Droid M+: Developer Support for Imbibing Android's New Permission Model

Ioannis Gasparis
University of California, Riverside
igasp001@cs.ucr.edu

Azeem Aqil
University of California, Riverside
aaqil001@cs.ucr.edu

Zhiyun Qian
University of California, Riverside
zhiyunq@cs.ucr.edu

Chengyu Song
University of California, Riverside
csong@cs.ucr.edu

Srikanth V. Krishnamurthy
University of California, Riverside
krish@cs.ucr.edu

Rajiv Gupta
University of California, Riverside
gupta@cs.ucr.edu

Edward Colbert
U.S. Army Research Lab
edward.j.colbert2.civ@mail.mil

ABSTRACT

In Android 6.0, Google revamped its long criticized permission model to prompt the user during runtime, and allow her to dynamically revoke granted permissions. Towards steering developers to this new model and improve user experience, Google also provides guidelines on (a) how permission requests should be formulated (b) how to educate users on why a permission is needed and (c) how to provide feedback when a permission is denied. In this paper we perform, to the best of our knowledge, the first measurement study on the adoption of Android's new model on recently updated apps from the official Google Play Store. We find that, unfortunately, (1) most apps have not been migrated to this new model and (2) for those that do support the model, many do not adhere to Google's guidelines. We attribute this unsatisfying status quo to the lack of automated transformation tools that can help developers refactor their code; via an IRB approved study we find that developers felt that there was a non-trivial effort involved in migrating their apps to the new model. Towards solving this problem, we develop Droid M+, a system that helps developers to easily retrofit their legacy code to support the new permission model and adhere to Google's guidelines. We believe that Droid M+ offers a significant step in preserving user privacy and improving user experience.

KEYWORDS

android permissions; mobile privacy; mobile security

ACM Reference Format:
Ioannis Gasparis, Azeem Aqil, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, Rajiv Gupta, and Edward Colbert. 2018. Droid M+: Developer Support for Imbibing Android's New Permission Model. In *ASIA CCS '18: 2018 ACM Asia Conference on Computer and Communications Security, June 4–8, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196494.3196533

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.
ASIA CCS '18, June 4–8, 2018, Incheon, Republic of Korea
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5576-6/18/06…$15.00
https://doi.org/10.1145/3196494.3196533

1 INTRODUCTION

Application sandboxing and the permission system are key components of modern mobile operating systems for protecting the users' personal data and privacy. Prior to Android 6.0, Android used the *ask-on-install* permission model: (1) developers declare a set of required permissions in the app's manifest file, (2) at installation time, Android asks users to review the requested permissions and, then (3) users either grant all the requested permissions or refuse the installation. Many prior studies have shown the problems with this design (e.g., [15, 22, 47]).

The first frequently criticized aspect of the old Android permission model is that it is complex and unintuitive. Over the years, the number of system permissions have increased from 75 (API level 1) to 138 (API level 25) [45]. Without a good understanding of the permission model, developers tended to ask for more permissions than needed [14, 22]. For example, one-third of the 940 apps analyzed in [14] were over privileged. Requesting unnecessary permissions is a big security problem since attackers can leverage a multitude of combinations of these permissions to compromise user privacy (*e.g.*, leaking personal photos over the Internet). On the user side, due to the same problem, few people are likely to carefully review the requested permissions and even fewer will correctly understand how permissions are mapped to sensitive resources [15, 26].

The second problem with the old permission model is the lack of flexibility; users can neither grant a subset of all requested permissions, nor revoke granted permissions. A recent user study [47] showed that 80% of the participants would have preferred to *decline* at least one requested permission and one-third of the requested accesses to sensitive resources because of the belief that (1) the requested permission did not pertain to the apps' functions; or (2) it involved information that they were uncomfortable sharing. The lack of such flexibility has also led Android users to either ignore the permission warnings or to not use the app [15]. For instance, a recent survey [31] of over 400 adults showed that over 60% of the participants decided not to install an app because it required many permissions. Irrevocable permissions also pose privacy concerns to users as apps can retain their access to sensitive sensors (*e.g.*, microphone) while running in the background [25].

Wijesekara *et al.* [47] proposed using Nissenbaum's theory of context integrity [30] as a guideline to determine whether accesses

to protected resources would violate users' privacy. Under this guideline, the ask-on-install model clearly lacks enough *contextual information* which makes it very difficult for normal users to determine why a permission is needed and if the app would violate their privacy [15]. Moreover, they also found that even if permissions are requested during runtime, the lack of proper mechanisms to explain why a particular resource was necessary could also lead to incorrect perceptions and less willingness to grant the permission.

In Android 6.0 or Android M(arshmallow), Google revamped the Android permission model to solve the aforementioned problems. Specifically, Android no longer promotes users to grant permissions during install-time; instead, *normal permissions* (*i.e.*, no great risk to users' privacy and security) are automatically granted and *dangerous permissions* are requested during runtime. To further streamline the number of requests, dangerous permissions are put into *permission groups* and granting one dangerous permission would automatically grant the others in the same group. To help developers convey to users why a permission is needed, Google also added an API to check whether further explanation may be needed. Finally, users can revoke a granted permission at anytime using the system settings. We discuss the new permission model in detail in § 2; note that the permission model carries over to later versions of Android.

While the new permission model significantly improves (compared to the old model) user control over privacy and in making apps more appealing for adoption (recall that asking permissions at install-time may affect users' decisions on installing an app), we find that only a few apps have effectively migrated to the new permission model. To verify whether this is a general issue for the entire Google Play Store, we conduct, to the best of our knowledge, the first systematic measurement study and an IRB approved developer survey towards answering the following questions: **(a)** How many newly released apps have adopted the new permission model? **(b)** For those that have not, what are the likely reasons? and, **(c)** For those that have adopted, how well do they adhere to Google's guidelines [17]?

Our analysis results show that, despite a 26.7% market share of Android M, (i) very few of the apps have adopted the new model and, (ii) even those apps that have done so, have significant shortcomings. We attribute one cause for this unsatisfying status quo to be the lack of a good development tool. In particular, to migrate to the new model, developers have to make non-trivial changes to their existing code. This is especially true if they intend to follow Google's guidelines, *i.e.*, properly checking if a permission was revoked, educating a user in-context on why a permission is needed, and properly handling instances where a permission request is denied. We conduct a developer survey wherein a majority of the respondents say that they have not migrated their apps to Android M because of the lack of an easy-to-use tool to help to migrate to the new model.

As a key contribution, we develop such a tool set, Droid M+, to help developers to retrofit their legacy code to the new permission model. Given the source code of an Android app, our tool will (1) identify different *functionalities* (*i.e.*, context) of the app; (2) identify permissions that are required for each functionality; (3) automatically populate the entry of each functionality with an annotation that allows developers to review the requested permissions and provide corresponding justifications; (4) automatically translate the annotation into real Java code; and (5) provide a default callback function to handle denied requests. In summary, Droid M+ allows developers to easily morph their app(s) to support revocable permissions and adhere to Google's guidelines, with minimal changes to their existing code. Without Droid M+, it is currently a challenge to handle the asynchronous `requestPermissions()` calls, as the code after the check still executes. To place permission checks properly (ask only when necessary), it requires significant refactoring of the code. Our evaluations show that Droid M+ can facilitate easy permission revocations as intended by Android M, hiding tedious details from the developers.

In summary, this paper makes these key contributions:

- We perform an in depth measurement study of 4743 top free apps from the Google Play Store and examine the adoption of the new Android permission model. Our study shows that only 62.6% of the apps have migrated to the new model. Of these, nearly 45% of the apps do not follow the Google guidelines. Finally, about 2.9% of them refuse to run if a permission request is denied.
- We conduct a developer survey which indicates that about 54 % of the responding developers have not migrated their code to the new model because they feel it is hard, and there is no helper tool available to facilitate such a transition.
- We design, implement and evaluate Droid M+, a tool set to help developers migrate to Android's new permission model and adhere to Google's guidelines.

