RPCDroid: Runtime Identification of Permission Usage Contexts in Android Applications

Michele Guerra[a], Roberto Milanese, Rocco Oliveto[b] and Fausto Fasano[c]
University of Molise, Italy
{*michele.guerra, roberto.milanese, rocco.oliveto, fausto.fasano*}*@unimol.it*

Keywords: Security and Privacy, Application Security, App Permission, Android, Dynamic Analysis, Android Permission Model.

Abstract: Over the years, there has been an explosion in the app market offering users a wide range of functionalities especially since modern devices are equipped with many hardware resources such as cameras, GPS, and so on. Unfortunately, this is sometimes associated to indiscriminate access to sensitive data. This exposes users to security and privacy risks because, although resource usage requires explicit user authorization, once permission is granted, a mobile application is usually free to access the corresponding resource until the permission is expressly revoked or the app is uninstalled. In this work, we introduce RPCDroid, a dynamic analysis tool for run-time tracking of the behavior (UI events and used permissions) of Android mobile applications that use device resources requiring dangerous permissions. We assessed the effectiveness of the tool to identify usage contexts, discriminating between different kinds of access to the same sensitive resource. We executed RPCDroid on a set of popular applications obtaining evidence that, in many cases, mobile applications access to the same resource though different user interactions.

1 INTRODUCTION

In recent years, the mobile device market has grown enormously: as of today, an estimated 4.7 billion people worldwide own at least one smartphone, and the number of active users is set to increase further, exceeding 5.5 billion by 2025[1]. In this scenario, Android is confirmed as the most popular operating system, with a market share exceeding 70%[2] and nearly three million applications on the Play Store ™, which collectively count tens of billions of downloads each year (Scoccia et al., 2021).

Mobile devices are increasingly becoming sophisticated terminals enriched with numerous integrated peripherals and sensors, with the aim of supporting increasingly innovative features for both users and developers. As a side effect, it is common that recent applications gain access to sensitive and confidential data not only to realize the features made available to the user but also to study the user's behavior, preferences, and settings (Verderame et al., 2020). Android's permission management mechanism plays a crucial role in safeguarding user privacy. A recent empirical study by Scoccia et al. (Scoccia et al., 2021) showed that users concerned about their data privacy tend to evaluate the entire application negatively in case of over-permissions, thus confirming how proper permissions management is essential to a product success. Currently, control of access to sensitive device resources is managed by the individual Application Framework's sensitive APIs, which mediate and regulate each application requests according to the preferences specified by the user.

In the last decade, there have been several improvements to the authorization system, primarily aimed at increasing security and control on understanding how and when applications access personal data. However, the user awareness about when and how the permissions are exploited by each application is still lacking ans so is the flexibility of the current approach, contributing to bad practices like users grating permissions ignoring the possible impact on their privacy (Scoccia et al., 2021). One of the main weaknesses of the permission manager currently in use in Android is that approving a permission request usually enables the entire application to in-

[a] https://orcid.org/0000-0001-5507-9084
[b] https://orcid.org/0000-0002-7995-8582
[c] https://orcid.org/0000-0003-3736-6383
[1] https://www.statista.com/forecasts/1143723/
[2] https://www.statista.com/statistics/272698/

herit that permission globally. That means that, once an app is granted the access to the camera, it is allowed to access it without needing additional further approval in the near future and, potentially, even in background (Wijesekera et al., 2018).

In this work, we present RPCDroid a tool that helps in the identification of the uses of permission constrained resources. RPCDroid is a dynamic analysis tool installed as an Xposed module either on an Android real device with root permissions (so called rooted device) or on an emulator. We developed a dynamic logging system that directly monitors user actions and associates them to the specific functionality requested. The aim is to investigate if the above mentioned scenario is potentially exploited by real applications. Once the monitored application starts, RPCDroid can track all actions performed by the user at the interface level (regardless of whether they exploit one or more permissions) and the system calls to the access control and validation mechanism. In addition, for each of these events, the tool also captures a screenshot, further facilitating the identification and reconstruction of the different usage contexts. Finally, at the end of the app execution, the tool generates two detailed logs, one containing all the user-generated events and the other with all the permission accessed by the app, which allow us to identify which actions led to access to critical resources.

