Michele Guerra

Enhancing Mobile User Privacy through a Context Aware Permission Model

PhD Thesis

University of Molise, Italy

June, 2024

University of Molise

Department of Biosciences and Territory

Ph.D. Course in Biosciences and Territory
XXXVI Cycle
S.S.D. ING-INF/05

Ph.D. Thesis

Enhancing Mobile User Privacy through a Context Aware Permission Model

Ph.D. Coordinator
Prof. Filippo Santucci De Magistris

Tutor
Prof. Fausto Fasano
Prof. Rocco Oliveto

Ph.D. Candidate
Michele Guerra

Academic Year 2023/2024

Abstract

The pervasive use of mobile devices has fueled the growth of mobile applications, many of which demand access to sensitive user data and system resources. This raises significant privacy concerns, particularly as current permission models often lack flexibility and user awareness. This Thesis introduces SCoPe, a novel Android permission model designed to enhance privacy through a proactive, context-aware, and user-centered approach. SCoPe addresses the limitations of static permission systems by incorporating dynamic privacy notifications and context-sensitive permission controls. Using RPCDroid, our developed tool, we analyzed how Android apps misuse permissions, often accessing sensitive data unknowingly to users. Our study on Privacy Indicators (PI) in Android 12 revealed their limited efficacy, prompting further investigation into PI designs via eye-tracking technology to assess user attention and perception. SCoPe, leveraging these insights, tracks user interactions and correlates them with permission accesses, allowing permissions to be granted on a per-feature basis and minimizing user interruptions. Our evaluation shows SCoPe's high accuracy in context recognition and its ability to reduce "warning fatigue" by limiting authorizations to new contexts. Additionally, we incorporated AI through LLMs, specifically GPT-4, to analyze app interfaces and assess permission requests, which showed significant promise in enhancing mobile security and privacy.

Contents

1 Introduction **1**

I Background & Related Works **9**

2 Android Architecture **12**

2.1 Android permissions 14

3 Related Works **19**

3.1 Analysis of App Permission Usage 19

3.2 Studies on User Behavior Regarding Permissions 21

3.3 User-Centered Tools for Permission Management 23

3.4 Fine-Grained Permission Models 24

3.5 Dynamic approaches to contrasts Logic Bombs in Android Apps . 27

3.6 Eye Tracking in Mobile Computing Privacy 28

II Enhancing Android Security: A Dynamic Analysis with RPCDroid 31

4 RPCDroid: Runtime Identification of Permission Usage Contexts in Android Applications 34

4.1 Introduction 34
4.2 Design and Implementation 36
4.3 Preliminary Evaluation 44
4.4 Final Remarks 47

III Advancements in Android's Permission Model: A Deep Dive into the Role and Impact of Privacy Indicators 48

5 An Empirical Study on the Effectiveness of Privacy Indicators 51

5.1 Introduction 52
5.2 Empirical study Design RQ_1 55
5.3 Results RQ_1 65
5.4 Evaluating an alternative solution RQ_2 73
5.5 Threats to Validity 77
5.6 Final Remarks 80

6 Visual Attention and Privacy Indicators in Android: Insights from Eye Tracking 82

6.1 Introduction 83
6.2 Design of the Study 85
6.3 Technical Implementation of the Used System 93
6.4 Results 95
6.5 Final Remarks 99

IV Advancing Android Security through Comprehensive Analysis of Context-Aware Permission Model: From SCoPe Implementation to LLM-Based Security Assessments in Android 101

7 SCoPe: a fine-grained, context-aware permission model for Android 104

7.1 Introduction . . . 105
7.2 SCoPe: Tool Design and Technical Implementation . . . 110
7.3 From Manual to Automated: A Preliminary Study on Transitioning Context Detection from RPCDroid to SCoPe . . . 138
7.4 Smarter Testing for Android Apps: Moving Beyond Randomness with AI . . . 142
7.5 Evaluating the Efficacy of SCoPe Automated Context Identification Against Manual Analysis in Android Apps . . . 159

