate
System	package size, install shortcut, delete package, battery info, reorder tasks, boot, boot completed, wap push, run in background, root
Storage	write storage, storage, read storage, sd card, SD card, file
General keywords	permission, access, intrusive, identity, personal info, malware, virus, malicious

Table 5.1 shows the list of compiled keywords we use in our analysis. This list results in approx. 1.85M reviews that are potentially security and privacy related. We randomly picked 4,000 reviews to manually label them. We consider a review as SPR if the user mentions the app's requested permissions, keywords related to accessed resources, or other general SPR keywords (see Table 5.1); otherwise we consider the review as non-SPR. After removing some malformed reviews (e.g., we were unable to determine what the reviews meant), our training set contained 3,891 reviews (SPR: 586, non-SPR: 3,305). To account for imbalanced data (SPR vs. non-SPR), we apply SMOTE [36] to

over-sample the SPR class.

Features extraction Characters of n-grams are commonly used features in text classification [33, 29, 85]. Character n-gram features for a review are all n consecutive letters in that review. For instance, the 5-grams for the review *"Why does this app need access to my location"* are *why d*, *hy do*, *y doe*, *does*, *oes t*, *es th*, [...] , *locat*, *ocati*, *cation*. We use n-grams of characters instead of words, because reviews written by users often contain typos, and by using n-grams of characters, we can reduce the influence of typos onto the classification. Prior work of McNamee et al. [109] showed that $n = 4$ (characters) is a good choice for European language text retrieval, while Dave et al. [47] reported that unigrams ($n = 1$) of words outperform bigrams when conducting text classification of movie reviews. Inspired by their findings, we choose $n = 3, 4, 5$ as our n-gram models, which also yielded the best results during experiments with our training data. Before extracting n-grams of the reviews, we apply different text pre-processing techniques to obtain a better quality data set since user reviews are often written on smart phones, hence they tend to be very short and usually contain grammatical mistakes or typos [82, 46, 44]:

- Remove stopwords: remove articles from the user reviews (e.g "a", "an", "the")
- Stemming: reduce inflectional forms to a common base form of a word (e.g. am, are is→ be)

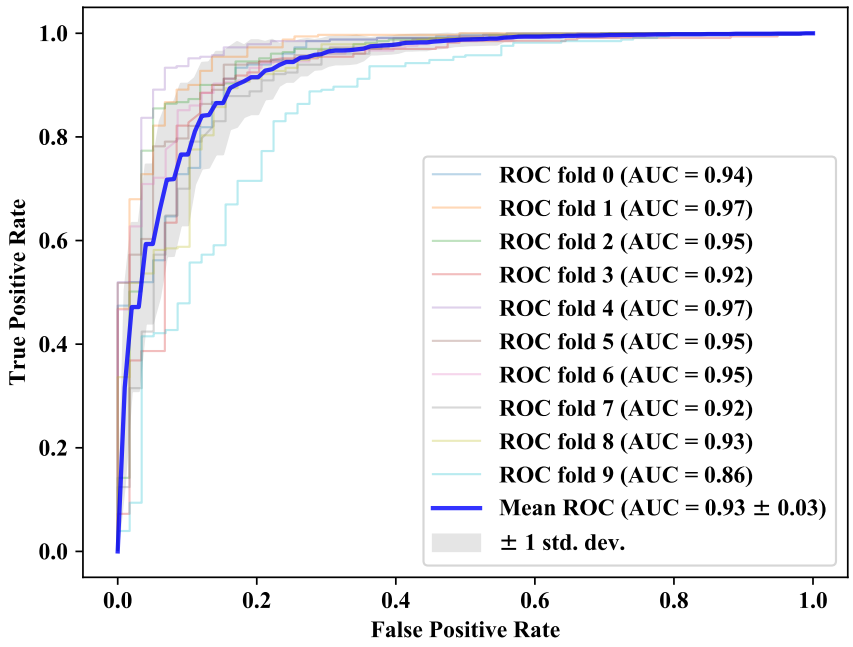

Figure 5.4: ROC curves of the 10-Fold cross-valication for our SPR classifier

Machine learning model Classifying reviews belongs to the task of natural language processing (NLP) and the most common NLP approach for using machine learning to classify text documents is using *Bag of Words* [143]. With bag of words, each text document is represented as the bag of its words regardless of its grammar forms and its orders. Occurrences of each word is used as feature for training classifiers. We use a Support Vector Machine (SVM) Linear kernel for our classifier as it has been shown to be effective for text classification [90, 148], especially for short documents [23]. We form bag of words by splitting the reviews at spaces and punctuation marks, and use n-gram model to extract features for our classification task.

Validation To validate our approach, we use k-Fold cross validation with $k = 10$, as prior work of Kohavi [95] has shown that this is the best method for cross validation. Besides, we choose AUC (area under the ROC curve [83]) as our classifier evaluation metric because it is not sensitive to imbalanced class distribution (SPR vs. non-SPR) and was widely used in prior work as the metric for imbalanced data classification [35, 129]. Figure 5.4 shows the AUC values for our 10-Fold cross validation. Our classifier has an AOC's mean value of 0.93 as its accuracy in classifying SPR (a classifier with perfect accuracy would have an AUC of 1.0).

5.4.3 Static App Analysis

So far we have built the data model that allows us to map SPR to the enclosing set of app versions by using both app version release dates and the date of the review. In order to measure the effect of an SPR on app security and privacy, we conduct static analysis on the version immediately preceding the SPR and the updated versions after the SPR to find potential SPU. For the majority of end-users the install-time permission list and runtime permission requests are the only information to assess whether the advertised functionality (e.g., via the app description, app category, etc.) seems legitimate. To determine the change of permissions and usage of APIs that require permissions across versions, we leverage the permission lists of the `axplorer` project [20]. Its authors provide mappings of Android SDK APIs to required permissions up to Android version 7.1. In a first step, we extract the list of declared permissions from the apps' manifest files. We further extract the target SDK versions to determine whether or not the app supports runtime permissions (target API higher or equal to 23). We subsequently scan the apps' bytecode for APIs that require dangerous permissions.

Attribution is another important aspect of the analysis, i.e., are permissions and their respective APIs used within the app developer code or within some third-party library code. In such cases, we would like to know the exact library (version). To this end, we leverage the open-source tool *LibScout* [19] that is capable of providing this information for a set of 205 commonly used libraries. To cover cases of unknown third-party code, we extend the implementation to classify any code not identified by LibScout into app or library code based on the app package name as a heuristic. We finally add functionality to attribute identified permission API calls to either app code or library code (either detected by LibScout or via our heuristic).

This collected information allows us in the following to identify security and privacy relevant changes as a potentially immediate result of an SPR.

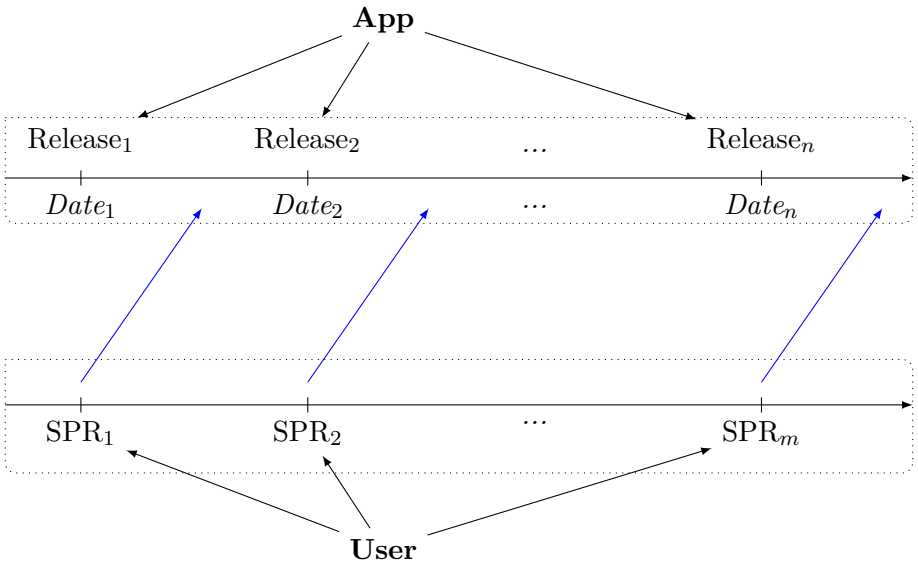

Figure 5.5: Mapping SPR to security-/privacy-relevant app updates (SPU)

5.4.4 Mapping SPR to SPU

The final step in our work-flow (see Figure 5.1) is to correlate the SPR for an app with the security and privacy related changes of an app. First, we identify potential candidate app versions that might contain relevant app changes in connection with an SPR, afterwards we analyze the candidate versions for security and privacy relevant updates (e.g., in the app manifest or code).

Identifying candidate app versions Figure 5.5 illustrates how we map SPR to candidate application updates. For every SPR, we first assign the SPR to the immediate preceding app version, *SPR app version*, released before the SPR. We then look for security and privacy related updates in later versions of the app after the *SPR app version*. In case we do not have the release date of an app version, we skip that version. When an SPU is found, this connection between the SPR and newly found SPU is considered a match (i.e, the SPR potentially influences the SPU).

SPR to SPU version distance While in ideal scenarios, we would expect SPU right after SPR, there are other factors which may contribute to the reasons why the next update may not be an SPU. For instance, developers are working on a particular feature of the app or they may only read user reviews irregularly (e.g., reviews come in large number [119]). We therefore take the distance between an SPR and an SPU release into account. In particular, if there is an SPR for version$_1$ but an SPU is found at version$_4$, then the distance is 3. The longer the distance is, the less likely the SPU is triggered by the SPR.

Table 5.2: Security- & privacy-related reviews per app category

Category (#apps)	Total #SPR	Mean #SPR/app
Tools (221)	1,343	7.5
Health And Fitness (30)	190	7.04
Shopping (35)	163	6.52
Sports (13)	54	6.0
Business (23)	113	5.95
Productivity (73)	364	5.69
Communication (66)	322	5.55
Media And Video (62)	192	5.33
Social (56)	215	5.12
Transportation (14)	42	4.67
Lifestyle (48)	136	4.53
News And Magazines (13)	47	4.27
Travel And Local (27)	89	4.05
Entertainment (98)	251	3.92
Personalization (112)	310	3.69
Finance (10)	25	3.57
Weather (19)	51	3.4
Photography (141)	228	3.3
Books And Reference (36)	73	3.04
Music And Audio (73)	144	2.94
Games (889)	1,149	2.78
Education (14)	22	2.44

5.5 Empirical Analysis

We present the results and findings of our analysis of security- and privacy-relevant reviews on Google Play and the corresponding relevant changes in app updates. We refer to Section 5.7 for a discussion of our findings.

5.5.1 Security and Privacy Related Reviews (SPR)

In order to analyze whether SPR trigger security and privacy relevant changes in app updates, it is first necessary to map SPR to features that can be checked with code analysis techniques. To this end, we first map SPR to permissions (groups) and subsequently check for permission-based features in app versions released after the SPR was posted.