During a preliminary evaluation of the tool, we identified several unexpected accesses to sensitive resources, potentially endangering users' privacy. In addition to the manual execution of the application, the tool can automate the entire approach by having the device perform a series of pseudo-random operations on the UI (but still confined to the package of interest).

We assessed the effectiveness of the tool with 10 widespread and popular applications with the aim to investigate if real mobile applications access to the same resource in different contexts. This is worth to be studied because in this case the end user does not have the possibility to choose when the app is allowed to access the resource and when the access should be denied.

2 BACKGROUND

2.1 Android Permission Model

In Android, access to sensitive user data (such as contacts in the address book, received messages, or calendar appointments) or critical system resources (such as the camera, microphone, or biometric sensors) by applications is regulated through the mechanism of permissions(Wang et al., 2021).

Each permission is identified using a unique string (e.g., `android.permission.READ_SMS`), and it allows the use of certain *sensitive* APIs of the Application Framework (in this case, all those related to text messages) to applications that declare it with a `<uses-permission>` element in the *Manifest*[3].

Permissions are then categorized into three different risk levels[4]:

- *Normal*, assigned to all those permissions that control access to external data and resources but only marginally impact the user's privacy and the operation of other applications (e.g., `android.permission.SET_WALLPAPER` allows the user to replace the launcher wallpaper).
- *Dangerous*, assigned to all those permissions that control access to confidential data (e.g., `android.permission.READ_CALENDAR` allows to obtain calendar-related data), or `android.permission.PHONE_CALL` allows to make phone calls without the need for user confirmation).
- *Signature*, assigned to all those permissions that control access to particularly critical features (e.g., `android.permission.WRITE_SETTINGS` allows you to change system settings), and which for this reason cannot be used by third-party applications[5].

During the installation phase of an application, the *normal* and *signature* level permissions are granted automatically by the system(Li et al., 2021), without requiring explicit approval (the user can access the Play Store to consult the list of required permissions). In contrast, *dangerous* level permissions must be explicitly granted (or denied) by the user(Li et al., 2021). Depending on the version of Android, this request occurs:

- At *install-time* (up to Android 5.1.1), and the user can only choose whether to grant all permissions "in bulk" or not, without the possibility to revoke them later or to express different choices for each permission.

[3]The Manifest file describes the essential information of an application, such as the name of the package, its components, and the required permissions and features

[4]https://developer.android.com/guide/topics/permissions/overview

[5]This applies only to system permissions (i.e., those defined by Android itself). It is actually sufficient for the application requesting to be signed with the same certificate as the application that defined it within its Manifest with an element `<permission>`.

- At *runtime* (starting from Android 6.0), the system verifies the permissions to access the requested resource or feature, and an explicit consent is required if it has not been previously granted.

Thus, in recent versions of Android, the user has the opportunity to reject an access request (in Android 11, the approval can even be limited to a single usage) and to revoke previously approved permissions from the system settings.

However, even in the *runtime* model, permissions are granted to the entire application rather than to individual features or specific usage contexts. This behavior limits the control over how, when, and why data access is made (Scoccia et al., 2021).

One step forward is the introduction in Android 12 of *privacy indicators*, which indicates whenever an application is using the camera or the microphone.

2.2 Static and Dynamic Analysis Approaches to Tracking Permission Usage

Static analysis consists of the estimation and evaluation of certain properties without actually executing its code(Autili et al., 2021), but merely considering its structure or an abstract representation of it (e.g., a *control-flow graph*).

In contrast, the dynamic analysis techniques (including *testing* and *debugging*) are based on the observation of the behavior of the program, usually appropriately instrumented, at runtime(Ball, 1999), thus during its execution.