8 Beyond Detection: Using SCoPe to Defuse Logic Bombs in Android Environments 164

8.1 Introduction . . . 164
8.2 Experimental Design . . . 168
8.3 Evaluation . . . 170
8.4 Conclusion . . . 173

9 Assessing the Effectiveness of an LLM-based Permission Model for Android 175

9.1 Introduction . . . 175
9.2 Experimental Design . . . 177
9.3 Analysis results . . . 197
9.4 Assessing Effectiveness on Malicious Apps . . . 201
9.5 Final remarks . . . 208

10 Conclusion **209**
10.1 Lessons Learned 211

Appendices **215**

A Publications **215**
A.1 Other Publications 216
A.2 Software Artifacts 216

List of Figures

2.1 The Android software stack 13

4.1 Process of *logging* an event related to an Activity 40
4.2 Process of *logging* an event related to a View 41
4.3 Logging process of permission usage 42
4.4 Checks made through `checkSelfPermission` are ignored to reduce false positives 43

5.1 An example of Privacy Indicator in action on Android 12. 53
5.2 An example of prototypes created using Figma. 58
5.3 Experimental protocol. 61
5.4 Demographic users information. 65
5.5 An example of our enhanced privacy indicator. 74
5.6 Demographic users information in our solution 76

6.1 An example of Privacy Indicator in action on Android 12. 85
6.2 Comparative Display of Classic Privacy Indicators and Proposed Alternatives 88
6.3 Sequential Diagram of the Controlled Experiment Process 89

6.4 Calibration process screen displaying markers for eye-tracking accuracy improvement 91

6.5 Two of the experiment tasks requiring low spatio-temporal attention (*Quiz*) on the left and high spatio-temporal attention (*Game*) on the right 92

6.6 Average fixation durations on Privacy Indicators (*Classic*, *Edge*, *Disk*) during low (*Quiz*) and high (*Game*) spatio-temporal attention tasks, highlighting the differential impact of task type on user's attention to PIs. 97

7.1 Schematic of the application launch process from the `Zygote` . . . 112

7.2 The overall system architecture. The dashed components are injected with XPOSED into the processes of the monitored app or system 114

7.3 Package UML diagram with the packages in which SCOPE is organised 115

7.4 Class diagram with the main classes and interfaces of the `hook` package 120

7.5 Inter-process communication through the `Binder` infrastructure . . 122

7.6 Class diagram with the classes of the `service` package 124

7.7 Class diagram with the main classes and interfaces of the `context` package 128

7.8 Class diagram with the main classes of the `permission` package . . 134

7.9 Example of SCOPE permission request dialogue in a specific identified context (after touching *Tap to Shazam*). 137

7.10 Different microphone access contexts identified by SCOPE in Shazam 141

7.11 Location access contexts considered as different from RPCDROID, but merged by SCOPE 142

7.12 Example *"file.html"* generated at the end of the app analysis 146

7.13 Boxplot illustrating various coverage levels categorized by the corresponding methodology 150

7.14 Code snippet from the **Clean Timer** application analyzed in a manual session. The red section indicates the unexecuted *swipe* functionality. 156
7.15 Example of an Easter egg in the Android Operating System version "Red Velvet Cake," accessible by tapping the Android version in the settings eight times. 157

8.1 The approach *dynamically identifies* individual permission usage contexts, showing the user the *decision prompt*. 169

9.1 Iterative experimental process 180
9.2 Accept rules for download(a), read rules and accept for download (b) 206

List of Tables

4.1 Number of different UI events (UIE), identified context (CTX), and resource access (contextualized/total) 46

5.1 Tasks and values assigned to the PI, CB, and R variables for each group. 59
5.2 Difference of resource identification between latent and explicit tasks in presence or absence of PI. 68
5.3 Summary table with the number of participants who correctly identified the use of the resource (camera or microphone) reported in the single row and the respective behavior (latent or explicit), when PI were visible and not visible. 68
5.4 The number of participants that, during the *response acquisition phase*, did not identify the resource and instead, during the *matching phase*, changed their response correctly. 69
5.5 Logistic Regression Results 73
5.6 Summary of the experiment results 77

6.1 Average time spent looking at the PI in the different conditions. . . 96
6.2 Statistical results 97

7.1 List of apps against which the ability of SCoPe to extract use context has been evaluated . . . 139
7.2 Number of identified usage contexts (SCoPe / RPCDroid) for each app and resource type . . . 140
7.3 This table compares code coverage achieved by different algorithms for all 100 applications from the dataset. The value in *green* represents the highest code coverage achieved for each application. The equal coverage values are highlighted in *orange*. . . . 151

