On Understanding Permission Usage Contextuality in Android Apps

Md Zakir Hossen and Mohammad Mannan

Concordia Institute of Information Systems Engineering
Concordia University, Montreal, Canada
{m_osssen,mmannan}@ciise.concordia.ca

Abstract. In the runtime permission model, the context in which a permission is requested/used the first time may change later without the user's knowledge. Our goal is to understand how permissions are *requested* and *used* in different contexts in the runtime permission model, and compare them to identify potential inconsistencies. We present ContextDroid, a static analysis tool to identify the contexts of permission request/use, and analyze 6,790 apps (chosen from an initial set of 10062 apps from the Google Play Store). Our preliminary results show that apps often use permissions in dissimilar contexts: 15% of the apps use the permissions in contexts where users are not prompted and may be unaware; 46% of the apps use the permissions in multiple contexts while only 20% of the apps request permissions in multiple contexts. We hope our study will attract more research into non-contextual usage (and possible abuse) of permissions in the runtime model, and may spur further work in the design of finer-grained permission control.

Keywords: Android · Smartphone · Permission Model · App Analysis.

1 Introduction

The runtime permission model enables context-based control of resources. The new model was introduced in Android 6.0 to facilitate user decision by providing situational context (e.g., current state of the app representing the purpose of resource access) when the permissions are requested for the first time. However, an app can trick a user to grant a permission in a valid context, and then use it in malicious/unexpected contexts without the user's consent/knowledge. For example, accessing GPS when the user attempts to find the current location in a map is a valid context, but accessing GPS when the app is in the background may be unwanted. Indeed, such contextual differences defy user expectations [10, 13, 14]. In contrast to the contextual analysis of resource access in the old install-time permission model [7, 8, 15], such studies in the runtime model are limited. Wijesekera *et al.* [13] modify an older version of Android to analyze contextual integrity of Android apps and conclude that users mostly rely on the surrounding context in which a permission is requested to grant/deny a permission [14]. In this work, we focus on regular apps that are developed/adapted for the runtime

permission model and perform the first study to understand the contextual use of resources in the runtime permission model using 6,790 regular apps.

We develop ContextDroid, a static analysis tool that extracts the *context* when a permission is requested and used in an app using an app-wide call graph. We define a context based on the active User Interface (UI) component that is requesting the permission (and using it, if granted). To differentiate between user activities and identify contexts, we leverage five Android components (`Activity`, `Fragment`, `Service`, `AsyncTask` and `Broadcast Receiver`), representing different types of UI and functionality. There are several challenges in statically extracting contextual information from Android apps, e.g., handling obfuscation introduced by widely-used ProGuard [1] and similar tools. While Android framework classes and methods are excluded from obfuscation, classes derived from support libraries that are shipped with the APK are obfuscated by ProGuard (unless configured otherwise by the developer). We must identify `Fragments` and permission related APIs that are derived from the support libraries. We use a combination of an extended call graph and sub-signature matching to identify contexts in obfuscated code.

Our evaluation reveals a large difference between permission request vs. use. Only 20% apps request permissions in multiple contexts, while 46% of the apps use the permissions in multiple contexts, indicating that apps use permissions more often than they request for and in varying contexts without the user's knowledge. Moreover, we find that apps request a permission in one context without using it and use the permission in another context without requesting the user. The context of permission use doesn't match with the context in which it is requested in 15% of the apps. Our findings suggest that apps often fail to provide situational context while requesting permissions – one of the best practices suggested by Google [3] in the runtime permission model.

Contributions. (i) We present ContextDroid that statically extracts the contexts in which the permissions are requested and the contexts in which they are used, by leveraging the call paths that lead to sensitive API calls associated with permissions. Our methodology for context identification may be useful for other studies. (ii) We analyze 6,790 (chosen from 10,062) regular Android apps to understand contextual resource usage under the runtime permission model. To the best of our knowledge, this is the first study on contextual resource usage in the runtime permission model, involving apps that target only the new model. (iii) Our tool, albeit primitive, can be used by app market maintainers to identify apps that may be violating user expectation and subject them to further analysis. We will make our tool and source code publicly available.