2 BACKGROUND AND MOTIVATION

The Android M Permission Model: Android uses application sandboxing to isolate apps and protect users' privacy. Accesses to sensitive sensors, users' personal data, and other services/apps are mediated by the Android permission system. To request authorizations, an app must declare the required permissions in its "manifest file". Normal permissions (*i.e.*, permissions with no great risk to the user's privacy or security) are automatically granted at install. Android will prompt the user when an app seeks to access a resource guarded by a *dangerous permission*. Users can then choose from three options: (1) grant the permission and the app will retain access to the resource; (2) deny this particular request; or, (3) for a permission that has been previously denied, a chat box is provided using which, automatically deny all future requests. If the user denies a permission request, Android M allows the app to either continue running with limited functionality or completely disables its functionalities. Dangerous permissions are put into permission groups viz., calender, camera, contacts, location, microphone, phone, sensors, sms, and storage. If one permission in a permission group is granted, the remaining permissions in the same group are taken to be automatically granted. Android M also allows the user to modify the permissions granted to apps using system settings. Note that a user can also revoke permissions for legacy apps (API level < 23); in such cases, the platform will disable the APIs guarded by the permission by making them no-ops, which may return an empty result or a default error.

Steps for Migration: To provide revocable permissions, developers should update their apps via the following steps (Fig. 1):

- At each instance when an app needs to invoke API(s) that require a permission, the developer should insert a call to explicitly check if the permission is granted. This is critical because users may revoke granted permissions at anytime, even for legacy apps. Developers

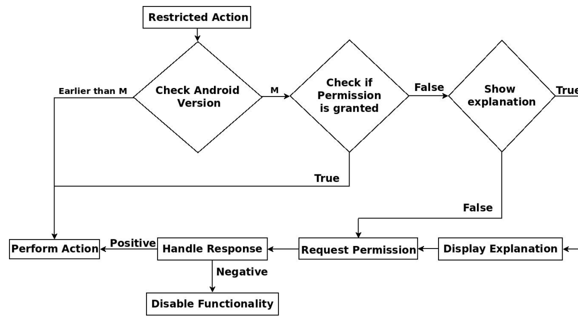

Figure 1: The permission workflow of Android M.

can use the `ContextCompat.checkSelfPermission` method from the support library or directly invoke platform APIs to do so.

- Optional but as recommended by Google, to help the user understand why the app needs a permission, the developer should write code to display a justification before requesting a permission.
- If the app does not have the permission(s) required to complete a restricted action, the developer should insert a call to request the permission(s). Developers can do so by calling the `ActivityCompat.requestPermissions` method from the support library or directly invoke platform APIs. If a requested permission has not been permanently denied, the platform will display a dialog box to the user showing which permission group is requested.
- Since permission requests are asynchronous, a callback method (by overriding the `onRequestPermissionsResult` method) must be provided to handle the results of such requests. Upon invocation, the system will provide a list of granted and denied permissions.
- In the case of denied permissions, the developer must ensure that the app either continues execution with limited functionality, or disables the corresponding functionality and explain to the user why the permission was critical.

Google's Guidelines: Google's guidelines for permission management [17] suggest that permission requests be simple, transparent and understandable. These attributes help the adoption of an app [41, 47], *i.e.*, users were more willing to grant permission(s) when requested in-context and with proper justifications. Google recommends that permissions critical to the core functionalities of an app (*e.g.*, location to a map app) be requested up-front, while secondary permissions be requested in-context. If the context itself is not self-evident (*e.g.*, requesting the camera permission when taking a photo makes sense but the reason for requesting location at the same time lacks clarity), the app should educate the user about why the permission is requested. The education recommendation also applies to critical permission(s) asked up-front. When a permission is denied, the app should provide feedback to the user and if possible provide other available options. If critical permissions are denied, the app should educate the user as to why the permission is critical for it to function and offer a button so that the user can change the settings. For secondary permissions, the app should disable the corresponding features and continue providing the basic functionalities.

3 ANALYZING ANDROID M ADOPTION

Next, we present (a) our in-depth measurement study on the adoption of Android M's new permission model and (b) our developer survey to offer possible explanations to the findings from the study.

3.1 A Motivating Example

As a motivating example, we consider an app `Any.do` [5], one of the most popular to-do apps on Google Play (checked in November 2016). This app requires access to the microphone, location, contacts, calendar, the device identifier, and local storage. It is unclear why a to-do app needs all these permissions. The app description page on Google Play offers no proper information either. Further, although this app does target the new permission model, the way it requests permissions does not adhere to Google's guidelines. Specifically, when the app is first launched, all the permissions are requested up-front (Fig. 2). At this time, it is unclear why these permissions are required, and no justifications are offered (even though the permissions are legitimately used). This motivates us to perform an in depth measurement study on how top apps on the Google Play Store adopt Android M's permission model.

3.2 Measurement Tool

We design and implement a novel tool, the *Revocable Permission Analyzer*, to experimentally quantify via different metrics, the way existing apps are developed using the new permission model of Android M. The Analyzer first uses apktool [7] to decompile and decode the APK's resources including the manifest file and the UI XML file. It then uses androguard [3] to generate the call graph of the APK. Using the call graph, the Analyzer looks for invocations of *checkSelfPermission*, *requestPermissions*, *shouldShowRequestPermissionRationale*, and *onRequestPermissionsResult*. By focusing on these API calls, it examines (1) whether the app is requesting and checking for dangerous permissions, (2) whether the app shows a rationale for requesting the permission, and (3) what it does after the user responds to the permission request.

Specifically, to collect permissions requested up-front, Revocable Permission Analyzer checks the main "Activity" of a given app (*i.e.*, *onCreate*, *onStart*, and *onResume*) and looks for invocation of *requestPermissions()* in the call graph rooted by the main activities. Any permission not asked upfront, is considered as being asked in-context. To check if customized messages are included to justify requested permissions, Revocable Permission Analyzer looks for the invocation of *shouldShowRequestPermissionRationale()*. This API returns whether a customized message needs to be shown; if so, the app can display a message to the user and then call the *requestPermissions()*. If *shouldShowRequestPermissionRationale()* is not invoked, then it is a strong indication that *no* customized message (education) is included. When customized messages are shown, we look up the message from the strings.xml resource file.

Recall that Google's guidelines (§ 2) suggest that only critical and obvious permissions should be asked up-front and developers should

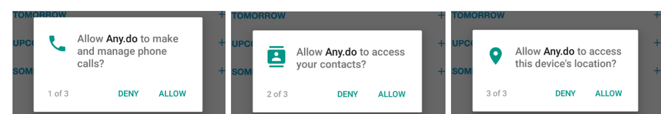

Figure 2: Any.do permissions during startup.

educate the users when they request non-obvious permissions. To check the compliance of an app, we manually process the analysis results of Revocable Permission Analyzer. In particular, we consider a permission to be critical if it is needed by functionalities indicated in the app's description, by its title, or by its category. For example, a camera app is expected to request the Camera permission; similarly a travel/navigation app can be expected to ask for the location permission. For educational messages, we use natural language processing (i.e., significant manual work is not needed). Specifically, (a) we extract all the strings from the app's resources, (b) we remove any occurence of the default educational messages that Google provides, and (c) we extract semantics from the educational messages by leveraging [34] and compute their similarities with the default educational messages.

3.3 Results and Inferences

Android Applications Dataset: Our measurement study is based on 4743 applications that are obtained by downloading the top free apps from each available category (e.g. Social, Games, etc.) as per Google's Play Store [20] charts from June 2017.

Adoption of Android M Permission Model: *A large number of Android M apps do not support revocable permissions.* From the 4743 apps, only 2973 are developed for Android M or above (with the `targetSdkVersion` of 23 or higher). Note, that some of these apps can be extremely popular with over 100 milions downloads (e.g., ES File Explorer File Manager) [21]. Further, there are 302 apps out of the 2973 that do not require *any* so called dangerous permissions and thus, in a normal way do not invoke any Android M APIs. We want to point out that, surprisingly, there are apps like Ringdroid, which is developed by Google itself and still does not support revocable permissions even though they are developed with the latest Android SDK (see § 5 for more details).