Static approaches allow more *path* of execution to be considered (and thus to achieve almost complete levels of code coverage)(Gomes et al., 2009). On the other hand, they cannot always capture the link between a program's inputs (i.e., context) and its behavior,(Ball, 1999) which adds to the inability to evaluate any information generated directly during execution,(Liu et al., 2019) or even to analyze obfuscated code.

In the literature, several tools developed to make the permissions mechanism in Android more flexible rely on static analysis techniques. However, these approaches need preliminary steps to work properly, such as APK decompilation and subsequent repackaging (which, of course, is done using a different *signature* than the original developer's, implying a whole series of important *drawback*, like the inability to directly update the application from official sources). Many of these static tools are based on *M-Perm*, an analysis tool proposed by Chester et al.(Chester et al., 2017) that works on the decompiled package. They searches, within the Java source code reconstructed through *reverse engineering*, for all instructions that contain a string traceable to a permission (e.g., `android.permission.READ_EXTERNAL_STORAGE`).

In addition to limitations related to the possible obfuscation of the code, these methods could be unable to detect any *stealth* uses of API sensitive (i.e., occurring without first verifying that the application has the necessary permissions but directly handling any *SecurityException* with a simple *try-catch* block). Even if permission is not granted, the application will continue to run normally without crashing and without the user becoming aware of the attempt to access personal data.

Scoccia et al.(Scoccia et al., 2021) have proposed *Android Flexible Permissions*, a finer-grained permissions management mechanism that allows the user to grant permissions to individual*features* (rather than to the entire application). The drawback of this approach is that the developer himself who has to define a *mapping* between application parts and features (and permissions). Moreover, they assume that *developers* explicitly adopt the framework and that their *feature-permission* associations will reflect what is actually implemented. This excludes the case of deliberately malicious behavior. In the area of dynamic analysis techniques, Wijesekera et al.(Wijesekera et al., 2015) conducted a field study to assess the frequency with which applications access sensitive data and resources without the user's knowledge. Differently from us, the authors investigate on repeated access to the same resource, while we focus on the contexts in which the access happened. Liu et al. (Liu et al., 2019) implemented a hybrid methodology (partially based on static techniques), to consider the problem of *data leakage* in the use of third-party libraries (mainly of *analytics* and *advertising*). Heid and Heider propose a generic design for an automated dynamic app analysis environment and highlights the required components as well as functionality to reveal security and privacy issues(Heid and Heider, 2021). None of the previously mentioned approached investigate the role of the usage context to discriminate between several accesses to the same resource.

3 RPCDroid DESIGN AND IMPLEMENTATION

In this paper we want to assess the effectiveness of adopting the usage context as a mean to discriminate between different kind of accesses to the same sensitive resource To this aims, we developed RPCDroid, a tool to monitor the execution of a mobile applications using specific device resources that need dangerous permissions to be accessed and support the process of linking resource usage and the access context. We define the usage context as a combination of the feature the app is executing and the corresponding user action on the UI. RPCdroid Analyzer is a dynamic analysis tool installed as an Xposed module on the Android device / emulator to collect helpful information in identifying a resource usage contexts. Once the monitored application is started, RPCDroid Analyzer starts tracking any access to resources that require dangerous permission at system level (e.g., the camera, microphone, storage, or location) through a dynamic injection mechanism based on hooking and callback techniques. These include all the actions performed by the user at the interface level as well as any system call to the Android access control and validation mechanism. Our approach allows the semi-automatic identification and *a posteriori* analysis of the usage contexts. To execute a completely automatic analysis, without the user interaction, the tool can be cofigured to execute random UI interactions on the apps and observe the permission requests. It both cases, RPCDroid automatically associates the observed permission requests and UI actions to categorize each resource uses within its own context.

In the next sections, after a brief overview of the leading technologies used, the system's overall architecture will be presented together with details on its components.