2 Background

In this section, we briefly describe the necessary background of the Android components and its permission model.

Android Components. `Activity`, `Service`, `Content Provider`, `Broadcast Receiver`, `Fragment`, and `AsyncTask` are some of the major components of An-

droid. Apart from `Content Provider` that helps manage app data, other components represent various elements associated with the UI and events. All these components have their own *life-cycle* methods, and act as entry points for the components (and the corresponding app functionality).

`Activity` and `Fragment` are foreground UI components allowing users to interact with the app. An `Activity` can represent a standalone full screen UI. `Fragment`s can be considered as UI modules representing part or full screen of an `Activity`. An `Activity` can hold multiple `Fragment`s, and a `Fragment` can be reused in multiple `Activities`. Multiple `Fragment`s inside the same `Activity` represents different UI and functionality.

`Service` is a background component that runs without any UI. `Broadcast Receiver` receives updates from the OS whenever there is a change of state and can perform tasks without interacting with the UI. App developers can also implement their own `Broadcast Receivers` and broadcast an update to trigger a background task. `AsyncTask` performs minor tasks in the background and communicates results to the UI thread.

All these components have their own entry points that can be used to perform specific tasks. Background components can also be used independently outside the visible user flow. We differentiate contexts mainly by identifying whether a permission is requested/used in any of these components.

Runtime Permission Model. Android maintains a set of dangerous permissions to protect privacy sensitive resources. While individual dangerous permissions regulate access to specific actions or tasks (e.g., READ_PHONE _STATE), they are categorized/clustered into permission groups to protect specific resources (e.g., READ_SMS, WRITE_SMS are grouped into SMS). In the runtime model, dangerous permissions are requested at runtime when the app first uses them. However, instead of showing prompt for each permission, it requests for permission for the permission group.

3 Methodology

We develop ContextDroid as a static analysis tool by leveraging app-wide call graph and Android permission mappings to extract the contexts. The app-wide call graph is generated by FlowDroid [4], a state of the art information flow tracking tool. We use permission mappings from Au *et al.* [5] and Backes *et al.* [6] to map API calls to associated permissions.

3.1 Context Definition

We define context based on what components the user is using when they are requested for a resource, or the resource is accessed. We determine this by identifying the five Android components (i.e., `Activity`, `Service`, `Fragment`, `Broadcast Receiver` and `AsyncTask`). As discussed in Section 2, these Android components represent different types of UIs and functionalities. We consider these components as the key elements representing different contexts.

We consider multiple instances of the same component type to be different in terms of context. For example, if an app uses the same permission in different `Activities`, they are considered as different contexts. If a permission is used in multiple `Fragments`, they are also considered as different contexts even if they reside inside the same `Activity`. `Activity` and `Fragment` are considered as foreground contexts as the user can directly interact with them through the UI. `Service`, `Broadcast Receiver`, and `AsyncTask` do not have their own UI, and we consider them to be background contexts. The difference in the way these components work enables us to differentiate between the active components of an app when a permission is requested or used. However, to infer what the users might consider as unexpected behavior is non-trivial, and may vary greatly depending on how they view the importance of their privacy [13]. While a finer-grained approach can be taken to further differentiate the context, we believe that our definition provides an overall view of how permissions are used in various components (contexts in our definition) by the apps.