As reported in [4], in December 2017, the share of Android users that use Android M and N, is 48%; one can expect this percentage to keep growing. Unfortunately, the above result shows that many of the apps still do not support revocable permissions. This implies that a user who has a phone with a version of Android that supports the latest fine-grained permission mechanism, will be forced to grant all the permissions to most of the applications; otherwise these applications will likely not function correctly [13].

Permissions Asked Upfront vs. In-Context: *A significant fraction of apps ask permissions upfront instead of in-context.* We find that 14.07% of the apps (376 out of 2671) request permissions during startup while most apps (2295 out of 2671) attempt to request permissions in-context. Some apps will request the same permissions both upfront and in-context. Often, permissions requested upfront are not really critical and the app can still function without them. Fig. 3 shows the distribution of the number of the critical permissions asked upfront by the 376 apps; 68.5% have only one critical permission, 22.6% require 2 permissions, and 8.9% require 3. This shows that in general very few permissions are considered critical and should be asked upfront. Unfortunately, in most cases, apps often ask more permissions. Fig. 4 shows the total number of over-asked permissions that those apps are requesting upfront. Clearly, with respect to more than 59% of the apps, one or more permissions requested upfront are in fact not critical. Note, that permissions that are being asked upfront by the apps in our dataset, are not usually sought by third party libraries. The are two reasons for this: (a) only an Activity or a Fragment can request dangerous permissions during runtime (third party libraries do not contain those) and (b) no permissions are sought during the initialization of the third party libraries.

Some of the apps expect all permissions to be granted upfront or will simply refuse to run. Interestingly, some apps that support the Android M permission model and use the corresponding APIs expect all the permissions asked upfront to be granted; otherwise they simply refuse to run. We leverage the Revocable Permission Analyzer to check the statements invoked when a requested permission is refused; if statements like "`System.exit(0); or finish();`" are encountered, it is evident that the app is simply voluntarily ending its run due to permission revocation.

This style of such an app defeats the purpose since it does not really intend to support revocation of permissions (even when some of them are not critical). Overall, using this approach, we identify 2.9% or 80 apps out of 2671 apps that ask at least one non-critical permission, and yet refuse to run if such a permission is not granted. The remaining 296 apps still ask for these permissions again in-context, if they were denied when requested up front.

User Education: *A significant fraction of apps does not provide meaningful explanations for non-obvious permissions.* We find that from the apps that request permissions in-context, only 54.17% (1447 in total out of 2671 apps) educate the users (i.e., a meaningful message that tells the user why the requested permission is needed for a given functionality, is provided). For example, when the app SONGily [39] requests the storage permission, the following message is provided: "Permission to write files is required". We deem this message does not educate the user. Similarly, AskMD [8] provides the following message when requesting for accessing the microphone: "AskMD would like to access your microphone. Please grant the permission in Settings." Contrary to those apps, theScore [43] provides the following message when it requests access to the user's calendar: "In order to add events to your calendar, we require the Android Contacts permission. We will not be reading or accessing your information in any way, other than to add the events.". We deem this message as following Google's guidelines by properly educating the user why it needs that permission.

Permissions asked upfront are less likely to have meaningful explanations. From the apps that ask permissions upfront, only 177 (47.07%) educate the user properly. This is a much lower rate compared to permissions that are asked in-context. Permissions that are asked upfront lack the context and it is generally even more important to educate the users about what the permissions are used for. Unfortunately, the results indicate that a majority of the considered apps fail to adhere to the Google's guidelines.

3.4 Developer Survey

To ascertain why developers may not have migrated their apps to the new permission model, we conduct a survey. We recruited developers with apps on GitHub and the Google Play store. The survey was approved by our institution's IRB and conducted in August 2017.

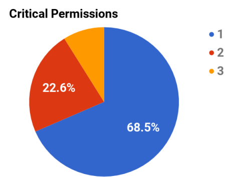

Figure 3: Critical permissions that can/should be asked upfront.

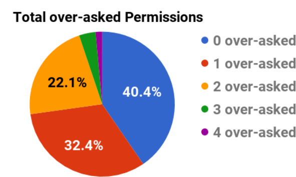

Figure 4: Over-asked permissions during launch.

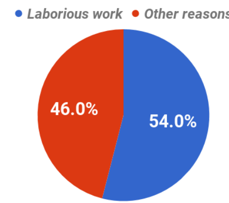

Figure 5: Fraction that gave technical laboriousness as primary reason.

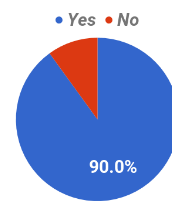

Figure 6: Fraction that would update if tool available.

Recruitment: For recruitment via GitHub, we used the new GitHub `graphql` API to get a list of repositories that contained the AndroidManifest.xml file. Recruiting developers via GitHub has previously been employed in [1]. Since there is no way to search for Android repositories using the GitHub API, we reasoned that any repository that contained the `AndroidManifest.xml file` was likely an Android app. Next, we cloned all the repositories returned via the API call and checked the manifest file for the target sdk to find apps that had not updated to Android M yet. Lastly, we used the *git log* command to extract developer email addresses. Github currently only returns a thousand responses for any API query. This limits our survey pool greatly. Therefore, we used a second recruitment resource, viz., the Google Play Store.

For recruitment via the play store, we followed the same strategy of crawling the top android apps as in our measurement study. We extracted email addresses from the app's respective play store pages. Most apps on the play store list email addresses for technical support to which we sent the survey.

We sent emails with links to the anonymous survey to 2500 email addresses. We did not offer recruitees any incentive for filling out the survey since we did not want our results to be biased. Specifically, while we can extract developer emails via GitHub, the play store mostly lists support email addresses which may or may not be answered by actual developers (could be support staff). We requested potential developers to only fill out the survey if they were developers who had not transitioned their apps to Android M.

Survey: We hosted the survey on SurveyMonkey and asked two key questions: *Is the reason why you have not migrated because it is hard to do so (technically laborious)?* and *if so, would you consider migrating if a tool was available to help you migrate your app to the new model?* We also included a question that invited survey takers to explain in a few words: *What, if anything, do you think can be done to make the permission model easier to use for developers?*

Results: Of the 2500 emails we sent, we have so far received a total of 187 replies. The results of the survey are summarized in Fig. 5 and Fig. 6. 99 developers($\approx$ 54% of those who responded) indicated that the reason for not migrating their app was because it was laborious to do so. Furthermore, 90 people out of those 99 developers ($\approx$ 90%) indicated that they would migrate if an automated helper tool was available. The ones that did not say it was laborious to migrate, gave no indication with regards to why they did not migrate their app.

Finally, the majority of survey takers did not respond to the descriptive question. We suspect that this was because the descriptive question required more effort on their part. About 10% of the survey takers responded and indicated that the permission model is a substantial change and as such, places an undue burden in making their apps forward compatible. They would have liked Google to offer some easier default knobs for transition.

In short, our survey results suggest that a large fraction of developers believe that the work associated with transitioning their apps from prior Android versions, is involved. The same developers also indicated their inclination to update if a tool was available.

3.5 Summary

Our measurement study demonstrates that only a significant percentage of applications that were built on the Android M platform, do not properly adopt the new permission model. An even smaller fraction of these applications, unfortunately, adhere to Google's guidelines. Our survey suggests that one the of main reasons for developers not fully and properly adopting the new permission model, is the complexity and the work associated with transitioning their apps from the previous Android version for which the app was developed, to the newer version. As users become more privacy conscious [31, 47], following Google's guidelines can be a key factor influencing their choice of apps. Below is a list of our key observations:

- Approximately 60% of the top apps supporting Android M API level 23 or above are not using its permission revocation APIs properly (59.6 % ask for at least one non-critical permissions up front).
- 45.83% and 52.93% of the permissions asked in-context versus upfront do not have informative explanations for why the permissions are sought as per the Google guidelines.
- Some of these apps ($\approx$ 3 %) simply refuse to run if any of the permissions asked upfront are not granted.
- 54% of the surveyed Android developers did not transition their apps due to the perceived difficulty. They also indicated a willingness to transition if an automated helper tool was made available.