8.1 Summary of the analysis results for the 70 apps . . . 170

9.1 Summary of results . . . 197
9.2 Evaluation of the final results . . . 198
9.3 Permission Metric Analysis: II Improving Analysis result . . . 200
9.4 Evaluation Permission for commercial application: III Main Experiment . . . 200
9.5 Evaluation Permission for Open-Source application: III Main Experiment . . . 201
9.6 Summary of analyzed application: Final Experiment . . . 201
9.7 Malicious application analyzed . . . 208

CHAPTER 1

Introduction

The mobile device market has witnessed exponential growth over the past decade, with the global smartphone user base escalating from approximately 1.5 billion in 2013 to over 5 billion in recent years [1]. This surge has been accompanied by a corresponding expansion in the mobile application ecosystem, highlighted by the availability of 3.5 million applications on the Google Play Store [2]. Android, capturing three-quarters of the global smartphone market share, has become a focal point for developers due to its widespread adoption [3].

Advancements in mobile device capabilities, including computing power, storage, sensor technology, and connectivity, have enabled applications to offer a myriad of functionalities, from health monitoring to multimedia management [4]. These applications frequently require access to personal data and critical system resources, such as geolocation and microphone input, raising significant security and privacy concerns [5, 6]. Despite the prevalence of legitimate applications, there is an alarming rate of data collection without user consent, ostensibly to tailor user experience based on behavior and preferences [7, 5].

In response, Android developed a permission model that regulates access to sensitive APIs based on user preferences, requiring explicit user consent before accessing protected resources [8, 9]. However, enhancements in the permission system have not fully addressed the issues of flexibility and user awareness, leading to often uninformed permission grants with potential privacy risks [10, 11].

A robust permission-based security model should balance effectiveness and efficiency, ensuring that users receive timely notifications and have the autonomy to make informed choices without being overwhelmed by frequent interruptions [12]. Originally, Android's permission model primarily requested all permissions at *installation time*. This model forced users to either grant permissions en masse or abort the installation, frequently leading to uninformed compliance to facilitate app usage [13, 14].

This initial approach was inadequate as it required users to make decisions prematurely, often without a contextual understanding of the permissions being requested [15]. In recognition of these limitations, Android 6 Marshmallow, released in October 2015, introduced a significant revision of the permission model, adopting an *ask-on-first-use* model that prompts users for permission at runtime, thereby providing a context-specific basis for informed decision-making [12, 16].

Despite these improvements, the model's rigidity in granting permissions per-app remains a critical vulnerability. While it is rational for a messaging app to access the microphone during voice message composition, this permission could potentially be misused across different contexts or by third-party libraries incorporated within the app [17, 10].

This motivated a *preliminary study* conducted to understand the misuse of permissions in different contexts by commercial Android apps, which often leads to unclear behaviors towards users. We developed RPCDroid, a tool that identifies when and how apps access sensitive resources, offering insights into the often opaque permission usage in apps. This tool tracks UI events and permissions, generating detailed logs for analysis. The aim was to reveal how much apps access resources in varying contexts, an aspect critical for user privacy and app transparency. We analyzed 10 popular apps using Monkey, an input generation tool that performs random actions on the user interface, with the aim of attributing permissions to the UI event automatically. The manual review of logs showed

that most apps used permissions in several contexts and accessed sensitive data without user awareness, compromising users privacy.

Starting from Android 12, as a response to the need for improved user privacy, new improvements have been introduced to the permission model regarding the use of resources and user awareness. In particular, Google introduced Privacy Indicators (PI), highlighting their commitment to user security. In particular, PI are notification indicators aimed to inform the user, through a colored dot on top of the screen, about using a device resource, such as the camera or the microphone. However, the effectiveness of this approach has not been assessed yet. In a *second contribution* [18], we show the results of an experiment on the effectiveness of using the PI to inform the user whenever an app accesses the mobile device camera or microphone. We conducted a controlled experiment involving 122 participants to measure the visibility and impact of PIs during resource usage in latent and explicit usage scenarios. In latent contexts, apps accessed resources unnecessarily, simulating potentially malicious behavior, whereas explicit contexts required such access for real app functionality. Participants were engaged in predetermined tasks with controlled variations in personal information visibility, both in latent and explicit contexts. Upon analyzing the collected data, we found that the PI did not significantly affect user awareness of resource usage. These findings highlighted the limited efficacy of the existing PI system. Building on this, we tested an enhanced PI featuring a more informative design with textual information. This new PI was subjected to a separate experiment with 38 participants, demonstrating a marked improvement in user recognition and awareness.