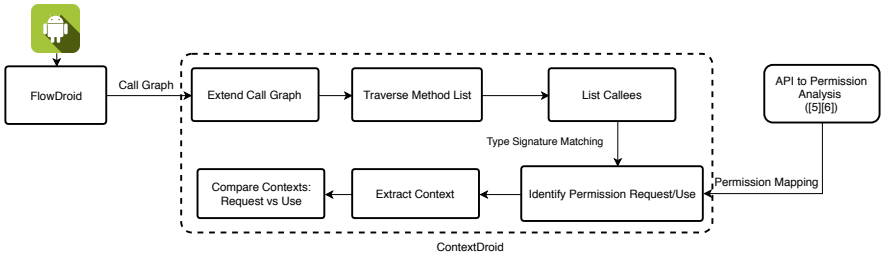

Fig. 1. Overview of ContextDroid

3.2 Identifying Contexts

We consider the following three factors while identifying the contexts in which permissions are requested and used: *Activation event* (an entry point of the call graph), *Request API* (used to show permission prompts at runtime) and *Sensitive API* (protected by dangerous permissions).

Context of Permission Use. We search each method in the call graph and identify API calls associated with permissions. We match the type signature of the APIs inside the method with APIs from the permission mappings. If we find a match, we first check whether it is a standalone method representing an *activation event*. If not, we traverse back to all the callers of that method until we find the activation events. We then extract contextual information that includes the component type (e.g., `Activity`), class name, method name and the callers of the method in which the sensitive API call is made.

We identify different instances of the same component type by using a combination of class name and component type. To handle class name obfuscation

where component type cannot be identified, we recursively traverse back to the parent classes of the method and identify whether it is a child of any of the Android component classes.

Context of Permission Request. In the runtime model, apps can request permissions by using one of the Request APIs. To infer the requesting context, we identify calls to different instances of *requestPermissions()* APIs and follow a similar approach described for permission usage to identify the active component.

However, unlike the system API calls that are not obfuscated by ProGuard (with Google Play Services library being an exception), support library APIs that are shipped with the APK can be obfuscated in the release build (unless the developer excludes certain classes from obfuscation). The support library contains APIs that also request permissions. For example, apps can request permission by *requestPermissions()* call through the *ActivityCompat* or *ContextCompat* class from the support library. To handle such instances, we use partial type signature matching to identify the context of permission request. We first identify whether the method contains permission strings, e.g., android.permission.AUDIO. If found, we examine subsequent API calls that take the permission strings as a parameter. Specifically, we identify whether the package name of the method partially matches with a support library package (e.g., *android.support.v4.a.a* partially matches with the package name of support library version 4). If a match is found, we further compare the parameter signature of the API with request APIs from the support library. If the partial type signatures match, we consider this as an instance where permission is requested.

Analyzing Contextual Differences. We analyze the contextual information associated with permission requests and usage to understand their differences. For each permission required by the app, we find whether the permission is requested and used in multiple contexts. We compare the number of contexts identified for a permission, both in terms of when it is requested and when it is used. If the number of contexts differ, we tag it as a permission that is used in unexpected and dissimilar contexts. If the numbers match, we directly compare the contexts to determine whether they match or not.

3.3 Extended Call Graph

Identifying obfuscated `Fragments` (e.g., by ProGuard [1]) that are derived from support libraries is not straight forward. To identify `Fragment` contexts that are otherwise excluded from the call graph (e.g., due to obfuscation), we use an extended call graph in ContextDroid. We first iterate through all the methods in the call graph and identify the class in which they are declared. If the component type of the class cannot be identified, we iterate through the parent classes to determine whether they can be identified as one of the contextual elements. If the component type cannot be determined, we attempt to find whether the class is derived from the support library `Fragment`.

We start with the package name of the method and perform partial matching with the support library package and iterate through the parent classes and their

package names until we find a match. Then, we extract the method list of that class. Android `Fragments` used in third party apps must override the *OnCreateView()* method. To determine whether the class is a `Fragment` component, we match the return and parameter types (i.e., sub-signature) of the listed methods with *OnCreateView()*. If the sub-signature matches, we tag it as a `Fragment` and include the methods of that class in the call graph.