4 DROID M+ TOOL SET

In this section, we describe the design of Droid M+ and its component tools. Droid M+ consists of three major components(see Fig. 7). The first component is a static analysis tool that helps developers identify different functionalities (*i.e.*, context) of their apps, what permission(s) each functionality requires, and the right place to request the permission(s). The second part is an annotation system that facilitates the easy integration of revocable permissions and conformance to Google's guidelines within existing Android app

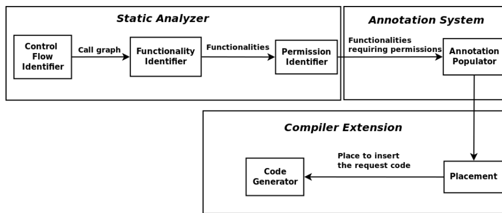

Figure 7: Droid M+ architecture.

code. Finally, Droid M+ contains a compiler extension that interprets the annotations and inserts the corresponding code.

4.1 Static Analyzer

The static analyzer has three tasks: (1) identify functionalities within an app, (2) identify permission(s) required by each functionality, and (3) identify the right place to annotate. Its main function is to help developers migrate apps developed against an old API level, to the new API level (≥ 23). However, apps that are already developed for the new API level can also utilize this tool to help refactor the code, *i.e.*, determine what permissions to request, where to place the requests, and what education message to display with each request.

Identify Control Flow: Before we can do any useful analysis on an app, the tool needs to first parse the source code and generate the corresponding call graph and control flow graph. These are standard techniques which we will not describe in detail. As discussed in the literature (*e.g.*, [37]), there are two challenges worth mentioning. First, point-to analysis [37] needs to be employed in order to generate an accurate call graph. Second, Java reflection needs to be handled to generate the complete call graph. Currently, we do not support the latter but there are ways to statically resolve the Java reflection calls [38], and we plan to incorporate these in the future.

Identify Functionalities: Given a call graph, we define a *functionality* as a collection of nodes in the control flow graph that are reachable from a single *entry point*. In Android, entry points include activities (*i.e.*, the `onCreate()` method of the `android.app.Activity` class), callback methods for UI interactions (*e.g.*, click of a button), content providers (app data managers), services (background threads), and broadcast receivers (system-wide broadcast events). These entry points can be identified by parsing the manifest files and analyzing the code. The reasoning is that these entry points represent user-triggered events, or significant activities that should be made aware to users (*e.g.*, background services). Each functionality should contain a sequence of instructions involving some usage of permissions. We believe it is a natural and reasonable place to request permissions; this practice is aligned with Google's guidelines and is actually used in many real-world apps that perform in-context permission requests (*e.g.*, WhatsApp Messenger [46]). As most existing static analysis tools for Android already support the identification of all the entry points of an app and building a complete call graph ([3],[40]), we omit the details here.

Identify Permissions: The first step in identifying the required permissions, is to parse the manifest file and find out the target API level of the app. Note that although our tool helps migrate the app to the new permission model of Android M (API level 23), this step is still needed to support the newer version of the SDK (*i.e.*, if the app's current `targetSdkVersion` is lower than 23, we assume 23; else, we use the app's `targetSdkVersion`). The API level is used to map SDK APIs to their required permissions. Specifically, we use the latest version of PScout [9] to generate the database that maps a permission to a set of SDK APIs that require this permission. There is no version of PScout for Android M yet. However most apps we test were developed for previous versions of Android. For cases where apps were developed for M, we check if they do an Android version check for API calls that were made available beginning Android M, and then explore the Android code manually to see if they need dangerous permissions. Note, that our dataset did not contain any app developed with the `minimumSdkVersion` equal to 23.

With this mapping information, identifying permissions required by a functionality is straightforward. In particular, given the complete static call graph, we use a standard reachability analysis for Android apps to identify all the potential invocable SDK APIs from the entry point of a functionality. Then we use the permission mapping to generate all the required permission(s) for this functionality.

Third-Party Libraries: Some third-party libraries such as advertisement libraries (*e.g.*, AdMob) and analytic services could also access protected resources (*e.g.*, location). Because these libraries are usually delivered in binary (bytecode) format, we need additional steps and different analysis tools to identify the permission(s) they require. Specifically, we first collect all calls to the third-party libraries. Then we decompile the byte code of the libraries. Finally, we perform the same reachability analysis starting from the invoked methods to identify all the SDK APIs that may be invoked and map them to the required permissions, which is similar to Stowaway [14]. Droid M+ currently does not support native libraries.

4.2 Permission Annotations

As per the Google guidelines [17] and prior studies [41], users are more likely to grant a permission if the developer provides an explanation for why it is needed. Developers should also provide feedback if a permission request is denied and accompany this feedback with a button leading to the system settings for enabling the permission(s). Unfortunately, while it is easy to automatically identify all the required permission(s) of a functionality, automatically generating the corresponding explanations and feedback is much harder. Hence, our current design seeks developers' help for generating the explanations and the feedback. To ease this process, we use the customizable Java annotation system [32] to capture the explanations. Lst. 1 provides an example of the annotation we use for declaring dangerous permissions and providing their justifications. Starting with `@Permission`, the annotation includes: (a) a name for the functionality; (b) an array of permissions, where each array element is a tuple `<perm, reason, feedback>`. `perm` denotes the requested permission, `reason` denotes the optional justification, and `feedback` denotes the optional message to be shown if the permission is denied.

For example, the "Attach photo to task" functionality of Any.Do could be annotated as:

Listing 1: Permission Annotation.

```
1 @Permission(
2   functionality ={"Attach photo to task"},
3   request={
4    {"READ_EXTERNAL_STORAGE", "Require storage to access your photos.", "You won't be
          able to attach photos."}
```

```
5  })
6  public void fromGallery() {
7    // code
8  }
```

Automatic Population: Assuming that a functionality has a single purpose, we ask developers to only provide the annotation once, as all accesses to the same protected API will share the same purpose. Given this, we automatically place the annotation at the entry point of each functionality, with two exceptions viz., background services and libraries. Background services are different since they are not a subclass of `Activity` and thus, cannot invoke the `requestPermission` method to prompt users. Libraries are different for many reasons. First and importantly, a library may be used in many functionalities, including background threads. Besides, the `onRequestPermissionsResult` method is bound to each Activity, and so if a library is used in different Activities, it also creates confusion. Second, it is unreasonable to ask first-party developers to provide explanations for why a third-party library needs a permission(s). Instrumenting libraries distributed in binary format requires additional effort. Thus, Droid M+ places the annotation at the method where the background services are started (`startService`) and where the library methods are invoked.

4.3 Compiler Extension

We use Droid M+'s compiler extension to interpret the permission annotations and generate the corresponding code. For each required permission, we use Google's example code [19] as the template towards generating the code:

Listing 2: Generated code.

```
1  SuitableMethod (...) {
2    // begin of template
3    if (ActivityCompat.checkSelfPermission(this, perm) !=
4      PackageManager.PERMISSION_GRANTED) {
5      if (ActivityCompat.
6      shouldShowRequestPermissionRationale(this, perm)) {
7        // display reason
8        ActivityCompat.requestPermissions (...);
9      } else {
10       // display feedback
11     }
12     return;
13   } else {
14     WrapperMethod();
15     return;
16   }
17   // end of template
18 }
19
20 @Override
21 public void onRequestPermissionsResult(int requestCode, String[] permissions,
22   int[] grantResults) {
23   // length will always be 1
24   if (permissions[0] == permission) {
25     if (grantResults[0] == PackageManager.PERMISSION_GRANTED) {
26       WrapperMethod();
27     } else {
28       // display feedback
29     }
30     return;
31   }
32 }
```

Here the `perm`, `reason`, and `feedback` are from the annotation. If the `reason` or the `feedback` is empty, we use the string "`{functionality} requires {perm}`". Our compiler extension ensures that the `functionality` cannot be empty.