To delve deeper into the analysis of PI effectiveness, we conducted a follow-up study using eye-tracking, our *third contribution*. Unlike the previous one that primarily relied on user-reported detection of resource usage, this approach takes an unexplored direction by capturing real-time, visual engagement data to assess whether users notice the PI. This methodology enables a more detailed and granular analysis of user attention and perception. By utilizing eye-tracking technology, we are not only able to determine whether users are aware of resource usage but also to pinpoint the exact moments and contexts in which these indicators are observed or overlooked. Additionally, the study explores user responsiveness to two innovative PI designs (edge and disk) across two interactive

contexts. The executed tasks were intended to explore the effectiveness of PI notifications within apps that significantly differ in terms of user engagements: low spatio-temporal attention during quizzes and high spatio-temporal attention during interactive gaming scenarios. The data collected involving 29 participants indicate that the cognitive load required by a specific task notably impacts the likelihood of the user detecting a PI. This was particularly evident in the Game setting, which is characterized by a high degree of spatio-temporal focus, and where both Android's native PI, as well as alternative animations, were consistently overlooked. These findings suggest inattentional blindness, where users caught up in demanding tasks may fail to notice secondary stimuli, such as the PI. There is also evidence that each PI's characteristics directly impact its visibility, as our analysis revealed that the Disk was marginally more effective at capturing user attention than the other PI examined in this study. This suggests that specific visual features, such as variations in color or motion, may increase their detectability. The Classic PI, as implemented, fails to capture the user's attention consistently during tasks requiring high levels of concentration, so malicious apps could exploit this flaw to conceal privacy-damaging actions during periods of intense user engagement. These results suggest a clear path forward for the evolution of PI, emphasizing the need for designs that respect user attention and promote an active understanding of privacy risks. These improvements could lead to more trustworthy mobile environments, wherein users can confidently manage the privacy of their sensitive information.

Furthermore, while informative, the privacy indicators introduced in Android 12 are often too subtle to be noticed during regular device use, thus providing minimal practical oversight.

This *Thesis* proposes SCoPe *(Sensitive Context-aware Permission Enforcement Framework)*, a novel permission model for Android that incorporates a proactive, context-aware, and user-centered approach to privacy management. Building on the foundational analyses of Android's existing permission mechanisms, SCoPe seeks to overcome the limitations of static permission systems by integrating dynamic, real-time privacy notifications and context-sensitive permission controls.

We highlighted the weaknesses in Android's traditional permission model, notably its lack of contextual awareness and tendency to overwhelm users with

decisions made prematurely, often out of the actual usage context. This approach has led to a diminished user ability to make informed decisions about their data privacy. The inadequacies of such a system are particularly evident when permissions granted for one context are misused in another, potentially leading to privacy violations without user awareness.

SCoPe addresses these challenges by employing a dynamic analysis methodology that tracks user interactions with the GUI and correlates these actions with the permissions being accessed. This allows SCoPe to offer a finer granularity of control, providing permissions on a per-feature basis rather than per-app. Such an approach not only enhances user understanding and control over data access but also minimizes interruptions, thereby avoiding 'warning fatigue' which can lead to desensitization to privacy risks.

The design and implementation of SCoPe leverage advanced input generation tools to achieve comprehensive app behavior analysis. By integrating techniques from both the ARES tool [19] and Monkey's randomness, SCoPe ensures high coverage and depth in-app behavior analysis, surpassing previous models. The system's ability to remember user preferences for each app further personalizes the experience, aligning with individual privacy expectations and behaviors.

The results from our investigation underscore SCoPe's high ability to recognize usage contexts, where it achieved an impressive accuracy rate of 84%. This high level is credited to SCoPe's comprehensive approach to gathering contextual information, which includes analyzing user interface interactions, permission usage logs, and even screenshots.