3.1 Used Technologies

3.1.1 Xposed Framework

Xposed is a widely used framework for *rooted* Android devices that through a rich set of APIs allows to register any Java method of any running application (through the so-called *hooking* process), and then inject code into it, monitor its use, prevent its invocation, or even replace it altogether. All new instructions are executed in the original process and in a completely transparent way, just as if no changes occurred (Osman et al., 2020). All processes in Android (including the system processes) are generated from a single parent, the *Zygote*, which after being created at device startup, loads all the libraries and resources required for the execution, e.g., the Application Framework code (Lee et al., 2014). The *Zygote* code is therefore inherited and shared by every other child process, in a memory-saving manner thanks to the *copy-on-write* strategy (Google, 2022a).Xposed leverages this mechanism by modifying the code executed by the *Zygote* at its startup file (which justifies the need for root permissions) and importing an additional package, `XposedBridge.jar`, which via the `handleHookedMethod` method invokes the *callback* `beforeHookedMethod` and `afterHookedMethod` eventually defined by the developer, thus making it possible to inject code before and/or after the execution of the original method, respectively. The methods to be tracked and the related *callbacks* are defined within Xposed modules, which are then installed on the device as if they were regular applications(Liu et al., 2019).

3.1.2 Monkey Exerciser

Monkey[6] is a command-line tool, included in the Android SDK, that allows generating pseudo-random sequences of user events (such as touches, gestures, and keystrokes) within an emulator or a device connected in *debug* mode, thus simulating a typical application usage session.

In addition to providing a high *code coverage* in relatively little time (Hasan et al., 2020), Monkey is extremely versatile. In fact, it is sufficient to run the command `adb shell monkey` followed by the number of events to be played, without the need for further configuration) and highly adjustable (for example, it is possible to customize the percentage for each type of event, or to restrict the session to specific packages (Google, 2022b)

3.2 System Architecture

To facilitate the system development process and, subsequently, to ensure a high degree of maintainability of the code, RPCDroid was designed according to the principle of functional decomposition. Each subsystem is responsible for implementing a specific functionality and making services available to the other parts. The *hooks*, *callbacks*, and *broadcast packages* are responsible for the method hooking procedure and subsequent code injection. At the same time, *config* and *logs* handle all aspects of configuration files (such as loading, parsing, and parameter extraction) and log files (providing both structures and read and write functionality). The coordination center of all these different components is represented by the *AnalysisModule* class, an Xposed module responsible for linking and orchestrating them appropriately. Specifically:

- As soon as the package related to the Android System Server is loaded into memory, the module requests to load (from the configuration file) all

[6]https://developer.android.com/studio/test/monkey

the parameters for the analysis. The parameters are then set, after which it registers a hook that allows monitoring of all requests for permissions.

- Each time an application is launched, the module checks, through a flag, whether its execution should be tracked or not. If so, it takes care of defining all the hooks necessary for the tracking of user actions to occur correctly. In addition, it also registers within the target application a Broadcast Receiver deputed to the generation of logs related to permissions. It is worth noting that the logger runs in a separate thread, to avoid slowing down or, even worse, blocking the UI update.

With *callbacks*, *hooks* classes constitute the main components of RPCDroid, since, when used in combination, they make possible runtime instrumentation of the Application Framework code and all applications (including third-party applications). Specifically:

- Callbacks contain the code to be injected.
- Hooks define a correspondence between the methods of the original code and the particular callback that is to be called (and thus when it is to be executed).

All implemented callbacks extend the abstract class `XC_MethodHook`, provided by the Xposed API, within which the signatures of two methods are defined:

- `beforeHookedMethod`, which is called by the Xposed framework immediately before the target method it has been associated to. This also allows for preventing its execution with the `param.setResult()` statement, ensuring that the object passed as a parameter is compatible with the method's return type.
- `afterHookedMethod`, which is called by the Xposed framework after the target method it has been associated to. This also allows its return value to be obtained with the `param.getResult()` statement).