Mentioned permissions Our review classifier identified 5,527 SPR (0.12% of a total of 4,547,493 reviews) belonging to 1,269 distinct apps. Each of these apps has an average of 4.36 SPR (median = 2), where certain app categories received more SPR (e.g., Tools) while others received less (e.g., Games). Table 5.2 lists the number of SPR per category. For 2,898/5,527 SPR, we are able to identify 4,180 permission-related statements that can be assigned to 15 distinct permission groups. This implies that some SPR refer to multiple permissions. The remaining 2,629 security and privacy related reviews cannot unambiguously be mapped to permissions without extra knowledge, e.g., *"Worked fine,*

but removing due to permission change without saying why...and if it is just for ads say that" and *"A Nice game, but ridiculous permissions the game is very good, but the permissions in the last update is ridiculous."*

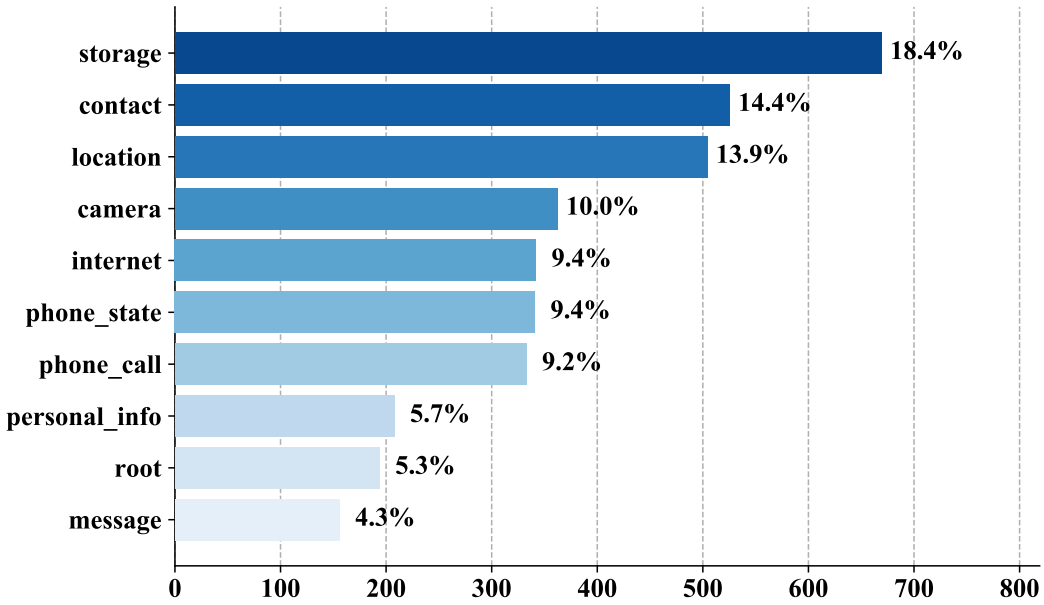

Figure 5.6: Ten most mentioned permissions in SPR

Figure 5.6 presents the permissions mentioned most in SPR. The list is headed by the permissions to access external storage, contacts and location. We created a separate category "personal information" for SPR when users complain about such data without mentioning specific permissions.

Runtime permissions vs. install-time permissions In October 2015, Google officially released Android 6.0 (API level 23) and shifted from an install-time permission model to a runtime permission delegation in which apps request dangerous permissions dynamically at runtime. For the 2,126 applications for which we built the version history, 1,073 (50.5%) have adopted runtime permissions in their latest version as of September 2017. Among the 1,269 apps that have at least one SPR, a similar fraction (49.7%) has adopted runtime permissions. To empirically investigate the effect of runtime permission requests on users' perception, we calculate the percentage of SPR over the total number of app reviews *before* and *after* an app adopted runtime permissions. To this end, we check whether the *targetSDK* argument from the apps' manifests is set to API level 23 or higher. We then conducted a t-test to compare the ratio of SPR per total reviews of app versions with runtime permission and with install-time permission. We found that there is significant difference between the SPR ratio of apps with install-time permission (mean = 0.001) and SPR ratio of apps with run-time permission (mean = 0.0025) with a p-value of 0.02. This suggests that apps with runtime permissions receive a significantly higher number of SPR.

Developer responses Some SPR are written by users that do not understand context-specific permission requests or do not have sufficient knowledge to assess the necessity of a request [62]. Incomplete or missing app descriptions that provide an intuition about the apps' permission usage is one contributing factor. To allow interaction with users, Google offers a *Reply to Reviews API* [134]. To analyze to which extent app developers make use of this feature, we crawl any developer replies that have been made to the set of 5,527 SPR. In total, we found 673 replies. With respect to the 5,527 SPR, developers also implicitly reacted in 3,359 cases with SPU in the subsequent app version. In 256 cases, the developer replied (without making SPU) and in 417 cases we could observe both replies and SPU. We manually examined these replies and grouped them into the following categories:

- **Explain** (397): Developers explain the necessity of the mentioned permissions
- **Contact** (130): Developers asked the user to contact them and to provide more information
- **Fix** (96): Developers confirmed the SPR and reported that a fix is already published or in progress.
- **Pre-defined generic** (50): Developers replied with pre-defined generic answer templates

In about 56% of the cases, the developer explained the necessity of permissions. Besides missing app descriptions, install-time permissions are one of the factors that make it difficult for the user to make the connection between permission and functionality. In our data set, the developers replied with explanations for 234 app versions that were using install-time permissions, in contrast to only 163 explanations for apps with runtime permissions. Oftentimes, the developers ask users to provide more information via mail. The reason for this is the 350 character limit imposed by Google for both reviews and replies. This severely impedes providing comprehensive and detailed information about a specific issue. In 96 cases the developer confirmed the user observation and reported that the issue has already been fixed or the fix is in progress. For 78/96 cases, we could identify SPU in the subsequent version of the respective app. For these cases, we can be very certain that the SPU has been an immediate effect of an SPR. In 50 cases, the developer simply replied with a pre-defined template without responding to details of the review.

Figure 5.7 gives three examples for typical scenarios where developers reply to SPR. In the first example, the developer acknowledges the issues and announces a fix without providing more details. In the second example, the developer explains the necessity of the location permission. In this case, no SPU are to be expected in subsequent versions. In the final example, the developer announces a switch to runtime permissions in a future version to provide more context for requests.

Acknowledge → Fix

USER: New permissions Why would this app need access to my location?
DEVELOPER: Fixed in version 2.6.2. Update will be available in some hours.

Explanation → No Update

USER: Why do you need access to my location? Why on gods good green earth does your app need access to my location info? One star for the privacy steal.
DEVELOPER: Location used for showing and maintenance ads only. App doesn't use/save/share your location.

Explanation → Update

USER: Why does it wait until you install it to tell you it's a trial Why does it need access to my photos and videos
DEVELOPER: Hello. We need an access to SD card only for the specific situations like saving configuration files etc. We are working on new version with runtime permissions support, so with upcoming version we will request the permissions only when it will be necessary. Team (*removed*).

Figure 5.7: Examples of user reviews and developers' responses

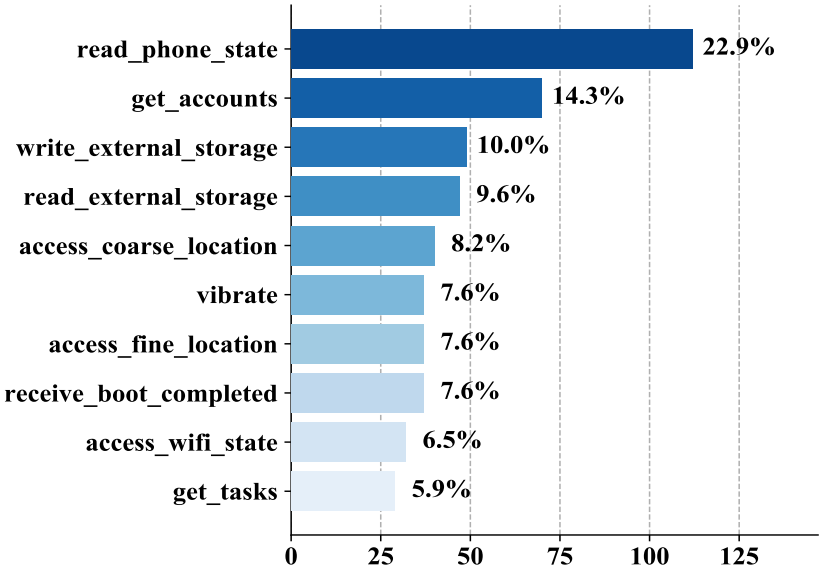

Figure 5.8: Top 10 permissions removed from application manifests.

5.5.2 Security and Privacy Relevant App Updates (SPU)

We consider changes in the permission usage of an app as SPU, since these changes would reflect what the user might perceive in terms of security and privacy. Our static analysis found the following SPU between all consecutive app releases in our data set:

- Requested permissions removed from app: 1,608

- Permission-protected API calls removed: 1,085
- Lib calls removed that trigger protected APIs: 940

In the following, we analyze those changes further.

Permission changes Figure 5.8 shows the top 10 permissions removed from the apps' manifests with app updates after an SPR was posted on Play. Reading the device's phone state, access to user accounts, and access to the external storage are the most frequently removed permissions, with external storage also being the top mentioned permission in SPR (see Section 5.5.1). The majority of removed permissions allow access to sensitive data, thus indicating a raised privacy awareness of users.

Figure 5.9 shows the top 10 permissions from permission-protected API calls that were removed from the apps. However, removing permission-protected calls does not necessarily mean that the app does no longer require that permission.

Change attribution We identify the root causes for the different results that we observe for removed permissions and permission APIs. An important aspect is statically included third-party code. Figure 5.10 lists the top 10 removed permissions required by permission-protected API calls triggered by calls to third-party libraries. Many of these permissions allow to retrieve data suitable for user tracking, which has been found in a variety of tracking and advertisement libraries [25]. An interesting case is the *WAKE_LOCK* permission. A study about wake lock misuse [103] showed that improper usage of this permission often manifests in battery drain, crashes, and app instabilities. Besides permission requests, these are events that can be observed by the user as well.