Our analysis also highlighted a significant finding: the majority of the apps examined accessed sensitive resources at least once, underscoring the widespread need for robust permission models that can effectively protect user privacy. Moreover, our study revealed a notable distinction between open-source and commercial apps. Open-source apps demonstrated a lower propensity to access sensitive resources across multiple contexts, likely due to the greater transparency and community oversight inherent in open-source development.

From a user experience perspective, SCoPe successfully minimizes unnecessary interruptions. By soliciting explicit user authorization only when resources are accessed in new contexts—distinct from those previously encountered—SCoPe

adeptly balances security with user convenience. This selective approach to permission requests ensures that user interactions are streamlined and less intrusive, enhancing the overall user experience. Moreover, this method helps in averting *warning fatigue*, a phenomenon where continuous prompts for consent can diminish the effectiveness of user decisions. This fatigue can lead users to grant permissions perpetually, just to avoid repeated interruptions [20, 12], which ultimately compromises their privacy and security.

Seeking to validate our solution in a highly critical context, we have considered critical security threads like logic bombs, which are malicious codes activated under specific conditions and can lead to severe outcomes like data theft. Traditional security measures often fall short of effectively countering such threats due to the broad permissions typically granted to applications. Recognizing this gap, SCoPe has been rigorously tested in our *fourth contribution* against such extreme scenarios, including logic bombs cataloged in the TriggerZoo dataset. This critical testing underlines SCoPe's capability not just in managing permissions but in actively involving users to intercept and mitigate potentially malicious behaviors activated by logic bombs.

Despite Android's system giving users some control, the lack of technical knowledge among the general people often leads to unintentional data exposure and privacy violations. Moved from this, as a *fifth contribution*, we delve into the application of Artificial Intelligence (AI), specifically through the use of Large Language Models (LLMs), to scrutinize the user interface (UI) of applications, assessing permission requests to offer dynamic, context-aware recommendations. To evaluate the efficacy of this AI-driven approach, particularly GPT-4's capacity to discern appropriate permission settings, we have established an experimental framework involving the analysis of app UI screenshots and related log data using SCoPe. Utilizing OpenAI's APIs, interactions with GPT-4-turbo and DALL-E are conducted through well-formulated prompts that guide the model's analysis. Through iterative testing, we refine our experimental methods and optimize the prompts to maximize the effectiveness of GPT-4 in identifying suitable and unsuitable permission requests. The results from these experiments offer compelling evidence of the potential of LLMs like GPT-4 to revolutionize

permission management, providing users with the tools to make safer, more informed decisions regarding their privacy and data security.

This approach, marrying explainable AI with user-centric permission management, signifies a significant stride towards creating a more secure and privacy-aware mobile ecosystem, through collaborative efforts among AI researchers, technology developers, and end-users.

In conclusion, this Thesis's findings firmly establish SCoPe as a powerful tool for advancing Android permission models, effectively enhancing user privacy protection. With its high accuracy, broad applicability, and user-centered design, SCoPe represents a significant advancement in the suite of security measures available for Android applications.

The remainder of this Thesis is organized as follows. In Part I we provide background notions 2 on the Android architecture and related works 3 on the main topics of the thesis: analysis of app permission usage, user-centered tools for permission management, fine-grained permission models, dynamic approaches to contrasts logic bombs in Android apps, and eye tracking in mobile computing privacy. In Part II we present a dynamic analysis tool called RPCdroid. Especially, in Chapter 4, we report the study that motivates this thesis, *i.e.,* the study aiming at highlights the motivations behind giving users more control over permission managements and creating a context aware permission model. In Part III, we report our studies assessing the effectiveness of Privacy Indicators, showing the importance of a better explanation of permission usage contexts in Chapter 5 and studying their effectiveness in tasks with different cognitive levels in Chapter 6. Part IV is the main contribution of this Thesis. In Chapter 7 we present SCoPe, a fine-grained, context-aware permission model for Android. In Chapter 8 we report a study on defusing logic bombs in Android application using SCoPe. In Chapter 9 we assess the effectiveness of an LLM-based permission model for Android. Finally, in Chapter 10 we conclude the thesis by summarizing the main findings achieved. In Appendix A, the candidate contributions are summarized, including *publications* related to this Thesis (with highlighted Candidate CRediT authorship contribution statement), *other publications*, and *software artifacts* released to the scientific community.