While populating the template is straightforward, the challenge is determining where the permission should be requested. In [28], Livshits *et al*. proposed four properties for a valid prompt placement: ***(a) Safe***: Every access to the protected resource is preceded by a prompt check; ***(b) Visible***: No prompt is located within a background task or a third-party library; ***(c) Frugal***: A prompt is never invoked unless followed by an access to a resource; and, ***(d) Not-repetitive***: A prompt is never invoked if the permission was already granted.

In Android M, since a call to `checkSelfPermission` always guarantees the not-repetitive property and we have already annotated background services and calls libraries differently, we will focus on safety and frugality. To be frugal, we want to place the permission request as close to the resource access as possible, which also makes the request more likely to be *in-context*. However, the current design of the Android M's permission model makes it hard to implement this placement strategy. In particular, as already shown in the code template, `requestPermissions` is an asynchronous method and thus, when it is invoked, the execution will not be blocked. Hence when the execution reaches the next statement, the permission(s) may not be granted yet and invoking the protected API can crash the app. The standard way is to immediately exit the current method after requesting the permission. At the same time, after the user responds to the permission requests, the execution is resumed in the `onRequestPermissionsResult` callback function instead of the statement following `requestPermissions`. The problem is that if local variables are used in the access to the protected APIs, then they will not be accessible in the callback function; similarly and more fatally, if the method that accesses protected API returns a value (*e.g.*, location), then we have no way to return that value to the caller. Due to this problem, we choose to sacrifice some degree of frugality to avoid the need to drastically refactor the code.

Placement Algorithm: For each functionality that has an annotation, we use a placement algorithm to insert the "permission requesting" code. The algorithm is similar to the one proposed in [28], with two key differences. First, as mentioned above, because of framework support, we do not need to consider the non-repetitive constraint. Second, our algorithm does not try to avoid third-party libraries and background services because they are *not* annotated. Instead, we walk up the dominator tree to avoid each method whose return value depends on the protected API and is used by its caller(s).

First, for each annotated functionality, we initialize a job queue into which pairs `<sensitive call, current method>` are inserted. Here, a sensitive call denotes an invocation to a SDK API, a library method, or a background service that require permission(s). For each pair in the queue, we perform a backward search to check if the permission has already been requested. Note that according to Android's documentation [18] because the permission group may change in the future, developers should always request for every permission even though another permission in the same permission group may already be asked. Thus, when checking for existing permission requests, we do not consider (1) whether a permission within the same permission group has been requested and (2) whether a permission that implies current permission (*e.g.*, `WRITE_EXTERNAL_STORAGE` implies `READ_EXTERNAL_STORAGE`) has been requested.

If a permission has not been requested, we check whether the current method is a suitable method. A method is suitable if (1) it is a void method, (2) its return value has no dependencies on the sensitive call, or (3) its return value will never be used. If the current method is not suitable, for each call site of the current method, we push a new job pair `<current method, call site>` into the queue.

Once a suitable method is found, we place the permission request inside the method. We first create a wrapper method that replicates the code from the sensitive call to the end (return) of that branch. If the wrapper method depends on local variables, we use a `map` to store those variables before requesting the permission and retrieve them inside the wrapper method. After creating the wrapper method, we insert the permission request template right before the sensitive call and populate it with correct annotations and the generated wrapper method, as suggested in Lst. 2. Note that although some of the code after the template will become dead because the execution will always return before reaching that code, our current design does not try to eliminate it; instead, we rely on the existing dead code elimination pass of the compiler to eliminate this unreachable code.

The above process is repeated until the queue is empty. Note that because the entry point of a functionality is always a void method, this loop is guaranteed to terminate.

Background Services: Droid M+'s placement algorithm can handle almost all cases, but it cannot handle exported background services. These are services that can be started through a "broadcasting intent". Since such services can be started by the Android framework, if they require permissions, Droid M+ must request the permissions up-front. We identify such services by parsing the manifest file. For any service with attributes `enabled = true` and `exported = true` and requires permission(s), we add the permission requests in the `onCreate` method of the main Activity.

Denied Permissions: The Android M permission model places an onus on the developer to correctly implement denied (or revoked) permission call-backs. Droid M+ facilitates his by inserting the skeleton code that needs to be filled out (as seen in Lst. 2). Google's guidelines suggest that developers should handle the denied branch more gracefully than just displaying an error. We point out that Droid M+ does not make the implementation of the handler for a denied branch any easier or difficult. The developer has to still correctly implement the handler for the denied branch. By default, when a permission is denied, Droid M+ displays an error message and simply exits; this will cause the `onStop()` of the app's current activity to be invoked to do any necessary bookkeeping before terminating. It is not ideal as the user may need to restart the app. However, it at least will not cause unexpected program malfunction or crashes. We further discuss how to handle denied permissions in §7.

Critical Permissions: Droid M+ currently does not support identifying critical permissions that must be requested up-front. For such permissions, developers have to add the requests manually and provide proper education on the welcome screen [17]. However, because granted permissions can always be revoked by users at anytime (through system settings), the code snippets that Droid M+ inserts are still necessary for the correct functioning of the app.

5 EVALUATIONS

In this section, we present the Droid M+'s evaluations. Our evaluation focus on answering two questions: **(a)** How applicable is Droid M+ i.e., how well can it handle today's apps on the Play Store? and, **(b)** How good is our permission request placement algorithm?

We implement Droid M+'s static analyzer based on the soot [40] static analysis framework. We use apktool [7] and androguard [3] to analyze existing apps. Annotation interpretation and code insertion is done based on Java JDK 1.8 and using the Java parser [24].

5.1 Applicability

We design Droid M+ to be a source code level tool set. Unfortunately, as there are only a limited number of open sourced Android apps, we evaluate Droid M+ in two ways. First, using RingDroid as a case study, we showcase how Droid M+ would work on real world Android apps. Then we analyze 100 top apps from the Google Play store and quantify how many apps can be handled by Droid M+.

Case Study: Ringdroid. Ringdroid [35] is an open source app that records and edits sounds. In this case study, we use the commit `955039d` that was pushed on December 2, 2016; actual source code and line numbers can be found in [35]. Although Ringdroid was developed by Google and was targeting the latest Android SDK (API level 25), it surprisingly does not support revocable permissions (built against API level 22). Instead, it just wraps access to protected APIs with a `try` block and catches the thrown `SecurityException`. This makes it a good example to showcase the benefits of Droid M+.

Ringdroid requires four dangerous permissions: `READ_CONTACTS`, `WRITE_CONTACTS`, `RECORD_AUDIO`, and `WRITE_EXTERNAL_STORAGE`. The static analyzer in Droid M+, finds 11 functionalities that require permissions. Among them 8 are background functionalities and the remaining 3 are associated with button clicks. 9 requests are finally inserted by Droid M+ (2 are redundant). In all the 9 cases, the requests were inserted immediately before the associated sensitive call happens, because the containing methods are all void methods.

- The first functionality is the `onCreateLoader` interface, implemented by `RingdroidSelectActivity` when it returns a `CursorLoader` that requires the `STORAGE` permission. This method is invoked in the background by the Android framework when `LoadManager` related methods are invoked (*e.g.*, in the `onCreate` method at line 151 and 152). Droid M+ insert the requests before line 151 (152 is covered by this request), and the remainder of the code from line 151 to line 187 is replicated in a wrapper method.
- The `LoadManager` is also invoked in the `refreshListView` method. Since this method can be called from other UI events such as `onCreateOptionsMenu`), Droid M+ placed a request for the `STORAGE` permission at line 528 with lines 528 and 529 replicated in a wrapper.
- In the `onDelete` method invoked from a button click callback function, Droid M+ inserted a request for the `STORAGE` permission at line 473; the remainder of the function is replicated in a wrapper.
- The `ReadMetadata` method also requires the `STORAGE` permission. However, since it is not part of an Activity, Droid M+ has to move it up to the `loadFromFile` method of the `RingdroidEditActivity`. Droid M+ inserted a request at line 598, with the remainder of the function replicated in a wrapper. Note that inside the `loadFromFile` method, there is a background thread that also requires the `STORAGE` permission. However, as the permission has already been requested, no request is inserted for this thread.
- The `MICROPHONE` permission is also required by the `RecordAudio` method of the `SoundFile`. However, as it is invoked from a background thread, the request is inserted before the creation of the

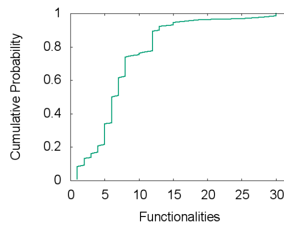

Figure 8: CDF of apps vs functionalities requiring permission(s).