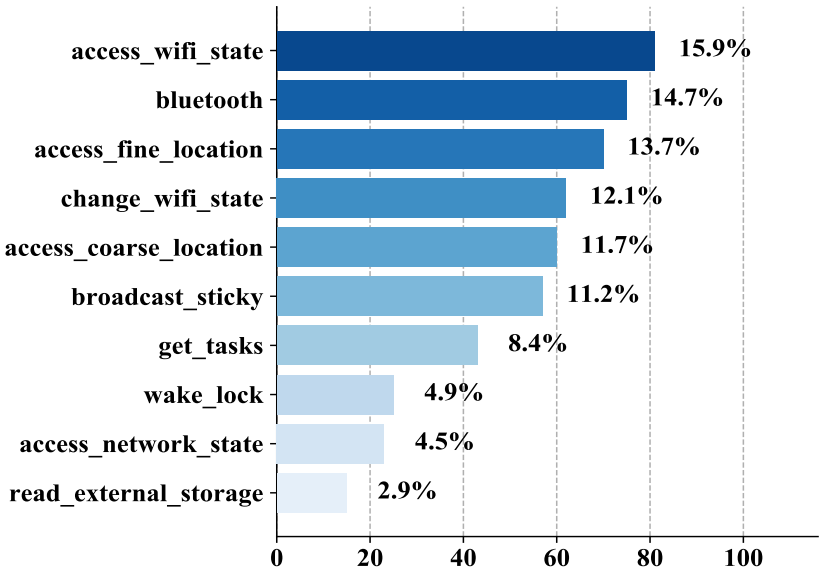

Figure 5.9: Top 10 permission-protected API calls removed from applications

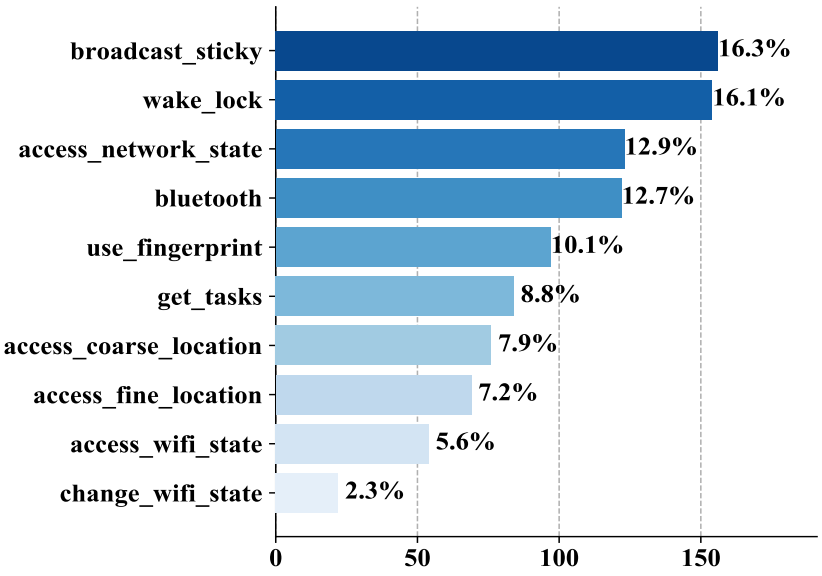

Figure 5.10: Top 10 permissions removed from third party libraries

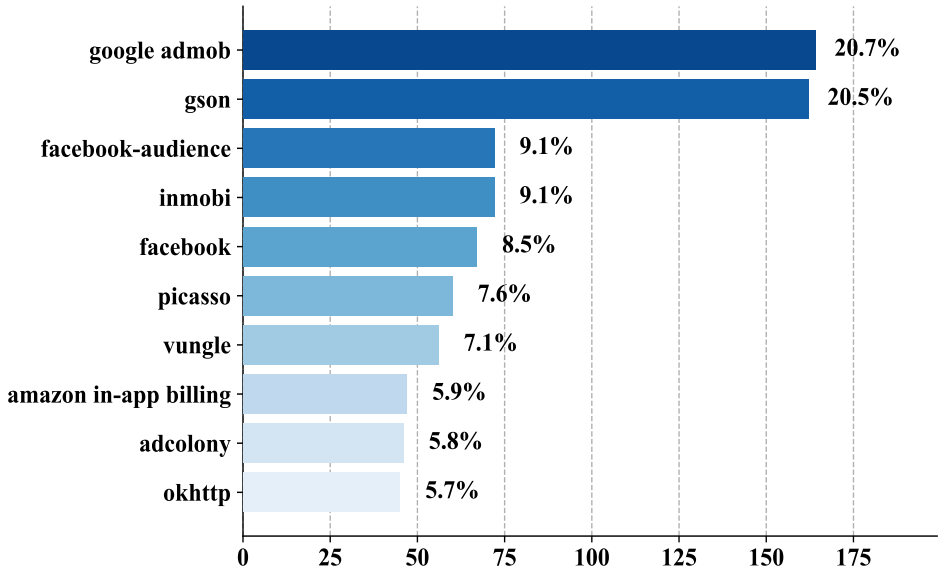

Figure 5.11: Top 10 libraries removed from apps (w/o Play and Support libs)

We also found that in 98.4% (925) of the cases the complete library was removed as part of the app update. In only 14 cases merely the library functionality that required the permission was removed. Figure 5.11 shows the frequently removed libraries without the extremely common *Play Service* and *Android support libraries*. Half of these libraries

constitute advertising libraries that require at least the *INTERNET* permission, typical uses often include *ACCESS_NETWORK_STATE* and location permissions as well. For apps that target install-time permissions, the *INTERNET* permission was frequently mentioned in SPR (228 instances), in particular when the app's core functionality, e.g., calculator or flashlight, obviously did not require network access. With Android 6, Google downgraded this permission to a *normal* protection level. As a consequence, it is granted automatically and no longer shown to the user by default. SPR complaining about the *INTERNET* permission for app versions targeting Android 6 or higher dropped to just 38 instances.

We further used the results of the analysis to attribute SPU to app developer and library code. To this end, we checked for each permission mentioned in the review whether the permission-protected API usage in the subsequent version is exclusively located in app code, library code, or used in both app and library. We found that in only 17% (72) of cases the permission-protected APIs were exclusively used by app code, while in 48.8% (206) of cases the APIs were exclusively used by library code. In the remaining 144 cases the permissions were used by both the application and the included libraries.

5.5.3 SPR to SPU Mapping

In this section, we report the results of mapping SPR to SPU and discuss factors that may influence the results.

Mapped SPR to SPU We are able to unambiguously map 4,898/5,527 SPR to the affected app versions. Only 629 SPR (11.38%) could not be mapped back to the affected versions, because the review was posted before 2012 for which we could not recover app upload dates. For 3,359/4,898 SPR (68.6%) we could map the SPR to SPU identified in one of the subsequent app versions. For the remaining 1,539 SPR, we could not detect security and privacy relevant changes in app updates. Figure 5.12 shows the distribution of the app version distance from SPR to detected SPU for the set of 3,359 SPR. In 76.8% of the cases we can observe SPU in the app version immediately following the SPR. If we consider the interval from one to five versions, this value increases to 94.4%. The likelihood that SPU in an app $version_x$ were triggered by the SPR gradually decreases with the number of new app versions released between SPR and $version_x$ and other external factors may have triggered the SPU instead.

SPU without SPR To evaluate to which extent our classifier misses SPR, we generate a backward mapping from SPU to reviews. For the 5,994 SPU our static code analysis found, 2,666 changes were observed without the presence of *any* review and 1,488 changes could be mapped to SPR. This leaves 1,840 SPU without SPR. Other reasons include external factors, such as updated libraries, internal code reviews, and developer notifications via different channels such as email.

SPR without SPU To further validate our approach, we also seek to find answers for the cases in which we identify SPR but no SPU in the subsequent app versions. When excluding the SPR that could successfully be mapped to SPU (3,359) and the SPR that

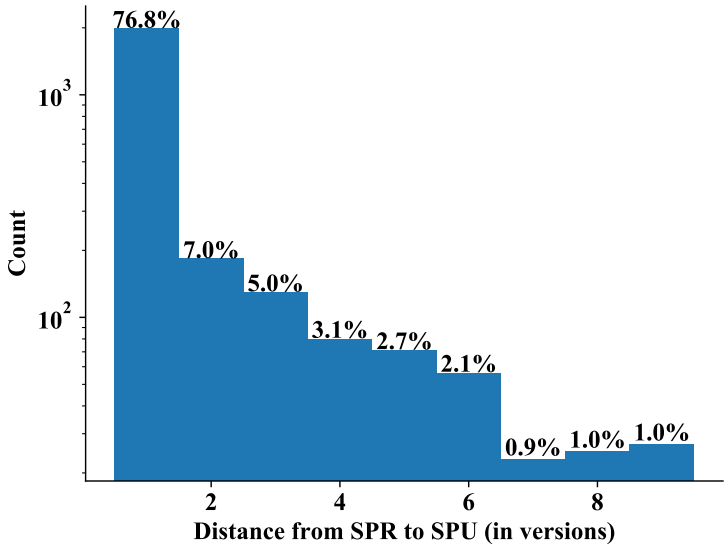

Figure 5.12: Distribution of distance to SPU for 3,359 SPR. Count in log-scale.

were posted prior to 2012 (629), this leaves 1,539 SPR for which we could not detect SPU. Reasons for this include: 1. Replies to reviews in which the developers explains the necessity of permissions or acknowledges the report without modifying the application. In these cases, no SPU are to be expected. 2. Limitations of our static analysis, e.g., LibScout can detect about 205 popular libraries, but there might exist more libraries that could have an effect on the security and privacy of an application. In addition, our analyzer checks for permission-protected APIs only. This misses permission-protected content providers such as *Contacts* for which an additional API argument analysis would be necessary. 3. The app is no longer maintained. To investigate why developers did not respond—neither with SPU nor replies—we check whether developers still update their apps after an SPR was posted. To this end we consider an app no longer maintained if its last version is older than one year starting from the day our crawler checked the latest version of the app). This threshold is reasonable since prior work [32, 49] has shown that most apps—in particular top apps—release updates biweekly or monthly. We found 728/1,539 (47.3%) SPR belonging to 244 unmaintained apps. Since these apps are still available on Play, they receive new reviews but new updates should not be expected.

5.5.4 Summary of Findings

We briefly summarize our findings from the empirical analysis. We first crawled 4.5M reviews for 2,583 distinct apps and identified 5,527 SP related reviews (SPR), out of which 2,898 SPR could be mapped to permissions. With our static code analysis we could identify 5,994 SPU in app versions following an SPR. In 60.8% of cases, we could successfully map SPR to SPU. If we consider corresponding SPU (changes to 3,359

SPR) and replies (273) as responses from developers and exclude SPR of unmaintained apps (728), we can calculate the developer response rate (RR) to SPR as follows:

$$\text{RR} = \frac{\#\text{SPR with SPU} + \#\text{SPR with replies}}{\#\text{SPR} - \#\text{SPR of unmaintained apps}} = 75.68\%$$

Despite a small overall number of SPR compared to the total number of reviews and the limited number of allowed text (350 characters), we can observe that these short texts are an effective means to trigger fast responses from developers. In almost 76% of cases, the app developer responded to a SPR.

5.6 Modeling Security and Privacy Updates

To examine the impact of different factors on Android application updates (SPU and non-SPU), we conducted multiple regression models that predict whether an app update will be security-/privacy-related or not. Our models include the effects of user reviews, user rating, app's permission mechanism (install-time or runtime), developer reply ratio, and app category. To account for possible effects of multiple updates of the same application, we use mixed models in which the updates are attributed to their application (i.e., nested data) and include random effects by allowing the intercepts to vary at application level but aggregating them over all applications. We compared a model with random effects against a model without random effects, and our results show that the model with random effects is significantly superior in its predictions.

Data set From the collected app history, static analysis results, and the identified SPR we built a data set of app updates (including both SPU and non-SPU). In this data set, we consider every change between two app version codes for which at least one review is available as a data point for our regression model, which yields 15,835 data points in total (12,540 non-SPU and 3,295 SPU) when excluding the "Comics" and "Libraries and Demo" categories since they only have one and three apps, respectively. We consider the following variables for every data point (i.e., app update) as predictors in our regression models:

- *SPR ratio*: ratio of SPR over the total number of reviews
- *Average score*: average rating score that the corresponding app version received since the last app update
- *Permission mechanism*: permission mechanism (runtime or install-time) used by the app version
- *App category*: as defined in Google Play
- *Reply ratio*: the ratio of developer replies over the total number of reviews since the previous app version

We consider SPR ratio and average score to be *user variables*, while permission mechanism, app category, and reply ratio are *app variables*. To account for SPR and average score that can potentially have an impact on later versions of the app but not the immediate version (see Section 5.5.3), we included the impact of SPR and average score of the previous versions within the *version distance* into the final SPR ratio and final

average score, respectively. The *version distance* between a version (version_i) that has the SPR ratio and average score, and the version that is being considered (version_j) is calculated by the number of versions between version_i and version_j for the analysis. Version_i is here a preceding version of version_j. The final SPR ratio and final average score of the currently being considered version are the cumulation of all of the previous SPR ratios divided by their corresponding *version distance*, and the cumulation of all previous average scores divided by their corresponding *version distance*, respectively.

5.6.1 Correlation Analysis

Since the coefficient estimates of mixed models can be unstable and difficult to interpret if the model has multicollinear variables, we first performed a correlation analysis of the independent variables, such as ratio of SPR over total reviews, reply ratio, and average score. The analysis showed that there is no significant multicollinearity between any variables of SPR ratio, reply ratio, average score in our data set. We did not include a variable that accounts for the SPU location, i.e, whether security and privacy issues mentioned in an SPR were located in application code or library code, since this variable is derived from SPRs, which would violate the requirement for regression analysis that predicting variables must be measured independently. In our data set, the location of those issues are completely dependent on SPRs, which is already represented by the *SPR ratio* variable. We therefore excluded such a location variable from our model.