Part I

Background & Related Works

"Nothing in life is to be feared, it is only to be understood.
Now is the time to understand more, so that we may fear less."

Marie Curie

CHAPTER 2

Android Architecture

Contents

2.1 Android permissions 14

2.1.1 Permission Request Process 15

2.1.2 Permission Enforcement Process 17

Android is an open-source platform developed by Google and the Open Handset Alliance for mobile, wearable, and embedded devices. Launched in 2008, it has become the fastest-growing operating system [21]. Its popularity with developers is due to its open nature, which allows for quickly deploying complex apps across a wide range of devices using its powerful development framework. This benefits researchers, as they can study, assess weaknesses, modify, and improve the source code since it is freely available.

The core components of Android are organized in layers, as shown in Figure 2.1. Each layer utilizes the services provided by the layer below it to offer services to the layer above it. In particular:

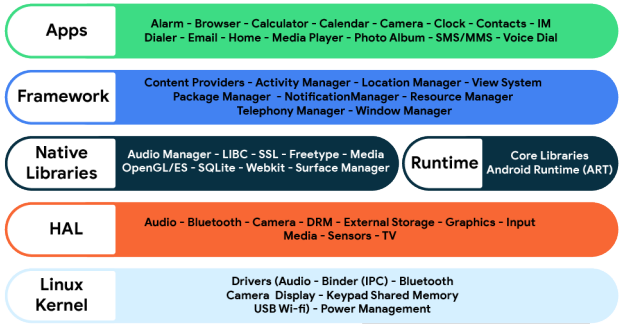


Figure 2.1: The Android software stack

- The platform is built on a customized version of the Linux kernel and provides essential operating system functions, such as process, memory, and hardware device management. It also manages various security mechanisms, including process isolation, file system permission management, and secure inter-process communication [22]. Currently, version 6.1 is being used on Android 14.

- The Hardware Abstraction Layer (HAL) provides a standard interface between software and hardware drivers. It is divided into modules, each of which implements the interfaces for a specific type of device component. This design ensures that Android remains agnostic about lower-level driver implementations [23].

- Native libraries are shared libraries written in C/C++. They are essential for many core Android components and services, including the Android Runtime, which is compiled directly from native code. Apps that have code written in a native language can access these libraries through the Android NDK [24].

- The Android Runtime (ART) is the managed runtime environment for Android apps and certain core system services. Each of these runs in an isolated process with its own instance of ART. ART offers superior performance over its predecessor, the Dalvik VM, which was replaced in

Android 5. This is because DEX bytecode files are no longer interpreted just-in-time, but compiled ahead of time, meaning they are translated into machine instructions at install time. Other new features include a more efficient garbage collection system and support for additional debugging tools [25].

- The Application Framework offers developers a collection of pre-compiled Java classes, interfaces, and code accessible through high-level APIs. These form the foundation for building apps [23] and include a rich and extensible view system for creating UIs, an activity manager that governs app life cycles, and context providers that enable access to data from other apps and facilitate data sharing. This approach promotes a high degree of reuse of basic components, modules, and services, which greatly simplifies the development process [24].

2.1 Android permissions

The permission mechanism regulates app access to sensitive data, such as contacts in the address book, received text messages or appointments in the calendar, critical resources, such as the camera, microphone, and body sensor, and restricted actions, such as connecting to a paired device. Its crucial role is to ensure user privacy and data protection.

To ensure process isolation, Android runs each app as a separate Linux user. Permissions are represented by unique strings, such as `android.permission.READ_SMS`, associated with the app's UID, which allows the use of certain sensitive APIs of the Application Framework. All permissions required by an app must be declared in advance in its Manifest file. This file provides crucial details about an app, such as its components, permissions, and necessary hardware and software features. Specifically, a `<uses-permission>` element is added for each required permission.

Android allows apps to define their own custom permissions in addition to the standard set of privileges already defined at the system level. These custom permissions can be used to control the sharing of resources and capabilities or the

access to exported components by other apps. All permissions defined by an app must be stated in the Manifest file by one or more `<permission>` elements.