Both methods accept a parameter of type `XC_MethodHook.MethodHookParam` and allow not only to obtain helpful information about the target method itself (e.g., the instance on which it was invoked or the values the parameter) but also to influence its execution or to modify its return value.

3.2.1 Callbacks for Events Monitoring

To specify behaviors for a particular group of Activity or View, respectively, filtered according to their class, we defined two abstract

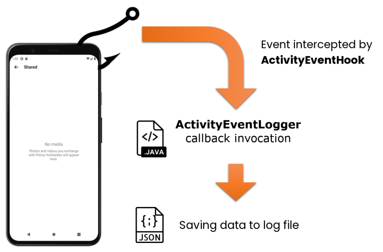

Figure 1: Process of *logging* an event related to an Activity.

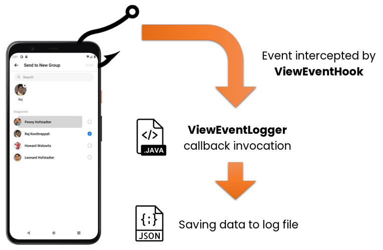

Figure 2: Process of *logging* an event related to a View.

classes, `ActivitySpecificCallback` and `ViewSpecificCallback`. This is extremely practical to limit the *logging* of events to certain types of user interfaces, for example, to *checkbox* only.

It is important to note that there is always the possibility to specify an empty filter to obtain a generic behaviors. This is precisely what has been done within the analysis module with the `ActivityEventLogger` and `ViewEventLogger` implementations.

The process of logging an activity (Figure 1) or a view (Figure 2) are reported in figures. These two *callbacks* are associated with the corresponding *hooks*. During the testing process, they allow the generation of two separate logs, one containing all application-related events (such as the start of a new Activity or a *tap* on a View) and the other containing all the permissions requested and then actually used.

Both classes execute the *override* of `beforeHookedMethod`, where, after obtaining a reference to the instance on which the *hooked* method was called, they verify that the filter condition is met (if set), then generate a new log item and add it to the queue of the respective JSON file, as well as save a *screenshot* associated with the event.

3.2.2 Callbacks for Permissions Monitoring

A third abstract class, `PermissionSpecificCallback`, allows specific behaviors to be associated with `(package, set-permissions)` pairs, with the possibility of also specifying global filters (i.e., valid for all running applications), denoting the constant `ANY_APP_PACKAGE`,

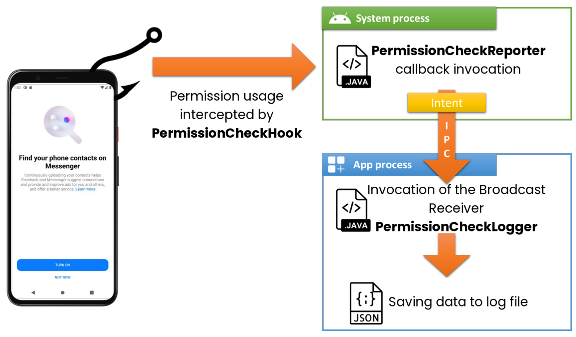
Permission usage intercepted by PermissionCheckHook
System process
PermissionCheckReporter callback invocation
JAVA
Intent
IPC
App process
Invocation of the Broadcast Receiver PermissionCheckLogger
JAVA
Saving data to log file
JSON
Find your phone contacts on Messenger

Figure 3: Logging process of permission usage.

or empty filters (which apply instead to all permissions declared by a single application, including non-system permissions).

Finally, the decision-making process for determining whether to execute the *callback* code or not occurs in the following order:

- If an application-specific *entry* is present, it is verified whether the specific permission being used matches the filter (if set). If not, the *code injection* does not occur.
- If a *entry* is present. *ANY_APP_PACKAGE*, it is checked that the specific permission being used is present within the global filter (or that an empty one has been specified): if not, the code will not be injected in this case either.
- In all other cases, the *callback* method is never called, and the execution of the *target* method continues normally.