5.6.2 Building the Models

To have a quality model, we need to only include variables that are necessary and can account for as much of the variance in the empirical data as possible. We start with a base model without any independent variables and then subsequently extend it with more predictors. Table 5.3 presents the goodness of fit for the relevant steps in building the corresponding models. Since the dependent variable of our analysis is binary—either an app update is SPU or non-SPU—we use logistic regression. Moreover, to verify that a mixed model suits our data better than a simple base model, we tested the base model without any independent variables against the mixed models. The result is that mixed models fit our data significantly better. In particular, we extend the base model as follows:

- Start with base model with a random effect to account for effects from updates of the same app
- Include variables at user level: SPR ratio, average score
- Include variables at app level: permission mechanism, app category, developer's reply ratio
- Include interaction between SPR ratio and average score

In each step, we calculated the model fit and used log likelihood model fit comparison to check whether the later model fits our data significantly better than the previous one. For the final model, we chose the one with the best fit that was significantly better in explaining our data than the previous. This is a well established approach for model selection [24, 43, 63, 86].

Table 5.3: Goodness of fit for the models predicting SPU. AIC = Akaike information criterion; Df = Degree of freedom; logLik = Log likelihood; Pr(>Chisq) quantifies statistical significance. Statistically significant variables are shaded.

	AIC	**logLik**	**Df**	**Pr(>Chisq)**
simple regression	16198.23	-8098.12		
mixed base regression	15654.05	-7825.03	1	<0.001
+ user variables	15570.16	-7781.08	2	<0.001
+ app variables	14830.06	-7388.03	23	<0.001
+ interaction	14831.54	-7387.77	1	0.471

We compared all models according to their corresponding *Akaike information criterion* (AIC), see Table 5.3, which estimates the relative quality of statistical models for a given set of data. Smaller AIC scores indicate a better fit. Moreover, we also used likelihood-ratio tests, which are evaluated using Chi-squared distribution, to compare the models.

From Table 5.3, we can see that the model with user and app variables and without interaction has the lowest AIC score (14830.06) and explains the data statistically significantly better than other models. For permission mechanisms and category, we choose *install-time* and *News And Magazines* category respectively as base lines for categorical variables: permission mechanism is a binary variable (either runtime or install time) and we want to see to which extent changing from install-time to runtime permission affects the interaction between user and Android application; and *News And Magazines* category's average number of reviews per app coincides with the global average number of reviews per app among all categories (see Table 5.2).

5.6.3 Results and Interpretation

Table 5.4 presents our regression model that examines the effect of different predictors for SPU. We can see that, in comparison to install-time permission mechanism, runtime permission dialogs have significantly positive impact on SPU (odds ratio of 3.9). This means that updates of applications (including app versions) whose permission mechanism is runtime are significantly more likely to be relevant to security and privacy. This further supports our earlier results that runtime permissions raise users' suspicions. Moreover, in comparison to apps of News And Magazines category, apps belonging to other categories do not differ significantly with regards to SPU. This indicates that SPU of an app seemingly do not depend on the app's category. Most importantly, we can see that SPR ratio is a significantly positive and strong predictor of SPU (odds ratio: 13.04). This indicates that the more SPR the app developers receive, the more likely they will release SPU. In contrast to SPR ratio, reply ratio has negative impact on SPU (odds ratio: 0.66), indicating that if developers reply to a review, the less likely the following app updates are security- and privacy-related. When we consider SPR, we see that most of the developers' replies are *Explanation* (see Section 5.5.1) why such permissions are needed. This is further supported by our regression model. We argue that if permission requests of Android apps are more transparent (e.g., better

Table 5.4: Logistic regression mixed model predicting SP changes. Statistically significant variables are highlighted. pm = Permission mechanism; cat = App category

	Estimate	**Std. Error**	**z value**	**Pr(>| z |)**
(Intercept)	-1.092	0.315	-3.465	<0.001
SPR ratio	2.568	0.796	3.225	0.001
avg_score	-0.094	0.006	-15.093	<0.001
reply_ratio	-0.420	0.157	-2.678	0.007
pm:run-time	1.360	0.054	25.276	<0.001
cat:Books_Reference	0.096	0.367	0.261	0.794
cat:Business	-0.039	0.374	-0.103	0.918
cat:Communication	0.098	0.347	0.281	0.779
cat:Education	-0.403	0.418	-0.965	0.335
cat:Entertainment	0.526	0.334	1.573	0.116
cat:Finance	0.012	0.462	0.026	0.980
cat:Games	0.601	0.316	1.903	0.057
cat:Health_Fitness	0.428	0.362	1.184	0.237
cat:Lifestyle	-0.105	0.346	-0.303	0.762
cat:Media_Video	-0.035	0.346	-0.102	0.919
cat:Music_Audio	0.324	0.339	0.956	0.339
cat:Personalization	0.047	0.331	0.141	0.888
cat:Photography	0.374	0.326	1.146	0.252
cat:Productivity	0.131	0.342	0.384	0.701
cat:Shopping	0.205	0.362	0.567	0.571
cat:Social	-0.043	0.341	-0.127	0.899
cat:Sports	-0.195	0.391	-0.498	0.618
cat:Tools	0.168	0.323	0.520	0.603
cat:Transportation	0.065	0.420	0.155	0.876
cat:Travel_Local	0.115	0.400	0.288	0.773
cat:Weather	0.216	0.392	0.551	0.581

explanation, request in context), users would understand why such requests are indeed reasonable, hence developers would not need to explain themselves in their replies. Finally, the average score has a negative impact on SPU. More precisely, if an app is receiving high scores (on average), then the next updates are less likely to be related to security/privacy (odds ratio 0.91)

5.7 Discussion

We discuss shortcomings of our approach and interpret our findings. Then, we highlight future work and a call for action.

5.7.1 Threats to Validity and Future Work

Our approach relies on the ability to map SPR back to app versions in order to measure possible app changes as reaction to user reviews. Similar to related work [19], we could not retarget the upload dates for all versions of our dataset. In particular, for app versions released before 2012 there exists no reliable third-party source that can be queried for upload dates. As a consequence, we failed to map 629/5,527 SPR (11.38%)

back to app versions and therefore cannot assess the impact of these SPR on the app's security and privacy.

Further, we use static code analysis to identify security and privacy related changes in app versions (immediately) following an SPR. This empirical evidence is a strong indicator that these changes have been made as a (direct) consequence of the SPR. Reasons for SPU range from following the principle of least privilege to protecting users' privacy to monetary reasons due to bad ratings and a decline in the number of app installations. For the small number of SPRs to which the developer replied and confirmed the issues, we can directly verify our findings. However, in general, collecting the ground truth would require conducting a developer survey to ask directly for the incentive of these changes. Prior studies [2, 49] have shown that recruiting a reasonable number of developers in Google Play for a survey is challenging without direct infrastructure support of the market (i.e., response rates <1%). We abstained from conducting a survey, as we only have a limited set of 2,583 top apps, with an even smaller number of distinct app developers and, hence, given prior experiences [2, 49], a too small expected number of responses.

Another improvement would include adding a sentiment analysis to our binary review classifier (SPR/non-SPR). This could help in understanding ambiguous SPR where users complain about requested permissions but still like the application or when users complain but are explicitly fine with a good explanation of the permission usage.

Lastly, we focused in our study on the top apps in Google Play, for which a higher level of maintenance and developer responsiveness to reviews would be expected. Our results might not apply to the long tail of apps on Play. However, since the top apps account for the bulk of the app downloads on Play [153, 161], our results apply to the apps with the highest impact on Android's user base.

5.7.2 The Effect of SPR

Previous work has not given much attention to the influence of end-users on security and privacy of apps via app reviews (see Section 5.8). Our results show that end-user complaints based on observable evidence (permissions, crashes, or anomalies like unusual battery drain) often lead to app changes that improve security and privacy aspects. In cases where the issues can be attributed to closed-source components (see Section 5.5.2), the developers might not even have been aware of these problems without an involved code analysis, e.g., when the library documentations miss important details.

User reviews can also force app developers to react quickly to issues due to the snowball effect. In many cases it is not a single SPR that triggers app changes but a series of SPR by different users (*SPR ratio* in regression model) or SPR followed by a series of follow-up reviews with low star ratings agreeing with the initial SPR (*avg_score* between app versions). Developers are then forced to react due to a fear of losing reputation (star rating) and user base that typically manifests in significant impact on revenue. As a result, developers either try to quickly resolve the problem by providing a better explanation to end-users or by addressing the issue with an SPU.

Although our results emphasize the positive effect of SPR in general, reviews could be much more informative and effective without the current size limitation of 350 characters for both reviews and replies imposed by Google. Such limits force users to omit important details in reviews and make them use alternative, unrestricted communication channels, such as email (see developer responses in Section 5.5.1). This also prevents comprehensive app reviews as we know it from consumer reviews for shops, such as Amazon. Although Google is aware of this problem for years [73], no improvements have been made to remedy the situation.

5.7.3 The Effect of Runtime Permissions

Permissions are one of the most important security and privacy indicators of apps that can be perceived even by less tech-savvy users. However, the way how permission requests are presented to the user greatly affects their effectiveness (e.g., habituation effects, user understanding, etc., see Section 5.8). The most drastic recent change in Android's permission system is the switch from install-time to runtime permissions, from which we can also observe a ripple effect onto users' reviews. Before Android 6, install-time permissions provided a one-time decision possibility without context. Without an explicit connection from permissions to functionality, users have to resort to (frequently missing or incomplete) app descriptions for permission decisions. With the introduction of runtime permissions, permission requests are (typically) shown in context and end-users may decide differently when the same request is displayed on different occasions. Further, developers have the possibility to augment permission requests with information to explain the necessity of a permission in a given situation. With the introduction of runtime permissions in Android 6, Google did also change the protection levels of a significant number of permissions. Before Android 6 (API level < 23) there have been 38 dangerous permissions that were prominently shown at install-time [20]. Starting with API level 23, Google refactored the permission system and specified only 20 dangerous permissions. The remaining 18 permissions have either been downgraded to normal permissions (that are granted automatically and are not shown to the user by default) or have been deprecated. Among the most prominent examples are the `INTERNET` permission, used by the vast majority of apps, and `READ|WRITE_PROFILE`. In addition, one single permission `READ_EXTERNAL_STORAGE` was upgraded to dangerous. This is also the top-mentioned permission in SPR (16%, see Figure 5.6).

Our regression model (see Section 5.6.3) suggests that applications adopting runtime permissions are significantly more likely to perform SPU compared to apps that stick to install-time permissions. But at the same time, the results indicate that for apps with runtime permissions there is still a high number of developer replies of type *Explanation* (163) in comparison to apps with install-time permissions (234). This suggests that many app developers do not follow runtime permission best practices [16], i.e., adding explanations for permission requests and requesting permissions in context rather than requesting permissions on app launch. In terms of transparency for the users, requiring the developers to add explanations in permission dialogs should be opt-out instead of opt-in by default. We think that our results support further investigation of how app developers use the runtime permissions.

Google recently announced that in the second half of 2018, Play will require new apps and app updates to adopt runtime permissions, i.e., to target an API level ≥ 23 [88]. This will likely allow end-user privacy assessments for a larger number of apps, i.e., in our dataset about 45% of the top apps have not adopted runtime permissions in their latest version. According to our results this will generate more SPR and, as a result, more SPU in apps.

5.7.4 User's Perception of Risks and Privacy Incidents

Compared to related work, our results from studying users' security- and privacy-related app reviews also suggest a change in the user's concerns over the last years. A large scale survey on the perceived risks of smartphone users by Felt et al. [127] indicated that sending premium messages, dialing premium numbers, and deleting contacts were among the top risks in 2012. Contacts are still in the Top 2 mentioned permissions (see Figure 5.6), but reading external storage and location—now the top perceived risks—have previously been among the lowest-ranked risks. This is partly because of additional security features that impede using monetary services without the user's explicit consent and due to raised privacy awareness of end-users. Past incidents have shown that simple flashlight [13, 156] or wallpaper apps [50] misused access rights to spy on the user or to exfiltrate personal data. As a result, Google specified both privacy policy for apps [130] and a general *Unwanted Software Policy* [74]. Developers were notified that, by end of March 2017, *"Google Play requires developers to provide a valid privacy policy when the app requests or handles sensitive user or device information."* In future work, we consider evaluating to which extent users' SPR can be used to create trend analyses of users' attitudes, in particular in response to regulatory (e.g., policies) and system changes (e.g., refactoring of permissions). For instance, in our data set, the downgrade of the *INTERNET* permission lead to a sharp decline in the number of SPR for that permission.