4 Analysis and Results

We select apps that target the runtime permission model from AndroZoo [2]. We analyze 10062 apps and identify permission request/usage of dangerous permissions in 6790 apps. For the rest of the apps, we could not identify any instance of permission use or request, although permissions were declared in the manifest (similar to past findings, e.g., [9]). We present our contextual analysis results mainly in terms of resources (i.e., permission groups). Each resource consists of related individual permission request/usage as identified by ContextDroid.

Performance. We perform our analysis on an Intel Core i7 3.60GHz processor with 24GB of memory running Ubuntu 16.04. For an average app (of size around 20MB), our analysis takes on average under a minute, including time taken by FlowDroid. Note that we first use FlowDroid to generate an app-wide call graph; this takes most of the time in our analysis (approximately 95%). We believe that ContextDroid can be integrated into app security tools (maintained by app markets) to identify permission use in unexpected contexts.

Permission Requests. We find that about 20% of the apps request at least one permission in multiple contexts. Access to Storage (35%), Location (23%), Camera (20%), Contacts (8%), and Phone (7%) are requested in multiple contexts more often compared to others. Users may see different contexts while they are requested for the same permission. Note that when a permission is given in one context, the app can use it in the other contexts without showing a prompt to the user.

Permission Usage. Permission usage in multiple contexts is more prevalent compared to permission requests. 46% of apps use at least one permission in more than one context. Location (57%) and Phone (26%) resources are used the most in multiple contexts. Comparing the number of times these two resources were requested in multiple contexts reveals that these resources are very often used in contexts where users may not see a prompt. For example, we found only 580 instances where Location access is requested in multiple contexts, while 3327 instances are found where Location is used in multiple contexts. In contrast, Storage is not as frequently used in multiple contexts (no. request: 892 vs. use: 53). Note that, this difference may be partly attributed to the relatively small number of permission mappings for Storage used in our analysis. Permissions for Location, Phone and Contacts are also often used the in background contexts, suggesting these permissions are used regardless of whether the app is being actively used or not; see Figure 2.

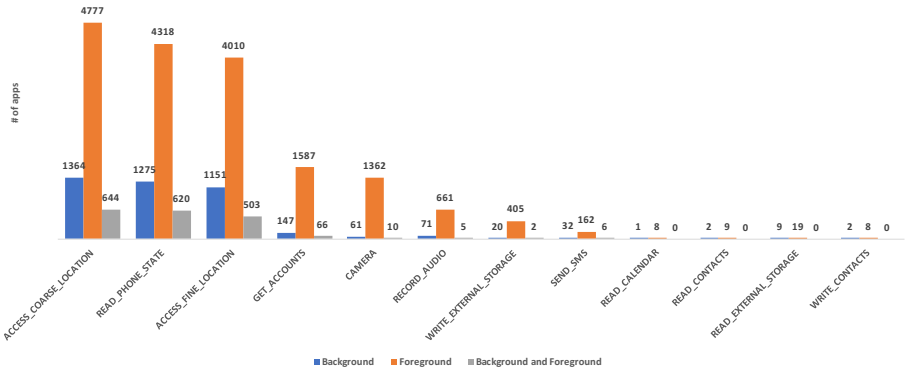

Fig. 2. The number of apps using permissions in background/foreground components, compared to the number of apps that use permissions in both types of components.

Contextual Differences. We investigate the prevalence of mismatching context between permission request and use. To make a fair comparison, we only include permissions for which we could extract both, the request context and the usage context in an app. We could identify 2722 permissions from 2012 apps with both request and usage contexts.

We find 1420 permissions being used in dissimilar contexts (i.e., the context of request for these permissions do not match the context of their use). More precisely, either these permissions are requested in contexts where they are not used at all, or the permissions are used in a context where the user may not see a request prompt. The average number of dissimilar contexts for these permissions is 2.25. In other words, if the permissions are granted in their request context, they may be used in two or more contexts without users' knowledge. Furthermore, we find 525 instances where dissimilar contexts include asking for a permission in foreground and using it in background. We acknowledge that such dissimilarity in context may be legitimate depending on the type of app. We do not attempt to justify the use of permissions in dissimilar contexts and leave it as future work.