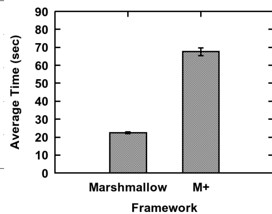

Figure 9: Average Compilation Time.

thread at line 755 inside the `recordAudio` method; the rest of the function is replicated in a wrapper.

- Three methods of the `SoundFile` class: `ReadFile`, `WriteFile`, and `WriteWAVFile` require the `STORAGE` permission. But as they are invoked in background threads, namely, one created in the `loadFromFile` method discussed above and another in the `saveRingtone` method, the request is inserted inside the creation method. In particular, this is done at line 1225. The rest of the function replicated in a wrapper.
- The `ChooseContactActivity` implements the `onCreateLoader` interface which will return a `CursorLoader` that requires the `CONTACTS` permission. The loader is initialized in the `onCreate` method. Thus, a request is inserted at line 129 with the code from line 129 to line 139 replicated.
- The `afterTextChanged` method handles UI events. A request to `CONTACTS` permission is inserted to enable `restartLoader`, at line 181; this line (line 181) is replicated in a wrapper.
- The `assignRingtoneToContact` method, invoked from a button click callback function, also requires the `CONTACTS` permission. A request is inserted at line 154 with the remainder of the function replicated in a wrapper.

The reasons for requesting these permissions are self-explanatory; thus, we omit the annotations that were created.

Extended Measurement: To understand the applicability of Droid M+ to general apps, we perform the following measurements. First, since Droid M+ cannot handle native code that requires dangerous permissions (*e.g.*, write to the external storage), we analyze how many apps from our measurement study (1) contain native code and (2) require external storage read/write permissions[1]. We found that 546 apps from among the 7000 apps match these criteria and thus, might not be handled by Droid M+.

Next, we analyze 100 top Android M apps that do not have native code and require dangerous permissions. We choose Android M apps to ensure that we can also compare the permission requests placements. From among these apps, we found 698 functionalities that would request a total of 158 permissions up-front. On average, each application has 7 different functionalities that require permission(s); 90% of the apps have 13 "permission requiring" functionalities or fewer. Fig. 8 presents the CDF of the number of permission requiring functionalities of the analyzed apps. Among these functionalities that require permissions, 203 are Activities, 232 are UI event handlers, 201 are third-party libraries, and 60 are background threads. Most of the functionalities (579) require only one permission, 98 require two, 21 require three or more.

[1] We have verified experimentally that this is the only common permission that is used within native code.

Group	Permissions
LOCATION	ACCESS_COARSE_LOCATION, ACCESS_FINE_LOCATION
CONTACTS	READ_CONTACTS, GET_ACCOUNTS
CALENDAR	READ_CALENDAR
PHONE	READ_CALL_LOG, READ_PHONE_STATE
MICROPHONE	RECORD_AUDIO
STORAGE	WRITE_EXTERNAL_STORAGE

Table 1: Dangerous permissions requested by Any.Do.

Regarding request placement, as was discussed in § 4, Droid M+ cannot place an annotation in a non-void method whose return value depends on the permission and would be used by its caller(s). To understand how common this situation is, we performed a more conservative measurement – how many non-void methods contain access to protected APIs. We found that 43% of the methods would returns a value and thus, the placement of the permission request may have to moved up to its callers. We also found that 11.3% of these requests might need to be placed on the entry method because all the other methods inside the functionality return a value.

Finally, by applying Droid M+ to these apps, only 48 permissions will be requested up-front (instead of 158), while the remaining will be asked instead, in-context; this corresponds to a decrease of 69.62% in permissions asked up front.

5.2 Quality of Request Placement

Next, we evaluate the effectiveness of our request placement strategy (*i.e.*, examine whether the processed app actually follows Google's guidelines [17]). To show that this is indeed the case, we first use an existing Android M app that supports revocable permissions to show that Droid M+'s placement matches the placement of the developers, *i.e.*, we are as good as manual placement. Then we reason about why the Droid M+-processed app will follow the guidelines.

Case Study: Any.do. In this case study, we analyzed the free version of Any.do [5], one of the most popular "To do" and task list apps on Google's Play Store with a total number of installations between 10 and 50 million. This app was updated last on November 21, 2016 and it supports the new revocable permission model. We input the downloaded APK to Droid M+'s static analyzer and find that the app requires 9 dangerous permissions (see Table 1).

Manual request placement. Upon launch, the app requests users for permissions for `LOCATION`, `CONTACTS`, and `PHONE`. No explanation or education is provided with regards to any of these. Upon further investigation, we find that none of the requested permissions are critical and the app would continue to function even if no permissions are granted. In addition to these up-front requests, all used permissions are also requested in-context; so they can be dynamically revoked.

- The `LOCATION` permission is used in four functionalities: one for attaching to a scheduled task for location based reminders, two for sharing the location to custom analytics classes, and one in a third-party location-based library.
- The `CONTACTS` permission is used in two functionalities: when the user shares a task with her friends and to display the caller of a missed phone call. If the permission is not granted, no feedback is provided and the corresponding functionality is not performed.

- The PHONE permission is used when the user seeks to be notified with regards to missed phone calls. If the permission is not granted, feedback "Missed call feature requires phone permission" is provided. It is also being used in the two analytics classes.
- The CALENDAR permission is requested in-context to cross-reference the tasks with the user's calendar for conflict detection, etc. If the permission is denied, the feedback "Post meeting feature requires calendar permission" is provided.
- The MICROPHONE permission is requested correctly in-context when the user wants to add a voice message to a task that she schedules for later. If the permission is not granted, the feedback "Recording permission is needed" is provided.
- The STORAGE permission is also requested correctly in-context when the user wants to add a photo or a document to her scheduled task. If the permission is not granted, the feedback "This feature requires access to external storage" is provided. But this permission is also used by three different third party libraries.

In summary, to support revocable permissions, all permissions are checked before use; however, only 3 (50%) are requested in-context. Moreover, Google's guidelines are not adequately followed: (1) non-critical permissions are requested up-front, (2) no education is provided. While admittedly in many cases the reasons for requesting a permission can be inferred from the functionality, some cases are less intuitive (e.g., displaying the callers for missed calls); (3) when permissions are denied, feedback is not always provided and the functionality silently fails (*e.g.*, share with contacts).

Droid M+'s request placement. Because Any.Do does not always request the permission in-context, we cannot do a one-to-one matching with its request placement; instead, we perform two measurements: (1) we check if their in-context requests matches Droid M+'s and (2) we check if their in-context checks matches the requests made by Droid M+.

For the three already in-context permission requests, because they are all inside click handlers, our placement is roughly the same as existing placement. There is a difference due to the fact that, to support ascynchronous permission requests, Any.do's code has already been refactored so that the protected APIs are all accessed inside void methods and the permissions are asked before the invocation of the methods. This makes Droid M+'s placement one level deeper than existing placement (permissions are asked inside the method). Unfortunately, since we do not have an older version of this app, we cannot verify if without such factoring, our algorithm would have yielded the same placement.

In the remaining places where permissions are checked but not requested, Droid M+'s static analyzer identified all of those as places where permissions should be requested. From among them, 16 of the accesses are inside third-party libraries and 4 of them are inside background threads. We have also manually checked whether our request placements are in-context and the answer is yes.