In contrast to event tracking, the code for the `PermissionLoggerReporter` class is associated with the *hook* `PermissionCheckHook` and is therefore executed in the system process (instead of the analyzed application process, as shown in Figure 3).

For this reason, the entire *logging* process is delegated to a particular Broadcast Receiver[7], `PermissionCheckLogger`, which is automatically injected and registered by RPCDroid when a monitored application is launched, and the *callback* merely send it an Intent[8] containing all the information about the intercepted request (such as the permission name and the *timestamp*).

[7]A Broadcast Receiver is an Android component that allows applications to perform operations in response to specific events notified using *broadcast* messages.

[8]In Android the Intent mechanism enables both the communication between distinct components of the same application (*intra-process communication*) and between different applications (*inter-process communication*).

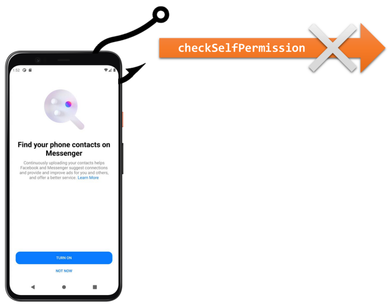

Figure 4: Checks made through `checkSelfPermission` are ignored to reduce false positives.

3.2.3 Others Callbacks

A few more words should be spent on `PermissionSelfCheckSuppressor`, which implements the *logging* of only the permissions actually used, and whose *check* takes place at the System Server level.

Applications have the opportunity to check, through a call to `selfCheckPermission`, whether given permission has been granted or not (and again, this results in an invocation to the `checkComponentPermission` method of the `ActivityManagerService`).

However, since the explicit checking of permission is not necessarily followed by its actual use, to avoid the risk of false positives, it was preferred, in contrast to other approaches(Wijesekera et al., 2015), to exclude from the log these requests (see Figure 4).

3.2.4 Outputs

The output of the monitoring process for each app includes:

- One or more log files[9], in `JSON` format, containing a record of all events handled (such as an Activity change or interaction with a UI component).
- One or more log files (also in `JSON` format) containing the permissions used.
- *Screenshots*, in `png` format, for each event or permission request (we use the `elementId`, in the former case and the `timestamp`, in the latter).
- A *screen recording* (saved in the file `capture.mp4`), which is processed by the script `analyze.rb` at the end of the session.

All the output files are stored in the device storage, appropriately organized into subfolders.

[9]Specifically, a new log file will be created each time the examined application is started (and thus for each different session of use).

4 PRELIMINARY EVALUATION

We conducted a preliminary study on the effectiveness of our approach to identify different usage context concerning the use of sensitive device features, i.e., those feature that need an explicit permission grant by the user before being accessed.

The study involved 10 apps selected from the top downloaded app within the main Play Store. We excluded those app that did not declare the use of at least one dangerous permission. We executed the apps on an emulator with Android 11 (API level 30), and, for each app, we instrumented Monkey to generate 800 interactions that required around 6 minutes per session. The tool's results have been assessed by analyzing the log, the screenshots, and the screen record[10].

In Table 1 we report the results of the study. In the first column, the total number of UI events caught by RPCDroid causing an access to the sensitive resource is reported. Note that not every event is associated with a different context. This is due to the fact that the same event on the UI can be associated with the same feature and exploit the same permissions. The number of contexts represents the distinct contexts identified by the tool. Contexts are considered distinct if the triggering element is different (e.g., a different UI component) or if the feature accesses a different set of resources compared to the previous executions. This choice was a compromise between the need to identify potentially different uses of the same resource and the will to reduce the total number of context. In particular, we do not want to have any access to a resource be considered a new context as this would reduce out approach to an "ask each time" rule, but we want to understand if the same resource is accessed in the background, after a random timeout, when the activity is started, when a specific button is pressed, and so on. In the last four columns we report the number accesses to 4 specific resources, namely camera, GPS, microphone and external storage. For each of them we report the number of times the access is referred to a distinct context and the total number of accesses. Note that in some cases more than one resource can be accessed within the same context. For example, when a screenshot is taken and saved to the storage both the camera and the storage may be included in the same context or during a video call both camera and microphone are usually accessed. On the other hand, in Table 1 we focus only on a subset of the resources accessed during the study, this is the reason why the number of context in some cases is higher than the sum of all the resource access contexts.