We identify seven resources that are used in dissimilar contexts. Location, Storage, Phone and Contacts are used in varying contexts more often than the rest. While not as high, Camera and Microphone are also used in dissimilar usage contexts where the user might be unaware. Interestingly, we find seven apps that request Camera permission in foreground and uses it in a background `Service`; and two apps use Microphone in a similar fashion. Figure 3 shows the number of times various resources are accessed in dissimilar contexts.

Case Study. We take a messaging app named TextTray as an example. As a messaging app it is obvious for the app to request for the SMS resource. We found one foreground context (`Activity` component) where the permission prompt is

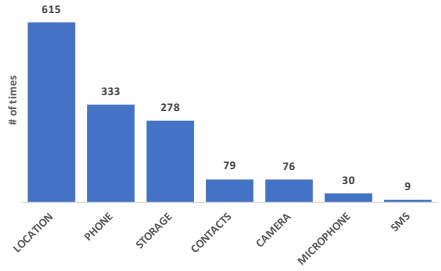

Fig. 3. The number of times different resources are accessed in dissimilar contexts.

shown. Assuming the user will grant the permission based on the context, we identify a valid use of that permission in the same context. However, we also notice a background context (in a `Service`) where messages are sent. While sending messages is the core functionality of any messaging app, such difference in the contexts can indeed be unexpected. Google later removed this app from Google Play.

5 Related Work

Several studies analyzed contextual resource usage in the install-time permission model. Yang *et al.* [15] define context based on environmental attributes (e.g., time of the day), to differentiate between malware and benign apps. In contrast, we define context differently, and target only regular apps. Wijesekera *et al.* [13] perform a user study to identify contextual differences and user reactions during permission use. They identify visibility to be an important context factor that validates resource access, and found user dissatisfaction while the context of permission usage changes subsequently. Both these studies analyze apps developed for the install-time permission model. Therefore, it is difficult to understand what context the user might see to make a decision on the permission in the runtime model. We analyze apps that are developed for the runtime model that enables us to identify the real contexts of a request prompt.

Another study by Wijesekera *et al.* [14] combines user privacy preference and surrounding contextual cues to predict user decisions. The key idea is to differentiate between the contexts of permission use and based on prior decisions made by the user, automatically grant or request users for a permission. While identifying the contextual differences in permission usage closely relates to this work, on both occasions [13, 14], the analysis was performed on a modified version of Android protected by the old permission model with apps not designed for the runtime model. In comparison to these studies that perform dynamic analysis to extract contextual information, we use static analysis to evaluate apps that are specifically designed and adapted for the runtime permission model.

Micinski *et al.* [10] tie user interactions to resource access in the runtime model. They develop a dynamic analysis tool named AppTracer to analyze the extent to which user interactions and resource accesses are related. A corresponding user study reveals that users generally expect resource access right after interaction with a related app functionality. In contrast, we focus on the different Android components in which the permissions are requested and used along with user interactions. Similar to AppTracer, Chen *et al.* [7] propose the Permission Event Graph (PEG) model, representing the relation between resource access and event handlers. They combine static and dynamic approaches to analyze regular and malicious apps. However, their analysis is also based on the old permission model, and cannot differentiate between the context of permission request and usage.

Another line of work use the permissions listed in the manifest to generate risk signals and rank apps based on permission usage. Wang *et al.* [12] use permission request patterns to identify potentially malicious apps. Taylor *et al.* [11] develop a contextual ranking framework based on listed permissions. They propose relative ranking of apps by identifying whether an app of a specific category requests for permission(s) that are not required by other apps in the same category. However, they do not consider how the listed permissions are used by the apps. Merlo *et al.* [9] propose a risk scoring framework based on permission utilization by apps. In comparison, we identify different contexts where the permissions are actually used and compare them with contexts where users see a request prompt.