Discussion: Although Any.Do is a single case study, we argue that the evaluations carry over to other apps. Specifically, the quality of the request placement is assessed based on whether the request would be *in-context*. In Droid M+ we achieve this goal in two steps: (1) we segment the code into different functionalities based on unique entry points and (2) we try to place the request as close to the permission use as possible. We believe this is effective for the following reasons. First, our functionality identification process essentially segments the app into Activities, UI events handlers, and background threads. Since most UI events handlers are single-purposed and very simple, permissions requests should also be in-context. Activities typically, at most represent a single context; thus, any request within an Activity is highly likely to be in-context. Furthermore, the quality of the placement inside a Activity is further improved in the second step (see above). For background threads, due to the limitation of the Android framework, the request would is less likely to be in-context; however, Droid M+ still improves the quality of the request by supporting the insertion of justifications. Finally, note that the fallback solution for placement is to place the requests in the main activity. However, this need did not arise in our test cases.

5.3 Performance

Next, we seek to experimentally quantify the overheads that accompany Droid M+ on the developer side. Towards this, again consider Ringdroid, the open source app that was previously described and modify it to support revocable permissions. We make two constructs; in one the API of Android M is used, and in the other system is used.

Compilation Overhead: Our case study was performed on Android Studio 1.4.1, running on a laptop with a quad core Intel Core i7 2.00GHz CPU with 16GB of RAM and a hard drive of 1TB at 5400 rpm. We seek to quantify the increase in the compilation time with Droid M+ compared Android M's API, because of the additional steps we add. We perform a clean before each compilation to achieve the maximum overhead that can be incurred. We perform 20 runs and on average (Fig. 9), Ringdroid using only Android M compiles in 22.43 seconds while using Droid M+ it compiles in 67.53. This corresponds to an increase of approximately 201%. Although this overhead is significant, it is experienced only when the app is compiled on the developer's computer (it does not affect the user).

6 RELATED WORK

In this section we discuss relevant related work.

Android Permission System: Android system prior M uses ask-on-install model to request permissions. Many research has shown the ineffectiveness of this design and the need for better permission management. First, few people would review the requests and even fewer can correctly understand how permissions are mapped to sensitive resources [15, 22, 26, 44]. The problem of apps routinely abusing sensors has been highlighted in [36, 42].

Different techniques to help users manage permissions have been proposed. Stowaway [14] can determine the set of API calls that an app uses, and map those API calls to permissions towards detecting privilege escalation. Ismail *et al*. [23] proposed using crowdsourcing to find minimal sets of permissions for apps. Liu *et al*. [10] proposed a personal assistant to help manage permissions. TaintDroid [12] performs dynamic taint analysis to help user uncover cases of potential permission misuse. Livshits *et al.* provides an automated way of inserting prompts before sensitive API calls [28]. A field study conducted by Wijesekera *et al*. [47] found that apps routinely abuse permissions once they are granted and that users would like greater control over sensitive information. Dynamic permissions

At most one	At least one	5 or less	5 or greater
49%	51%	60%	40%

Table 2: Permissions per functionality

(such as those in Android M and iOS) address some of the concerns but permissions once granted, are rarely revoked in practice. The study makes a case for finer grained, more intelligent permission management than that with Android M.

Android M: Andriotis *et al*. [2] conducted a study on users' adaptation of Android M and found that in general, users greatly prefer the new model to the old one. The results of the study further highlights the need for developers to migrate their apps to the new permission model. revDroid [13] empirically analyzes how often app crashes can occur due to permission revocation, in off-the-shelf Android applications targeting Android M. Their study highlighted the dangers of not handling permission revocation correctly and as such, Droid M+ builds on their results by making it easier for developers to correctly manage dynamic permissions.

Annotations: Google [11] has developed a set of annotations for permissions in Android but it is used only in code inspection tools such as *lint*. APE [29] uses an annotation language and middleware service that eases the development of energy-efficient Android apps. In Java, AOP is used by frameworks like Spring [6] to provide declarative enterprise services and to allow users to implement their own custom aspects. In our work, we automatically annotate Android code, to help developers adopt Android's new permission model.

7 DISCUSSION

Automated Annotation Extraction: Currently, Droid M+ cannot automatically generate the `functionality`, `reason`, and `feedback` within the annotations as it requires automated reasoning about the context. However, researchers have demonstrated that it is possible to use natural language processing (NLP) over apps' descriptions to infer their functionalities [33]. Using NLP, we may also be able to extract the context information directly from the target app. We plan to explore this direction in the future.

Per-Functionality Permission: Although the new Android permission model is a big step forward from its old model, it is still not ideal. Specifically, once a user grants a permission, the permission will be shared across all the functionalities that will require this permission. For example, when a user grants the `Location` permission to a "map" app to find her current location, this permission can also be used by third party libraries [48] and violate the user's privacy. A recent study [47] shows that for the "ask-on-first-use" strategy Android M and iOS employ, the participants subsequent decisions on whether to grant a permission would match their first decision, only about half of the time (51.3 %). This observation is also intuitive; for example, in the above example, while the user is willing to grant the `Location` permission to the core functionality of the map app, she may not want the advertisement library to know her current location.

To quantify the extent to which permissions are carried over across functionalities, we analyzed the same 1638 apps that contain revocable permissions (recall Section 3). Table 2 shows the results. In particular, 51% of the apps share at least one permission across multiple functionalities. For these apps, in 60% of the cases, the permission is shared by up to 5 functionalities; in the remaining cases, the permission is shared between 5 and 20 functionalities.

We discuss two further improvements that can address this problem. Under the constraint of the current Android M permission model, one solution is to provide more education to the user with aggregated explanation messages from multiple functionalities. This solution will offer the best transparency about all possible uses of any permissions. With Droid M+, we have in fact implemented this solution and provided an example screenshot shown in Fig. 10. If a user chooses to approve the permission use, he or she will be fully aware that the permission is enabled in all functionalities. Of course, the downside of such an approach is that it burdens the user with too much information and puts the onus on her to revoke a permission later to protect her privacy.

A better solution is to extend the ask-on-first-use strategy to work on a triplet `<app, functionality, permission>` instead of the pair `<app, permission>`. This means that a permission is approved per functionality. If the same permission is used in several functionalities, a user can independently approve and deny a subset of the same. With the help of Droid M+, an existing app can be easily ported to this new per-functionality permission model as Droid M+ can already help developers to identify and annotate the different functionalities of their apps, identify the required permission(s) for each functionality, and automatically generate the permission request code. To enforce this fine-grained permission model though, we can either insert additional code to maintain the per-functionality permission status, or require proper support from the Android OS.

Unpredictable App Behaviors: Since Droid M+ introduces non-trivial changes in the source code, a valid question is whether the changes break the app in some way or induce unpredictable behaviors. Although we only present case studies of two apps, we tested most apps in our data set to make sure that they worked as expected.

If requested permissions are always granted by the user, it is easy to see why Droid M+ will not cause any change in app behavior. It simply adds permission checks that if granted, will result in the app executing as before. However, the denied execution branch is more problematic as discussed in §4. Towards understanding the logic relating to a denied permission, consider first how Android M deals with apps that have not updated to the newest SDK. If an API requires a permission that is withdrawn, the app will simply throw an exception and most likely crash. Droid M+ helps resolve this by inserting a check for needed permissions before their use and indicating, using code comments, where the permission denied call-back should be implemented (as can be seen in listing 2). Of course, the developer still needs to fill the handler code for denied permissions. We are not making this job any easier or difficult. By default, Droid M+ displays an error message and exits if a permission is denied. This is to prevent the app from entering any inconsistent or bad state (e.g., as simply aborting the original execution flow will likely cause). Alternatively, depending on the nature of the denied permissions, we can generate more graceful handlers. Android APIs that require permissions generally fall under two categories: (1) those that return immediately with results (e.g., get the last location); (2) those that register a callback (e.g., receiving location updates). For (1), we can return randomized values so as to keep the program running (a strategy that has been used in prior work [16]); for (2), we can simply skip the registration of the callbacks because app developers cannot assume callbacks will always occur (e.g., at a given

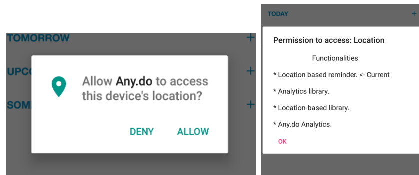

Figure 10: Any.do current version vs with Droid M+.

rate). This strategy is not likely to cause the program to malfunction. However, there is no guarantee that the two strategies will not lead to unexpected/unsafe program states.