[10]A sample output is available here: https://github.com/MicheleGuerra/RPCdroid-output

By analyzing the result of the usage contexts we note that the tested apps use the same resource at least in two different contexts, with a few exceptions. For instance, StickerMaker accessed the camera 10 times, but the tool identified the same context every time the app requested the related permission. This is what we expected to be the common behaviour. Instead, we noticed that most of the app involved in the study accessed the same resource within several distinct contexts. The resource that is mostly context independent is the Storage. This is understandable, because if many different features need to save feature-related data, each of them is identified as a separate context. Precise location is another resource that is mostly globally accessed within an application (when used). Concerning the Camera, we noted an unclear usage of such a privacy sensitive resource.

Note that, since we used a program that generates pseudo-random streams of user events (clicks, touches, or gestures) we cannot be sure that any different context has been explored during our study. This means that the number of contexts reported in Table 1 are only a subset of all the possible access contexts. On the other hand, the use of pseudo-random events does not invalidated our results for all the apps showing multiple contexts per resource, because all the context evaluated are legitimate and exploitable. Also, there is a possibility that asynchronous/background calls have caused the access context to be misidentified. To avoid this, the analysis of the screenshots captured for each event together with the possibility to run multiple random session were essential to mitigate such a bias.

It is important to note that the mere access to the same resource in multiple contexts is not a proof of malicious behaviour (e.g., independent features may need to legitimately access the camera). However, from the user perspective, there is no way of discriminating among multiple accesses. We think the user should be aware of such a behaviour. In Android 12 and later, every time the camera is accessed, a led is activated on the screen to inform the user. Even if this is an improvement in the usage awareness, it is still limited to few resources (camera, microphone, and location). Moreover, the access to the resource is granted, potentially leading to privacy issues.

5 CONCLUSIONS

In this paper we presented RPCDroid, a tool to monitor the execution of an Android mobile application that accesses specific device resources that need dangerous permissions. Such type of resources need to

Table 1: Number of different UI events (UIE), identified context (CTX) and resource access (contextualized/total).

Package (*version*)	**UIE**	**CTX**	**Cam**	**Loc**	**Mic**	**Sto**
Amazon (*24.6.0.100*)	59	20	3/3			20/59
Ebay (*6.48.0.1*)	15	8	3/5	6/13		3/3
Facebook-katana (*353.0.0.0.4*)	36	27	2/2	19/25		15/17
Google Earth (*9.151.0.2*)	39	21		5/6		3/4
Instagram (*223.0.0.0.18*)	14	11		11/13		2/2
Shazam (*12.6.0*)	98	29		6/6	1/1	
Spotify (*2.0.45*)	61	21				4/4
Sticker Maker (*0.0.2-82*)	14	5	1/10			4/4
Translate (*6.14.0.05.35*)	44	22	9/22		4/4	12/20
VideoLan (*3.4.3*)	27	12				2/2

be explicitly authorized by the user. However, once a permission is granted the mobile application is allowed to access the related resource until the grant is explicitly removed or the app is uninstalled. Thus, there is the possibility that an app uses resources like the camera or the GPS within several different features, making difficult for the user to discriminate among the feature that are allowed to access the permission and those that should be blocked.