6 Conclusions and Future Work

We present the first large-scale (using 6,790 regular apps) study on the contextual differences in Android apps in the runtime permission model. Our ContextDroid static analysis tool extracts the contexts in which apps request and use dangerous permissions. Our findings suggest a significant gap between the number of times the users see a request prompt indicating the use of a permission versus the number of times it is actually used. Difference in contexts of permission request and use implies the prevalence of permission use without users knowledge, even in the runtime permission model.

Both the ContextDroid tool and our experiments can be extended in future work. ContextDroid currently cannot identify sensitive and request API calls inside methods that are not included in the call graph (e.g., due to advanced forms of obfuscation beyond ProGuard [1] and similar tools). We identify sensitive API calls based on permission mappings from prior work [5, 6]. If an API is missing in the mapping list, ContextDroid will fail to identify the usage of the associated permission. Although we identify unexpected or dissimilar contexts, some of these occurrences might indeed be legitimate/benign depending on the functionality of the app. However, usage in contexts where users may not be aware should still be a matter of concern and needs a closer look. In this work, we do not attempt to classify whether such dissimilarity in contexts imply malicious intent by the app; we also leave this as future work.

Acknowledgements

We are grateful to anonymous reviewers for their comments and suggestions. The second author is supported in part by an NSERC Discovery Grant.

References

1. Shrink your code and resources (2017), https://developer.android.com/studio/build/shrink-code.html
2. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: AndroZoo: Collecting millions of Android apps for the research community. In: Conference on Mining Software Repositories. ACM (2016)
3. Android: App permissions best practices (2018), https://developer.android.com/training/permissions/usage-notes
4. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. ACM Sigplan Notices (2014)
5. Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: PScout: analyzing the Android permission specification. In: CCS. ACM (2012)
6. Backes, M., Bugiel, S., Derr, E., McDaniel, P.D., Octeau, D., Weisgerber, S.: On demystifying the Android application framework: Re-visiting Android permission specification analysis. In: USENIX Security (2016)
7. Chen, K.Z., Johnson, N.M., D'Silva, V., Dai, S., MacNamara, K., Magrino, T.R., Wu, E.X., Rinard, M., Song, D.X.: Contextual policy enforcement in Android applications with permission event graphs. In: NDSS (2013)
8. Felt, A.P., Egelman, S., Wagner, D.: I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns. In: SPSM. ACM (2012)
9. Merlo, A., Georgiu, G.C.: RiskInDroid: Machine learning-based risk analysis on Android. In: IFIP SEC. Springer (2017)
10. Micinski, K., Votipka, D., Stevens, R., Kofinas, N., Mazurek, M.L., Foster, J.S.: User interactions and permission use on Android. In: CHI. ACM (2017)
11. Taylor, V.F., Martinovic, I.: SecuRank: Starving permission-hungry apps using contextual permission analysis. In: SPSM. ACM (2016)
12. Wang, Y., Zheng, J., Sun, C., Mukkamala, S.: Quantitative security risk assessment of Android permissions and applications. In: Conference on Data and Applications Security and Privacy. Springer (2013)
13. Wijesekera, P., Baokar, A., Hosseini, A., Egelman, S., Wagner, D., Beznosov, K.: Android permissions remystified: A field study on contextual integrity. In: USENIX Security (2015)
14. Wijesekera, P., Baokar, A., Tsai, L., Reardon, J., Egelman, S., Wagner, D., Beznosov, K.: The feasibility of dynamically granted permissions: Aligning mobile privacy with user preferences. In: Security & Privacy Symposium. IEEE (2017)
15. Yang, W., Xiao, X., Andow, B., Li, S., Xie, T., Enck, W.: AppContext: Differentiating malicious and benign mobile app behaviors using context. In: ICSE. IEEE (2015)