Moreover, a recent large-scale investigation of hidden tracking behavior in Android apps revealed that misuse of sensitive data by third-party advertisement and tracking libraries is even larger than ever [15]. In consequence, Google extended its *Unwanted Software Policy* and additionally requires that *"if an app collects and transmits personal data unrelated to the functionality of the app then, prior to collection and transmission, the app must prominently highlight how the user data will be used and have the user provide affirmative consent for such use."* [7] This implies that adhering to the new policy requires transparency for all included third-party components. Related work [25, 78, 144, 133] indicated that particularly advertising and tracking libraries are the main source of privacy violations and also our results (see Section 5.5.2) show that the majority of SPRs complain about behavior that apps inherited from included libraries. However, most of these third-party components are distributed as closed-source binaries and many are not explicit about their usage of permissions and end-user data. An open question will be how app developers can handle these problems, as this kind of libraries is often used as the main monetization factor. A related study [49] showed that app developers need more support in handling third-party libraries, both with better development tools and a dedicated package manager for libraries. Similar assistance will be necessary for

developers to make educated decisions on the choice of libraries to adhere to the new policy and pro-actively avoid negative SPR on markets.

5.7.5 Call for Action

The strict enforcement of the 350 character limit, prevents comprehensive and high quality reviews and forces users to omit additional information. Increasing the limit will give users the opportunity to write meaningful reviews and report issues without having to resort to alternative, non-public communication channels such as email. Both end-users and developers could also benefit from dedicated reviewer programs, such as *Amazon Vine* [10], which promote trusted reviewers that provide high-quality app assessments for incentives, such as paid apps for free. Particularly developers of top apps receive a high number of reviews every day, but only few of them include a call for action. Approaches such as ours—as standalone-tool or directly integrated into the developer console—can effectively reduce the number of reviews that have to be considered, hence making the time-consuming, manual triage process more effective. Additionally, the process of writing a review needs to be simplified to engage a higher number of users to participate. Currently, writing a review with a device constitutes a multi-step process via the Google Play app. This could be optimized by extending app launchers to provide such an option as an app shortcut [17] as a default for all apps hosted on Play. Moving to runtime permissions is a valuable step towards increasing the risk awareness. The latest beta version of Android P continues this path by disallowing idle apps to access the microphone and camera [89]. Any attempt is shown to the user as symbol in a notification. It has to be shown whether this is already effective or whether such accesses should be highlighted in the status bar more prominently.

5.8 Related Work

Android security, and in particular application security and the role of developers in the mobile ecosystem, have been studied from different angles in the past. To put our study on user reviews and their connection with the security and privacy evolution of apps into a larger context, we present and discuss in this section briefly related works on using natural language written texts for app classification, app reviews in general and their automatic processing, as well as closest related developments in app security.

Using natural language processing Past research has successfully mined software artifacts and connected them with the app descriptions regarding security and privacy aspects. For instance, Gorla et al. used the applications' descriptions to examine whether or not the description matches the applications' behavior [76]. The authors proposed *Chabada*, a solution to cluster apps by their topics based on their description, and to identify outliers, i.e., apps whose behavior deviates from the usage of permission protected APIs within each cluster. Further, Pandita et al. [121] proposed *Whyper* and Qu et al. [131] proposed *AutoCog*, two systems that also mine Android application descriptions and then use natural language processing (NLP) to automatically bridge the semantic gap between what applications do and what users expect them to do

from their description. All of them, Chabada, Whyper, and AutoCog, work on app descriptions written by developers. On the other hand, our study focuses on reviews written by *users*, which are usually authored on smart phones, and hence often contain typos and do not necessarily follow grammatical structures [82, 46, 44].

Processing app reviews App reviews play an important role for the success of an app. They are the primary channel through which developers receive feedback about their applications, such as how users perceive their apps, which features users are requesting, or which aspect of the apps users favor. By default, this channel is public and available to current and potential future users. However, inspecting such reviews is a challenging task for developers as apps receive a high number of reviews every day. Prior work by Pagano and Maalej [119] found that iOS apps receive on average about 22 reviews per day and popular apps, such as Facebook, receive magnitudes more reviews. Moreover, reviews are not easy to automatically analyze given their unstructured forms. Existing work by Chen et al. [38] has shown that only about one third of the user reviews are actually informative to developers. Different prior works have focused on automatically identifying useful user reviews for developers. Palomba et al. [120] proposed *ChangeAdvisor* to support app developers in classifying feedback useful for app maintenance. ChangeAdvisor combines NLP, text analysis, and sentiment analysis to automatically classify app reviews written by end users. Fu et al. [65] proposed *WisCom*, a tool that analyzes user comments and ratings in mobile app markets. WisCom uses regression models and latent dirichlet allocation models to analyze the comments' topics. It is able to discover inconsistencies in reviews and determine why users dislike a given app. However, none of these works focuses on the connection between app reviews and the application's security and privacy evolution.

App security evolution Calciati et al. [31] studied how the permissions requested by apps evolve across different app versions. Their results show that apps tend to request an increasing number of permissions in their evolution and many newly requested permissions are initially an over-privilege of the app (i.e., a direct violation of the least privilege principle). Violation of least-privilege by app developers is unfortunately a long-standing problem, first identified by Porter Felt et al. [125]. Given the central role of permissions for data protection on Android, past research has also investigated how users should be confronted with permission requests, most noticeably early studies by Porter Felt et al. [126, 127] that investigated users' concerns connected to permission protected resources and that gave different recommendations, respectively, which are partially reflected in a recent paradigm shift of Android's design from install-time to runtime permission delegation. More disruptive proposals try to eliminate the explicit role of the user for permission granting, e.g., through user-driven access control as proposed by Roesner et al. [135] or the use of machine learning as proposed by Wijesekera et al. [157] and Olejnik et al. [118]. Most recently, different works pointed out the risks of third party libraries, in particular of advertisement libraries [78, 144, 48, 141] and of vulnerable libraries [123, 19]. However, to the best of our knowledge, we are the first to study the connection between user reviews and Android application security and privacy evolution.

5.9 Conclusion

In this work, we empirically studied the impact of user reviews on Android application security and privacy features. We automatically classified reviews into SPR and non-SPR. We mapped SPR to mentioned app versions and conduct a static code analysis to extract security- and privacy-related code changes for these versions (SPU). We find that in 60.77% of all cases the SPR triggered an SPU. The majority of these changes can be attributed to (closed-source) third-party code like advertising or tracking libraries. Furthermore, we built a regression model to evaluate the impact of different factors on SPU. With our regression model, we showed that SPR are significant predictors for SPU. In the majority of cases, app developers directly change the respective app code or publicly reply to users and explain why certain permissions are required. We have further seen that the adoption of runtime permissions has a significant positive effect on users' privacy perception. With the announced enforcement of runtime permission adoption, the absolute number of SPR is likely to increase in the near future, which in turn will help to improve app security and privacy in general.

Our results make a call for action to further increase the transparency of apps to foster more SPR as way to increase privacy-friendly app behavior; but also call for better tools to support developers in adhering to privacy regulations and their users' privacy preferences. Lastly, our approach might inspire future research to employ user reviews as a way to measure the effects of changes in regulations or Android's design.

6

Conclusion

While existing works have identified many malicious apps that abuse end user's personal information [67, 159, 124, 104], there is an increasing number of benign apps that have security & problems created inadvertently by software developers [59, 60, 41, 51, 56]. Particularly, developers can be inexperienced, and might not possess enough security knowledge to write secure code. Especially, the lack of tool support during development phase makes it hard for them to adhere to security best practices [49]. More importantly, security is oftentimes a secondary concern to developers [3, 115]. In this dissertation, we presented a line of work that aims to improve the security & privacy of Android apps with the focus on software developers.

Our work focuses on two aspects of improving Android app's security & privacy namely (1) the feasibility of tool support toward securing apps, and (2) the incentives and motivation for developers to write more secure and privacy preserving code. In the first part, we investigated to which extent can tool support help developers write more secure code. To this end, we first developed FixDroid (see Chapter 3) as an Android Studio plugin to identify common security pitfalls in Android projects, and provides developers more secure alternatives during development phase. We tested FixDroid with 39 participants, our results showed that code delivered with the support of FixDroid contains significantly less security & privacy issues. In a subsequent step, we targeted Android apps with outdated and insecure third-party libraries. We were motivated by existing work which shows developers' wish for tool support to keep their project dependencies up-to-date to avoid security & privacy problems introduced by third-party libraries [49]. We then developed Up2Dep (see Chapter 4), a static analysis tool that analyzes third-party libraries to find updatability and security information. The results are delivered to developers inside their development tool to inform about and guide them on the security and updatability of the libraries included in their projects. In our in-the-wild evaluation with developers (N=56), Up2Dep showed its impact by fixing not only outdated dependencies (n=108) but also insecure dependencies (n=8). With both FixDroid and Up2Dep we have shown that tool support indeed can help developers write more secure code. More importantly, there are plenty of room for future work to tackle security & privacy problems in Android apps from tool support perspective i.e., preventative approaches that help avoid security & privacy problems during development phase. In the final part of this thesis, we aimed to investigate factors that would create incentives and motivation for developers to develop more secure Android applications (see Chapter 5). We then set to measure the impact of user reviews on app security & privacy evolution by combining static analysis, natural language processing, and regression analysis techniques. Our results showed that security & privacy related reviews do have a positive impact on the security & privacy evolution of Android apps i.e., developers do consider user complains regarding their app's security & privacy problems, and update accordingly. From our results, we identified a clear call for action to not only support end users in making better choice with regarding to app security & privacy, but also in make app behavior more transparent to create incentives and motivation for developers to adhere to privacy best practices. At the same time, developers also need tool support to make their app behavior more user friendly, especially in estimating the security & privacy impact of third-party code onto their app's behavior.

This dissertation provides the first insights on the feasibility of tool support for developers during development phase to write more secure & privacy preserving code. Further, it shows that developers can indeed be motivated to adhere to security & privacy best practices using reviews of users to their apps. To increase the security & privacy of apps, it is necessary to (1) further investigate different aspects of developer's tool support and (2) identify factors that creates incentives for developers to follow security & privacy best practices.

Future Research Directions Integrating similar features like FixDroid's into developer's IDE is a necessary step to move forward in order to improve app's security and privacy. Specifically, IDE vendors like Google, JetBrains (i.e., with Android Studio) could consider providing developers effective security warning with more secure alternatives inside development environment to reduce the security & privacy problems, especially those that could be easily detected and avoided during development phase. Further, the author of this dissertation envisions a broader range of tool support for developers on programming tasks that are related to security & privacy. Future work could not only help developers write more secure code, but also further identifies situations where security is needed. Besides, learning code's (in)security from large code base (e.g., StackOverflow) could be a potential direction to provide developers help by means of tool support during development phase. It is also worth studying the long term impact of such tool support with developers e.g., how developer's behavior, and habit change overtime. Besides, future work could further study the design, work-flow, and the interaction with developers of security tool support to best help developers in their daily-programming tasks.