To assess the effectiveness of the tool to automatically discriminate between different kind of accesses to the same sensitive resource, we conducted a preliminary study on 10 popular app. The apps have bee automatically run by using the Monkey, a program that generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events (Google, 2022b). The results confirm that, in many cases, when the app accesses to a sensitive resource, this is done in more then one context. Even if the most recent versions of Android improved the awareness enforcing the privacy protection mechanism by means of special led indicators, nothing prevents the app from exploiting the acquired permission.

As future work, we plan to conduct a large scale analysis of the apps published in the Play Store ^{TM} to investigate how much widespread the problem is.

REFERENCES

Autili, M., Malavolta, I., Perucci, A., Scoccia, G. L., and Verdecchia, R. (2021). Software engineering techniques for statically analyzing mobile apps: research trends, characteristics, and potential for industrial adoption. *Journal of Internet Services and Applications*, 12(1):1–60.

Ball, T. (1999). The concept of dynamic analysis. In *Software Engineering—ESEC/FSE'99*, pages 216–234. Springer.

Chester, P., Jones, C., Mkaouer, M. W., and Krutz, D. E. (2017). M-perm: A lightweight detector for android permission gaps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 217–218. IEEE.

Gomes, I., Morgado, P., Gomes, T., and Moreira, R. (2009). An overview on the static code analysis approach in software development. *Faculdade de Engenharia da Universidade do Porto, Portugal*.

Google (2022a). Android developers guide: Overview of memory management. https://developer.android.com/topic/performance/memory-overview.

Google (2022b). Android studio guide: Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.

Hasan, H., Ladani, B. T., and Zamani, B. (2020). Enhancing Monkey to trigger malicious payloads in Android malware. In *2020 17th International ISC Conference on Information Security and Cryptology (ISCISC)*, pages 65–72.

Heid, K. and Heider, J. (2021). Automated, dynamic android app vulnerability and privacy leak analysis: Design considerations, required components and available tools. In *European Interdisciplinary Cybersecurity Conference*, EICC, page 1–6, New York, NY, USA. Association for Computing Machinery.

Lee, B., Lu, L., Wang, T., Kim, T., and Lee, W. (2014). From Zygote to Morula: Fortifying Weakened ASLR on Android. In *2014 IEEE Symposium on Security and Privacy*, pages 424–439.

Li, R., Diao, W., Li, Z., Du, J., and Guo, S. (2021). Android custom permissions demystified: From privilege escalation to design shortcomings. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 70–86.

Liu, X., Liu, J., Zhu, S., Wang, W., and Zhang, X. (2019). Privacy risk analysis and mitigation of analytics libraries in the Android ecosystem. *IEEE Transactions on Mobile Computing*, 19(5):1184–1199.

Osman, T., Mannan, M., Hengartner, U., and Youssef, A. (2020). Securing applications against side-channel attacks through resource access veto. *Digital Threats: Research and Practice*, 1(4):1–29.

Scoccia, G. L., Malavolta, I., Autili, M., Di Salle, A., and Inverardi, P. (2021). Enhancing trustability of android applications via user-centric flexible permissions. *IEEE Transactions on Software Engineering*, 47(10):2032–2051.

Verderame, L., Caputo, D., Romdhana, A., and Merlo, A. (2020). On the (un) reliability of privacy policies in android apps. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–9. IEEE.

Wang, Y., Wang, Y., Wang, S., Liu, Y., Xu, C., Cheung, S.-C., Yu, H., and Zhu, Z. (2021). Runtime permission issues in android apps: Taxonomy, practices, and ways forward. *arXiv preprint arXiv:2106.13012*.

Wijesekera, P., Baokar, A., Hosseini, A., Egelman, S., Wagner, D., and Beznosov, K. (2015). Android permissions remystified: A field study on contextual integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 499–514, USA. USENIX Association.

Wijesekera, P., Baokar, A., Tsai, L., Reardon, J., Egelman, S., Wagner, D., and Beznosov, K. (2018). Dynamically regulating mobile application permissions. *IEEE Security & Privacy*, 16(1):64–71.