Other issues: As discussed in § 3, the manual process of checking whether permissions are legitimately asked upfront is subjective. There are techniques that can be used to alleviate the subjectivity (*e.g.*the crowdsourcing technique [27]). Further, there is almost no work on understanding the privacy implications of normal permissions as defined by google. We defer the study of these issues, which we believe can supplement the utility of Droid M+, to future work.

8 CONCLUSIONS

Given criticisms on Android's permission models, Google revamped the model in Android 6.0. In this paper, we find via an in depth measurement study that many apps from the Google Play store have either not migrated to the new model, or do not follow Google's guidelines for adoption, effectively. We find some evidence that this unsatisfying status quo could be due to the lack of tools that allow developers to easily adopt the new model. Towards addressing this shortfall, we design and implement Droid M+, a tool that helps developers refactor their code to adopt the model. We show that Droid M+ can help developers in evolving their legacy code to adhere to Google's guidelines, via case studies and general app evaluations.

9 ACKNOWLEDGMENTS

This research was partially sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. The work was also partially supported by NSF Award 1617481. The authors would like to thank our shepherd and the anonymous reviewers for their constructive feedback.

REFERENCES

[1] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. Mazurek, and C. Stransky. 2017. Comparing the usability of cryptographic APIs. In *IEEE S&P*.
[2] P. Andriotis, M. Sasse, and G. Stringhini. 2016. Permissions snapshots: Assessing users' adaptation to the Android runtime permission model. In *IEEE Workshop on Information Forensics and Security (WIFS)*.
[3] Androguard. 2016. Tool to play with apk files. (2016). https://goo.gl/edcClw.
[4] Android. 2017. Android Share. (2017). https://goo.gl/9kiCgg.
[5] Any.do. 2016. To-do list, Task List. (2016). https://goo.gl/rPpZq8.
[6] AOP. [n. d.]. Aspect Oriented Programming. ([n. d.]). http://goo.gl/1UnkGS.
[7] Apktool. 2016. Reverse engineering apk files. (2016). https://goo.gl/JCh7U7.
[8] AskMD 2016. AskMD. (2016). https://goo.gl/3D5Vvw.
[9] K. Au, Y. Zhou, Z. Huang, and D. Lie. 2012. Pscout: analyzing the android permission specification. In *ACM CCS*.
[10] B.Liu, M.S. Andersen, F.Schaub, H.Almuhimedi, S.Zhang, N.Sadeh, Y.Agarwal, and A.Acquisti. 2016. Follow My Recommendations: A Personalized Privacy Assistant for Mobile App Permissions. In *ACM SOUPS*.
[11] Google Developers. 2016. Improving Code Inspection with Annotations. (2016). http://goo.gl/qSE9dh.
[12] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. Cox, J. Jung, P. McDaniel, and A Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems* (2014).
[13] Z. Fang, W. Han, D. Li, Z. Guo, D. Guo, X. Wang, Z. Qian, and H. Chen. 2016. revDroid: Code Analysis of the Side Effects after Dynamic Permission Revocation of Android Apps. In *ACM ASIACCS*.
[14] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. 2011. Android permissions demystified. In *ACM CCS*.
[15] A. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. 2012. Android permissions: User attention, comprehension, and behavior. In *ACM SOUPS*.
[16] Patrice Godefroid, Michael Y Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Queue* 10, 1 (2012), 20.
[17] Google. 2016. Material Design Patterns. (2016). https://goo.gl/QQcfEv.
[18] Google. 2016. Requesting Runtime Permissions. (2016). https://goo.gl/0enMi9.
[19] Google. 2016. Runtime Permissions Basic Sample. (2016). https://goo.gl/t59Dw9.
[20] Google. 2017. Google Play Store. (2017). https://goo.gl/kN0Nhz.
[21] Google. 2018. Play Store Top Charts. (2018). https://goo.gl/uPr4nj.
[22] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. 2014. Checking app behavior against app descriptions. In *ICSE*.
[23] Q. Ismail, T. Ahmed, A. Kapadia, and M. Reiter. 2015. Crowdsourced exploration of security configurations. In *ACM CHI*.
[24] Java 1.8 2016. Parser and Abstract Syntax Tree. (2016). https://goo.gl/qI1f34.
[25] J. Jung, S. Han, and D. Wetherall. 2012. Short paper: Enhancing mobile application permissions with runtime feedback and constraints. In *ACM workshop on Security and privacy in smartphones and mobile devices*.
[26] P. Kelley, S. Consolvo, L. Cranor, J. Jung, N. Sadeh, and D. Wetherall. 2012. A conundrum of permissions: installing applications on an android smartphone. In *International Conference on Financial Cryptography and Data Security (FC)*.
[27] J. Lin, S. Amini, J. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. 2012. Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing. In *ACM UBICOMP*.
[28] B. Livshits and J. Jung. 2013. Automatic mediation of privacy-sensitive resource access in smartphone applications. In *USENIX Security*.
[29] N. Nikzad, O. Chipara, and W. Griswold. 2014. APE: an annotation language and middleware for energy-efficient mobile application development. In *ICSE*.
[30] H. Nissenbaum. 2004. Privacy as contextual integrity. *Wash. L. Rev.* (2004).
[31] K. Olmstead and M. Atkinson. 2015. Apps Permissions in the Google Play Store. (2015). http://goo.gl/ph7KGk.
[32] Oracle. 2016. Java SE Annotations. (2016). http://goo.gl/g9b0Dh.
[33] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. 2013. Whyper: Towards automating risk assessment of mobile applications. In *USENIX Security*.
[34] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1354–1365.
[35] Ringdroid. 2016. Ringdroid. (2016). https://goo.gl/MhLqGW.
[36] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. 2011. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones.. In *NDSS*.
[37] Y. Shao, J. Ott, Q.Chen, Z. Qian, and Z. M. Mao. 2016. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *NDSS*.
[38] Y. Smaragdakis, G. Balatsouras, G. Kastrinis, and M. Bravenboer. 2015. More sound static handling of Java reflection. In *Asian Symposium on Programming Languages and Systems*. Springer.
[39] Songily 2016. SONGily. (2016). https://goo.gl/fFWI1m.
[40] Soot 2016. Soot - A Java optimization framework. (2016). https://goo.gl/UsmKcC.
[41] J. Tan, K. Nguyen, M. Theodorides, H. Negrón-Arroyo, C. Thompson, S. Egelman, and D. Wagner. 2014. The effect of developer-specified explanations for permission requests on smartphone user behavior. In *ACM CHI*.
[42] Z. Templeman, R and. Rahman, D. Crandall, and A. Kapadia. 2012. PlaceRaider: Virtual theft in physical spaces with smartphones. *arXiv:1209.5982* (2012).
[43] TheScore 2016. theScore: Sports Scores. (2016). https://goo.gl/iefw9C.
[44] Christopher Thompson, Maritza Johnson, Serge Egelman, David Wagner, and Jennifer King. 2013. When it's better to ask forgiveness than get permission: attribution mechanisms for smartphone resources. In *ACM SOUPS*.
[45] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. 2012. Permission evolution in the android ecosystem. In *ACSAC*.
[46] WhatsApp 2016. WhatsApp Messenger. (2016). https://goo.gl/W1QcPv.
[47] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. 2015. Android permissions remystified: A field study on contextual integrity. In *USENIX Security*.
[48] W. Xu, F. Zhang, and S. Zhu. 2013. Permlyzer: Analyzing permission usage in android applications. In *IEEE Symposium on Software Reliability Engineering (ISSRE)*.