With Up2Dep, this dissertation shows that analyzing cryptographic related problems and providing developers corresponding warnings would not be effective if such warnings are not comprehensible which can result in no action from developers. In the developer study with Up2Dep, not all developers could understand the warning messages about cryptographic API misuse. This suggests that future work needs to make cryptographic related warning messages more developer-friendly, e.g., make it easier to understand (similar to other domains such as browser security warnings [8, 61]). Besides, to motivate developers to take actions, technical warning messages that describe such security & privacy related problems are not enough. Future work could investigate on how to effectively notify developers about security & privacy problems e.g., by providing developers *context-aware* warning messages that comes with the consequence (in the corresponding scenarios of the app) of the security problems. Further, since many developers are not aware of the security & privacy implication of the libraries that they include in their apps. Analyzing third-party libraries and providing developers with security & privacy related information on the library's API that developers are using would avoid unwanted security & privacy implications. Up2Dep has focused on cryptographic API misuse and publicly disclosed vulnerability. Yet, one of the interesting area for future work to look at is how third-party libraries use permission protected API which has been shown to be a significant source of security & privacy issues in Android apps. Besides, the publicly disclosed vulnerabilities were currently manually added to the database of Up2Dep, an envisioned approach is to automatically

connect central repository of vulnerabilities and notify developers when the included third-party libraries are vulnerable.

Future work could also investigate ways to improve the recruitment methodology to involve actual developers with studies. Currently, the process of recruiting professional developers to studies takes a lot of time, and is not efficient. Existing work used to crawl contact information from market stores and send emails to developers [5, 49, 77]. This resulted in a very limited response rate. With Up2Dep, developers were contacted and recruited directly at conferences and via different social media platforms. However, the results showed that it is not possible to scale up to a larger population of developers. A promising direction for future work to look at is to investigate a platform for recruiting developers for studies. This would significantly facilitate the tasks recruiting suitable developers for behavioral studies, as well as for testing new tools to see how developers would receive such tools. Such a version of the platform can work in a similar way to Amazon Mechanical Turk [9], yet for professional developers.

Finally, this dissertation shows, end users do care about the security & privacy of their apps, and they express their concerns in form of app reviews about app security & privacy related behavior. Most importantly, developers actually listen to end users and take actions accordingly. This shows user reviews can motivate developers to adhere to security & privacy best practices. Hence, further exploring app behaviors beyond permission usage, and showing the end users what such apps do with their security & privacy would be beneficial to the security & privacy of the Android ecosystem. App stores on one hand side could provide developers automated systems to manage security & privacy related reviews which would ease the task of handling such reviews and hence would increase app's security & privacy. On the other hand, app stores could learn from behavior of apps that have a non-negligible amount of SPR to examine future apps with similar behaviors during their vetting process before making these apps available to end users. Further, user reviews on market stores are currently written in form of plain text with a rating score. One promising direction could be extending such reviewing mechanism to foster security & privacy related reviews e.g., providing in-app review option that adds context in which a security & privacy related behavior deems questionable.

Bibliography

Author's Papers for this Thesis

[P1] Nguyen, D. C., Wermke, D., Acar, Y., Backes, M., Weir, C., and Fahl, S. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. ACM, 2017.

[P2] Nguyen, D. C., Derr, E., Backes, M., and Bugiel, S. Up2Dep: Android Tool Support to Fix Insecure Code Dependencies. In: *Proceedings of the 36th Annual Computer Security Applications Conference*. ACSAC '20. Association for Computing Machinery, 2020.

[P3] Nguyen, D. C., Derr, E., Backes, M., and Bugiel, S. Short Text, Large Effect: Measuring the Impact of User Reviews on Android App Security & Privacy. In: *2019 IEEE Symposium on Security and Privacy (SP'19)*. 2019.

Further Publications of the Author

[S1] Nguyen, T. T., Nguyen, D. C., Schilling, M., Wang, G., and Backes, M. Measuring User Perception for Detecting Unexpected Access to Sensitive Resource in Mobile Apps. In: *Proceedings of the 16th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '21. Association for Computing Machinery, 2021.

[S2] Stransky, C., Acar, Y., Nguyen, D. C., Wermke, D., Kim, D., Redmiles, E. M., Backes, M., Garfinkel, S. L., Mazurek, M. L., and Fahl, S. Lessons Learned from Using an Online Platform to Conduct Large-Scale, Online Controlled Security Experiments with Software Developers. In: *10th USENIX Workshop on Cyber Security Experimentation and Test, CSET 2017, Vancouver, BC, Canada, August 14, 2017*. 2017.

References

[1] *2016 State of the Software Supply Chain*. `https://www.sonatype.com/software-supply-chain`. 2016.

[2] Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M. L., and Stransky, C. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In: *Proc. 37th IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 2016.

[3] Acar, Y., Fahl, S., and Mazurek, M. L. You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users. In: *2016 IEEE Cybersecurity Development (SecDev)*. 2016.

[4] Acar, Y., Backes, M., Bugiel, S., Fahl, S., McDaniel, P., and Smith, M. SoK: Lessons Learned From Android Security Research For Appified Software Platforms. In: *Proceedings of the 2016 IEEE Symposium on Security and Privacy*. SP '16. 2016.

[5] Acar, Y., Backes, M., Fahl, S., Garfinkel, S., Kim, D., Mazurek, M. L., and Stransky, C. Comparing the usability of cryptographic apis. In: *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE. 2017.

[6] Acar, Y., Backes, M., Fahl, S., Kim, D., L. Mazurek, M., and Stransky, C. You Get Where You're Looking For: The Impact Of Information Sources On Code Security. In: *Proceedings of the 2016 IEEE Symposium on Security and Privacy*. SP '16. 2016.

[7] *Additional protections by Safe Browsing for Android users*. https://security.googleblog.com/2017/12/additional-protections-by-safe-browsing.html. 2017.

[8] Akhawe, D. and Felt, A. P. Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness. In: *USENIX security symposium 2013*. Vol. 13. 2013.

[9] *Amazon Mechanical Turk*. https://www.mturk.com/. 2020.

[10] *Amazon Vine*. https://www.amazon.com/gp/vine/help. 2020.

[11] Android Developer Documentation. *App security improvement program*. https://developer.android.com/google/play/asi. 2019.

[12] Android Developer Documentation. *Shrink, obfuscate, and optimize your app*. https://developer.android.com/studio/build/shrink-code. 2019.

[13] *Android flashlight app tracks users via GPS, FTC says hold on*. https://www.techrepublic.com/blog/it-security/why-does-an-android-flashlight-app-need-gps-permission/. 2013.

[14] *Android Studio 1.0*. https://android-developers.googleblog.com/2014/12/android-studio-10.html. 2020.

[15] *Android Users: To Avoid Malware, Try the F-Droid App Store*. https://www.wired.com/story/android-users-to-avoid-malware-ditch-googles-app-store/. 2018.

[16] *App Permissions Best Practices*. https://developer.android.com/training/permissions/usage-notes.html. 2020.

[17] *App Shortcuts*. `https://developer.android.com/guide/topics/ui/shortcuts.html`. 2020.

[18] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49.6 (2014), 259–269.

[19] Backes, M., Bugiel, S., and Derr, E. Reliable Third-Party Library Detection in Android and Its Security Applications. In: *Proc. 23rd ACM Conference on Computer and Communication Security (CCS'16)*. ACM, 2016.

[20] Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., and Weisgerber, S. On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. In: *Proc. 25th USENIX Security Symposium (SEC'16)*. USENIX Association, 2016.

[21] Balebako, R., Marsh, A., Lin, J., and Hong, J. The Privacy and Security Behaviors of Smartphone App Developers. In: *Workshop on Usable Security (USEC'14)*. 2014.

[22] Bangor, A., Kortum, P., and Miller, J. Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of usability studies*, 4.3 (2009), 114–123.

[23] Basu, A., Walters, C., and Shepherd, M. Support vector machines for text categorization. In: *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*. IEEE. 2003.

[24] Bates, D., Maechler, M., Bolker, B., Walker, S., et al. lme4: Linear mixed-effects models using Eigen and S4. *R package version*, 1.7 (2014), 1–23.

[25] Book, T., Pridgen, A., and Wallach, D. S. Longitudinal Analysis of Android Ad Library Permissions. In: *MoST'13*. IEEE, 2013.

[26] Book, T., Pridgen, A., and Wallach, D. S. Longitudinal Analysis of Android Ad Library Permissions. *CoRR*, abs/1303.0857 (2013).

[27] Brady, P. *Anatomy & Physiology of an Android (Google I/O Session Videos and Slides)*.

[28] Brooke, J. et al. SUS-A quick and dirty usability scale. *Usability evaluation in industry*, 189.194 (1996), 4–7.

[29] Brown, P. F., Desouza, P. V., Mercer, R. L., Pietra, V. J. D., and Lai, J. C. Class-based n-gram models of natural language. *Computational linguistics*, 18.4 (1992), 467–479.

[30] Burnham, K. P. Multimodel Inference: Understanding AIC and BIC in Model Selection. *Sociological Methods & Research*, 33.2 (2004), 261–304.

[31] Calciati, P. and Gorla, A. How do apps evolve in their permission requests?: a preliminary study. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press. 2017.

[32] Carbunar, B. and Potharaju, R. A longitudinal study of the Google app market. In: *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015*. ACM. 2015.

[33] Cavnar, W. B. and Trenkle, J. M. N-Gram-Based Text Categorization. In: *In Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*. 1994.

[34] Chatzikonstantinou, A., Group, M., Ntantogian, C., Xenakis, C., and Karopoulos, G. Evaluation of Cryptography Usage in Android Applications. In: 2015.

[35] Chawla, N. V. Data mining for imbalanced datasets: An overview. In: *Data mining and knowledge discovery handbook*. Springer, 2009.

[36] Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. SMOTE: Synthetic Minority Over-sampling Technique. *J. Artif. Int. Res.* 16.1 (2002), 321–357.

[37] Chen, E. Y., Pei, Y., Chen, S., Tian, Y., Kotcher, R., and Tague, P. OAuth Demystified for Mobile Application Developers. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Association for Computing Machinery, 2014.

[38] Chen, N., Lin, J., Hoi, S. C. H., Xiao, X., and Zhang, B. AR-miner: Mining Informative Reviews for Developers from Mobile App Marketplace. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. ACM, 2014.

[39] Chin, E. M. Helping Developers Construct Secure Mobile Applications. In: UC Berkeley: Computer Science, 2013.

[40] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing Inter-application Communication in Android. In: *Proc. 9th International Conference on Mobile Systems, Applications, and Services (MobiSys'11)*. ACM, 2011.

[41] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in Android. In: *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM. 2011.

[42] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing inter-application communication in Android. In: *MobiSys'11*. ACM, 2011.

[43] Christensen, R. H. B. ordinal—regression models for ordinal data. *R package version*, 22 (2010).

[44] Ciurumelea, A., Schaufelbühl, A., Panichella, S., and Gall, H. C. Analyzing reviews and code of mobile apps for better release planning. In: *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE. 2017.

[45] *Cognicrypt Crypto API rules*. `https://github.com/CROSSINGTUD/Crypto-API-Rules`. 2019.

[46] Dalal, M. K. and Zaveri, M. A. Opinion Mining from Online User Reviews Using Fuzzy Linguistic Hedges. *Appl. Comp. Intell. Soft Comput.* 2014 (2014), 2:2–2:2.

[47] Dave, K., Lawrence, S., and Pennock, D. M. Mining the Peanut Gallery: Opinion Extraction and Semantic Classification of Product Reviews. In: *Proceedings of the 12th International Conference on World Wide Web*. WWW '03. ACM, 2003.

[48] Demetriou, S., Merrill, W., Yang, W., Zhang, A., and Gunter, C. A. Free for All! Assessing User Data Exposure to Advertising Libraries on Android. In: *Proc. 23rd Annual Network & Distributed System Security Symposium (NDSS '16)*. The Internet Society, 2016.

[49] Derr, E., Bugiel, S., Fahl, S., Acar, Y., and Backes, M. 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*. CCS '17. ACM, 2017.

[50] *Does an Android Live Wall Paper really requires any Special Permission? Beware of Android App Permissions*. `http://initpage.com/post/Does-an-Android-Live-Wall-Paper-really-requires-any-Special-Permission-Beware-of-Android-App-Permissions`. 2013.

[51] Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. An Empirical Study of Cryptographic Misuse in Android Applications. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. ACM, 2013.

[52] Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. An Empirical Study of Cryptographic Misuse in Android Applications. In: *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, 2013.

[53] Egele, M., Brumley, D., Fratantonio, Y., and Kruegel, C. An empirical study of cryptographic misuse in android applications. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013.

[54] Egelman, S., Cranor, L. F., and Hong, J. You've been warned: an empirical study of the effectiveness of web browser phishing warnings. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2008.

[55] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32.2 (2014), 5.

[56] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A Study of Android Application Security. In: *Proceedings of the 20th USENIX Conference on Security*. SEC'11. USENIX Association, 2011.

[57] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A Study of Android Application Security. In: *Proc. 20th Usenix Security Symposium (SEC'11)*. USENIX Association, 2011.

[58] *F-Droid App Repository*. https://f-droid.org/en/. 2019.

[59] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., and Smith, M. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In: *Proc. 19th ACM Conference on Computer and Communication Security (CCS '12)*. ACM, 2012.

[60] Fahl, S., Harbach, M., Perl, H., Koetter, M., and Smith, M. Rethinking SSL Development in an Appified World. In: *Proc. 20th ACM Conference on Computer and Communication Security (CCS '13)*. ACM, 2013.

[61] Felt, A. P., Ainslie, A., Reeder, R. W., Consolvo, S., Thyagaraja, S., Bettes, A., Harris, H., and Grimes, J. Improving SSL Warnings: Comprehension and Adherence. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI '15. ACM, 2015.

[62] Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. Android Permissions: User Attention, Comprehension, and Behavior. In: *Proceedings of the Eighth Symposium on Usable Privacy and Security*. SOUPS '12. ACM, 2012.

[63] Field, A. *Discovering statistics using IBM SPSS statistics*. sage, 2013.

[64] Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., and Fahl, S. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In: *Symposium on Security and Privacy*. Oakland'17. IEEE. 2017.

[65] Fu, B., Lin, J., Li, L., Faloutsos, C., Hong, J., and Sadeh, N. Why People Hate Your App: Making Sense of User Feedback in a Mobile App Store. In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '13. ACM, 2013.

[66] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and Shmatikov, V. The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software. In: *Proc. 19th ACM Conference on Computer and Communication Security (CCS '12)*. ACM, 2012.

[67] Gibler, C., Crussell, J., Erickson, J., and Chen, H. AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. In: *International Conference on Trust and Trustworthy Computing*. Springer. 2012.

[68] GitHub Help. *Viewing and updating vulnerable dependencies in your repository*. https://help.github.com/articles/viewing-and-updating-vulnerable-dependencies-in-your-repository/. 2019.

[69] Google. *Google Play Protect*.

[70] Google. *Improve your code with lint checks*.

[71] Google Help. *How to fix apps containing Libpng Vulnerability*. https://support.google.com/faqs/answer/7011127?hl=en. 2019.

[72] *Google I/O 2017: Empowering developers to build the best experiences across platforms*. https://android-developers.googleblog.com/2017/05/google-io-2017-empowering-developers-to.html. 2020.

[73] *Google Play Help Forum – Topic: Increase word count for reviews*. https://support.google.com/googleplay/forum/AAAA8CVOtD8OTjIUVr4g9I/?hl=en&gpf=#!topic/play/OTjIUVr4g9I. 2018.

[74] *Google's Unwanted Software Policy*. https://www.google.com/about/unwanted-software-policy.html. 2015.

[75] Gordon, R. A. and Arvey, R. D. Age Bias in Laboratory and Field Settings: A Meta-Analytic Investigation1. *Journal of applied social psychology*, 34.3 (2004), 468–492.

[76] Gorla, A., Tavecchia, I., Gross, F., and Zeller, A. Checking App Behavior Against App Descriptions. In: *ICSE'14: Proceedings of the 36th International Conference on Software Engineering*. 2014.

[77] Gorski, P. L., Iacono, L. L., Wermke, D., Stransky, C., Möller, S., Acar, Y., and Fahl, S. Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse. In: *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*. 2018.

[78] Grace, M., Zhou, W., Jiang, X., and Sadeghi, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In: *WISEC'12*. ACM, 2012.

[79] Gradle. *Gradle Transitive Dependency*. https://docs.gradle.org/5.6.2/userguide/managing_transitive_dependencies.html. 2019.

[80] *Gradle Build Tool*. https://gradle.org/. 2018.

[81] Group, N. N. *Field Studies*. https://www.nngroup.com/articles/field-studies/. Last visited: 12/09/2016.

[82] Gu, X. and Kim, S. "What Parts of Your Apps Are Loved by Users?" (T). In: *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ASE '15. IEEE Computer Society, 2015.

[83] Hanley, J. A. and McNeil, B. J. The meaning and use of the area under a receiver operating characteristic (ROC) curve. *Radiology*, 143.1 (1982), 29–36.

[84] Holzmann, G. J. Cobra: a light-weight tool for static and dynamic program analysis. *Innovations in Systems and Software Engineering* (2016), 1–15.

[85] Houvardas, J. and Stamatatos, E. N-gram feature selection for authorship identification. In: *International Conference on Artificial Intelligence: Methodology, Systems, and Applications*. Springer. 2006.

[86] Hox, J. J., Moerbeek, M., and Schoot, R. van de. *Multilevel analysis: Techniques and applications*. Routledge, 2010.

[87] Huang, J., Jr., N. P. B., Bugiel, S., and Backes, M. Up-To-Crash: Evaluating Third-Party Library Updatability on Android. In: *4th IEEE European Symposium on Security and Privacy*. 2019.

[88] *Improving app security and performance on Google Play for years to come*. https://android-developers.googleblog.com/2017/12/improving-app-security-and-performance.html. 2017.

[89] *In Android P, notifications for apps running in the background show if they're using the camera or microphone*. https://www.androidpolice.com/2018/05/09/android-p-notifications-apps-running-background-show-theyre-using-camera-microphone. 2019.

[90] Joachims, T. Text categorization with support vector machines: Learning with many relevant features. In: *European conference on machine learning*. Springer. 1998.

[91] Johnson, S. C. Lint, a C Program Checker. In: *COMP. SCI. TECH. REP.* 1978.

[92] Jovanovic, N., Kruegel, C., and Kirda, E. Pixy: A static analysis tool for detecting web application vulnerabilities. In: *Security and Privacy, 2006 IEEE Symposium on*. IEEE. 2006.

[93] Kaliski, B. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. United States, 2000.

[94] Kim, M., Bergman, L., Lau, T., and Notkin, D. An ethnographic study of copy and paste programming practices in OOPL. In: *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*. IEEE. 2004.

[95] Kohavi, R. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'95. Morgan Kaufmann Publishers Inc., 1995.

[96] Krüger, S., Nadi, S., Reif, M., Ali, K., Mezini, M., Bodden, E., Göpfert, F., Günther, F., Weinert, C., Demmler, D., and Kamath, R. CogniCrypt: Supporting Developers in Using Cryptography. In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*. ASE 2017. IEEE Press, 2017.

[97] Krüger, S., Späth, J., Ali, K., Bodden, E., and Mezini, M. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[98] Lam, P., Bodden, E., Lhoták, O., and Hendren, L. The Soot framework for Java program analysis: a retrospective. In: *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*. Vol. 15. 2011.

[99] Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., and Kirda, E. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In: *24th Annual Network and Distributed System*

Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017. 2017.

[100] Li, T., Agarwal, Y., and Hong, J. I. Coconut: An IDE plugin for developing privacy-friendly apps. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2.4 (2018), 178.

[101] *LibScout*. `https://github.com/reddr/LibScout`. 2019.

[102] *Lint Tool*. `http://tools.android.com/tips/lint`. 2018.

[103] Liu, Y., Xu, C., Cheung, S.-C., and Terragni, V. Understanding and Detecting Wake Lock Misuses for Android Applications. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. ACM, 2016.

[104] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. ACM, 2012.

[105] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012.

[106] Luo, T., Hao, H., Du, W., Wang, Y., and Yin, H. Attacks on WebView in the Android system. In: *Proc. 27th Annual Computer Security Applications Conference (ACSAC'11)*. ACM, 2011.

[107] Madiha Tabassum, Stacey Watson, and Heather Richter Lipford. Comparing Educational Approaches to Secure programming: Tool vs. TA. In: *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*. USENIX Association, 2017.

[108] Martin Georgiev Suman Jana, V. S. Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks. In: *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society, 2014.

[109] McNamee, P. and Mayfield, J. Character N-Gram Tokenization for European Language Text Retrieval. *Information Retrieval*, 7.1 (2004), 73–97.

[110] *Measuring U - Quantifying user experience*. `https://measuringu.com/`. 2019.

[111] Mohan, V. and Othmane, L. B. Secdevops: Is it a marketing buzzword?-mapping research on security in devops. In: *2016 11th International Conference on Availability, Reliability and Security (ARES)*. IEEE. 2016.

[112] Mohan, V., Othmane, L. ben, and Kres, A. BP: security concerns and best practices for automation of software deployment processes: an industrial case study. In: *2018 IEEE Cybersecurity Development (SecDev)*. IEEE. 2018.

[113] Muslukhov, I., Boshmaf, Y., and Beznosov, K. Source Attribution of Cryptographic API Misuse in Android Applications. In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ASIACCS '18. ACM, 2018.

[114] Mutchler, P., Doupé, A., Mitchell, J., Kruegel, C., and Vigna, G. A Large-Scale Study of Mobile Web App Security. In: *Proc. 2015 Mobile Security Technologies Workshop (MoST'15)*. IEEE Computer Society, 2015.

[115] Naiakshina, A., Danilova, A., Tiefenau, C., Herzog, M., Dechand, S., and Smith, M. Why Do Developers Get Password Storage Wrong?: A Qualitative Usability Study. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. ACM, 2017.

[116] *Number of available applications in the Google Play Store from December 2009 to June 2020*.

[117] Ogawa, H., Takimoto, E., Mouri, K., and Saito, S. User-Side Updating of Third-Party Libraries for Android Applications. In: *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. 2018.

[118] Olejnik, K., Dacosta, I., Machado, J. S., Huguenin, K., Khan, M. E., and Hubaux, J. P. SmarPer: Context-Aware and Automatic Runtime-Permissions for Mobile Devices. In: *Proc. 38th IEEE Symposium on Security and Privacy (SP'17)*. IEEE Computer Society, 2017.

[119] Pagano, D. and Maalej, W. User feedback in the appstore: An empirical study. In: *Requirements Engineering Conference (RE), 2013 21st IEEE International*. IEEE. 2013.

[120] Palomba, F., Salza, P., Ciurumelea, A., Panichella, S., Gall, H., Ferrucci, F., and De Lucia, A. Recommending and Localizing Change Requests for Mobile Apps Based on User Reviews. In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE '17. IEEE Press, 2017.

[121] Pandita, R., Xiao, X., Yang, W., Enck, W., and Xie, T. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In: *USENIX Security Symposium*. 2013.

[122] Pistoia, M., Tripp, O., and Lubensky, D. Combining Static Code Analysis and Machine Learning for Automatic Detection of Security Vulnerabilities in Mobile Apps. *Mobile Application Development, Usability, and Security* (2016), 68.

[123] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In: *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society, 2014.

[124] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In: *NDSS 2014*. Vol. 14. 2014.

[125] Porter Felt, A., Chin, E., Hanna, S., Song, D., and Wagner, D. Android Permissions Demystified. In: *Proc. 18th ACM Conference on Computer and Communication Security (CCS '11)*. ACM, 2011.

[126] Porter Felt, A., Egelman, S., Finifter, M., Akhawe, D., and Wagner, D. How to Ask for Permission. In: *Proceedings of the 7th USENIX Conference on Hot Topics in Security (HotSec'12)*. USENIX Association, 2012.

[127] Porter Felt, A., Egelman, S., and Wagner, D. I'Ve Got 99 Problems, but Vibration Ain'T One: A Survey of Smartphone Users' Concerns. In: *Proc. 2nd ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM'12)*. ACM, 2012.

[128] Porter Felt, A., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. Permission Re-Delegation: Attacks and Defenses. In: *Proc. 20th USENIX Security Symposium (SEC'11)*. USENIX Association, 2011.

[129] Prati, R. C., Batista, G. E., and Monard, M. C. Class imbalances versus class overlapping: an analysis of a learning system behavior. In: *Mexican international conference on artificial intelligence*. Springer. 2004.

[130] *Privacy, Security, Deception*. `https://play.google.com/about/privacy-security-deception/`. 2019.

[131] Qu, Z., Rastogi, V., Zhang, X., Chen, Y., Zhu, T., and Chen, Z. AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In: *Proc. 21st ACM Conference on Computer and Communication Security (CCS'14)*. ACM, 2014.

[132] Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., and Yao, D. (CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. ACM, 2019.

[133] Ren, J., Lindorfer, M., Dubois, D. J., Rao, A., Choffnes, D., and Vallina-Rodriguez, N. Bug Fixes, Improvements, ... and Privacy Leaks – A Longitudinal Study of PII Leaks Across Android App Versions. In: *Proceedings of the 25th Annual Network and Distributed System Security Symposium*. NDSS '18. The Internet Society, 2018.

[134] *Reply to Reviews API*. `https://developers.google.com/android-publisher/reply-to-reviews`. 2018.

[135] Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H. J., and Cowan, C. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In: *Proc. 33rd IEEE Symposium on Security and Privacy (SP'12)*. IEEE Computer Society, 2012.

[136] Schechter, S. E., Dhamija, R., Ozment, A., and Fischer, I. The Emperor's New Security Indicators. In: *2007 IEEE Symposium on Security and Privacy (SP '07)*. 2007.

[137] Sheffer, Y., Saint-Andre, P., and Holz, R. *Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*. RFC 7525. 2015.

[138] Shi, P., Xu, H., and Zhang, X. (Informing Security Indicator Design in Web Browsers. In: *Proceedings of the 2011 iConference*. iConference '11. ACM, 2011.

[139] Shuai, S., Guowei, D., Tao, G., Tianchang, Y., and Chenjie, S. Modelling Analysis and Auto-detection of Cryptographic Misuse in Android Applications. In: *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*. 2014.

[140] *Snyk: A developer-first solution that automates finding & fixing vulnerabilities in your dependencies*. https://snyk.io. 2019.

[141] Son, S., Daehyeok, G., Kaist, K., and Shmatikov, V. What Mobile Ads Know about Mobile Users. In: *Proc. 23rd Annual Network & Distributed System Security Symposium (NDSS '16)*. The Internet Society, 2016.

[142] Son, S., McKinley, K. S., and Shmatikov, V. Rolecast: finding missing security checks when you do not know what checks are. *ACM SIGPLAN Notices*, 46.10 (2011), 1069–1084.

[143] Sriram, B., Fuhry, D., Demir, E., Ferhatosmanoglu, H., and Demirbas, M. Short text classification in twitter to improve information filtering. In: *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*. ACM. 2010.

[144] Stevens, R., Gibler, C., Crussell, J., Erickson, J., and Chen, H. Investigating User Privacy in Android Ad Libraries. In: *MoST'12*. IEEE, 2012.

[145] Stevens, R., Gibler, C., Crussell, J., Erickson, J., and Chen, H. Investigating user privacy in android ad libraries. In: *Workshop on Mobile Security Technologies (MoST)*. Vol. 10. 2012.

[146] Team, A. *Android Lint tool*. https://developer.android.com/studio/write/lint.html. Last visited: 17/05/2017. 2017.

[147] Thomas, T. W., Lipford, H., Chu, B., Smith, J., and Murphy-Hill, E. What Questions Remain? An Examination of How Developers Understand an Interactive Static Analysis Tool. In: *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. USENIX Association, 2016.

[148] Tong, S. and Koller, D. Support vector machine active learning with applications to text classification. *Journal of machine learning research*, 2.Nov (2001), 45–66.

[149] *Top most popular libraries on Maven*. https://mvnrepository.com/popular. 2018.

[150] Vallée-Rai, R., Gagnon, E., Hendren, L., Lam, P., Pominville, P., and Sundaresan, V. Optimizing Java bytecode using the Soot framework: Is it feasible? In: *International conference on compiler construction*. Springer. 2000.

[151] Vaudenay, S. On the weak keys of blowfish. In: *Fast Software Encryption: Third International Workshop Cambridge, UK, February 21–23 1996 Proceedings*. Ed. by Gollmann, D. Springer Berlin Heidelberg, 1996.

[152] Vaudenay, S. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ... In: *Proceedings of the International Conference on the Theory*

and Applications of Cryptographic Techniques: Advances in Cryptology. EUROCRYPT '02. Springer-Verlag, 2002.

[153] Viennot, N., Garcia, E., and Nieh, J. A Measurement Study of Google Play. *SIGMETRICS Perform. Eval. Rev.* 42.1 (2014), 221–233.

[154] Wang, R., Xing, L., Wang, X., and Chen, S. Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation. In: *Proc. 20th ACM Conference on Computer and Communication Security (CCS '13)*. ACM, 2013.

[155] Watanabe, T., Akiyama, M., Kanei, F., Shioji, E., Takata, Y., Sun, B., Ishi, Y., Shibahara, T., Yagi, T., and Mori, T. Understanding the Origins of Mobile App Vulnerabilities: A Large-scale Measurement Study of Free and Paid Apps. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. MSR '17. IEEE Press, 2017.

[156] *Why Does My Flashlight App Need Access to All These Things?!* `http://www.sudosecure.com/why-does-my-flashlight-app-need-access-to-all-these-things/`. 2013.

[157] Wijesekera, P., Baokar, A., Tsai, L., Reardon, J., Egelman, S., Wagner, D., and Beznosov, K. The Feasibility of Dynamically Granted Permissions: Aligning Mobile Privacy with User Preferences. In: *Proc. 38th IEEE Symposium on Security and Privacy (SP'17)*. IEEE Computer Society, 2017.

[158] *Xposed Module Repository*. `http://repo.xposed.info/`. 2019.

[159] Yang, C., Xu, Z., Gu, G., Yegneswaran, V., and Porras, P. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In: *European symposium on research in computer security*. Springer. 2014.

[160] Zhang, J., Beresford, A. R., and Kollmann, S. A. LibID: Reliable Identification of Obfuscated Third-Party Android Libraries. In: ISSTA 2019. Association for Computing Machinery, 2019.

[161] Zhong, N. and Michahelles, F. Where Should You Focus: Long Tail or Superstar?: An Analysis of App Adoption on the Android Market. In: *SIGGRAPH Asia 2012 Symposium on Apps*. SA '12. ACM, 2012.

[162] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: *NDSS*. Vol. 25. 4. 2012.

A Appendix

A.1 FixDroid's Survey

A.1.1 FixDroid specific questions

- During the study, did you notice any interaction from FixDroid while performing the tasks? If yes:
  - What interaction did you see from FixDroid? Warning messages, Highlighted code, warning icon, quick-fixes, other
- Did the plugin provide you any additional information?
- Did you use any provided quick-fix in our lab study? If yes:
  - Did the inserted code FixDroid work?
  - Do you feel it was helpful?

A.1.2 General questions

- What is your age?
- What is your gender?
- Where are you from?
- For how many years have been programming in Android?
- What is your highest degree of education?
- Is programming your primary job? If yes: Is writing Android code part of your primary job?. If no: Was programming part of your job in the last 5 years?
- Do you have information security background?
- How many Android applications you have developed?
- Where do you usually look for security related coding questions? (website)
- Are you familiar with Android Studio (or IntelliJ IDE in general)?

A.1.3 Experience Sampling Survey

- How do you think this quick-fix is useful? (Strongly agree; agree; neutral; disagree; strongly disagree)
- It seems like you are copying code from somewhere, could you please tell us the website (the link) you have copied this code from?

A.2 Up2Dep Survey Questions

A.2.1 App Development

Q1: How do you prefer getting update notifications? [multiple choice]

- Yellow highlighting on the dependency version
- Pop up when new versions are available, with "Ignore" option
- When I build/compile my project?
- Other [free text]

Q2: Based on which criteria do you usually pick a library for your projects? [multiple choice]

- Popularity
- Easy to use
- Functionality
- Security
- Other

Q3: Have you developed any third-party libraries?[Yes/No]

- Yes: Which library is that? [freetext]
- No

Q4: How would you rate the security (whether a given version has security vulnerability) of libraries you decide to include it into your projects [single choice]
1-5

Q5: Did you notice any highlights regarding outdated library versions in your app's Gradle files? [single choice]

- Yes
- No
- I don't know

Q6: Where do you reach out for help while solving programming tasks that relate to third-party libraries? [multiple choice]

- StackOverflow
- Search engines
- Third party library's website
- Other [free text]

A.2.2 Up2Dep Usage

Q7: How did you get to know Up2Dep? [multiple choice]

- Friends, colleagues
- IntelliJ IDEA/Android Studio repository
- Twiter
- Android Developer Conference
- Other

Q8: Which features of Up2Dep do you find useful? (screenshots are included for each feature)

- Compatibility check (compatible version vs. latest version)
- Insecure version check
- Crypto API misuse check
- Show dependencies and alternative API suggestions
- Other [free text]

Q9: Since you started using Up2Dep, how many outdated libraries have you updated?

- 0
- 1
- More than 2
- Other [free text]

A.2.3 Up2Dep Usability - SUS Questions

Q10: For each of the following statements, how strongly do you agree or disagree (Strongly disagree, disagree, neutral, agree, strongly agree)

- I think that I would like to use Up2Dep frequently.
- I found Up2Dep unnecessarily complex.
- I thought Up2Dep was easy to use.
- I think that I would need the support of a technical person to be able to use Up2Dep.
- I found the various functions of Up2Dep were well integrated.
- I thought there was too much inconsistency in Up2Dep.
- I would imagine that most people would learn to use Up2Dep very quickly.
- I found Up2Dep very cumbersome to use.
- I felt very confident using Up2Dep.
- I needed to learn a lot of things before I could get going with Up2Dep.

A.2.4 Demographic

Q11: How many years have you been programming in Android?

- less than 1 year
- around 2 years
- around 3 years

- more than 3 years

Q12: How old are you?

- 18–30
- 31-40
- 41-50
- >50
- No answer

Q13: What is your gender?

- Male
- Female
- No answer

Q14: How many apps have you developed so far?

- 1
- 2
- more than 2
- 0

Q15: Do you have IT-Security background?

- Yes
- No

Q16: Where are you from? [free text]

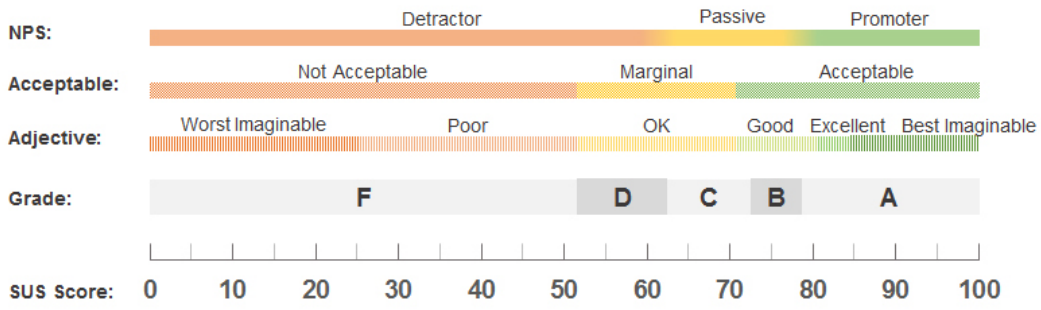

Figure A.1: SUS score and its meaning (110). NPS stands for Net Promoter Score - measuring how likely users recommend a system/product to a friend