Permissions, including those defined by third-party apps, are classified into three protection levels based on the potential risk associated with their use:

- Normal is a low-level privilege that provides access to isolated app-level features with minimal risk to other apps, the system, or the user itself [26]. For instance, `android.permission.VIBRATE`, which is required to control device vibration, falls into this category.

- Dangerous is a privilege with a higher risk level that grants access to the user's private data or control over the device. This access could have a negative impact on security [26]. For instance, the permission `android.permission.ACCESS_FINE_LOCATION` enables an app to access the user's precise location, and belongs to this category.

- Signature is a privilege that the system grants only to apps signed with the same certificate as the one that defined it [26]. System-defined permissions cannot be used by third-party apps. For instance, `android.permission.WRITE_SETTINGS`, which enables an app to read or write the system settings, falls into this category.

2.1.1 Permission Request Process

In addition to characterising the potential risk associated with a particular permission, the protection level also determines the procedure the system will follow when deciding whether or not to grant it to a requesting app [26]:

- Normal and signature permissions are granted automatically at install-time without requiring user approval. However, in the latter case, the requesting app must be signed with the same certificate as the declaring one, or else permission will not be granted. Before installing an app, it is always possible to check the list of permissions required by the app on the Play Store, and then decide whether or not to proceed [27].

- Starting with Android 6, dangerous permissions are requested individually at runtime. When an app attempts to access a set of protected resources, a dialogue is displayed to the user asking for confirmation before proceeding. If the permissions have already been granted, no prompt is displayed. Previous versions, on the other hand, used an install-time request model, which gave the user no choice but to either accept all permissions at once or delete the installation.

Whenever a user initiates a task or action that requires access to private data or protected resources:

1. The app should check if it has been granted the relevant dangerous permissions by passing its unique strings to the `ContextCompat.checkSelfPermission()` method. This method returns either `PERMISSION_GRANTED` or `PERMISSION_DENIED` [28].

2. In the first scenario, the app can directly use the appropriate permission-protected APIs of the Application Framework. Otherwise, it should first request permission by calling the `ActivityCompat.requestPermissions()` method. This method will prompt the user to approve or deny access.

3. After the user makes their choice, the request dialogue closes, and the callback method, `onRequestPermissionsResult()` in this case, handles the response, whether positive or negative.

With the current permission model, users can approve or deny an app's request to access personal data or perform restricted actions, and even revoke previously given consent directly from the system settings. Since Android 11, it is also possible to grant one-time access to location, camera, and microphone, providing temporary rather than permanent permissions [27].

Access rights are always linked to the entire app and not to its individual components or features [29]. Additionally, dangerous permissions are grouped according to the same resource or action. This means that if an app already has one permission from a group, it will automatically be granted another permission from the same group without prompting the user. For instance, an app that

has been granted read-only access to text messages can actually send them without further notice. This is because both `android.permission.READ_SMS` and `android.permission.WRITE_SMS` belong to the permission group `android.permission-group.SMS`.

On one hand, this reduces the number of requests presented to the user and minimizes their effort. However, on the other hand, it also limits the level of control, which is still too coarse-grained.

2.1.2 Permission Enforcement Process

Although the Linux kernel forms the core of Android, the Access Control List (ACL) mechanism enforces only a small number of permissions, all of which are classified as normal [30]. This mechanism associates each resource with different combinations of read (`r`), write (`w`), and execute (`x`) privileges for the owning UID (`u`), the owning GID (`g`), and everyone else (`o`). For instance, if an app declares the `android.permission. INTERNET` permission in the Manifest file, the corresponding Linux user is automatically added to the inet Linux group. This group has access to the relevant files and sockets, and all the checking is done at the kernel level [30]. All other permissions, including dangerous ones, are checked at the Application Framework level instead.

Android apps access all the services provided by the Application Framework through an inter-process communication (IPC) mechanism. The interfaces of these services are described using the Android Interface Definition Language (AIDL), and partial implementations, called stubs, are automatically generated based on the Binder protocol, which is an Android-specific IPC mechanism that also allows for remote method invocation. Each stub is an abstract implementation of its parent interface. It declares all the methods from the AIDL specification and must be extended by the actual service implementation. A proxy, which is also automatically generated, enables clients to call the service from remote processes, either directly or via a `Manager` class that wraps it and translates local method calls into remote procedure calls (RPC).

Currently, there is no centralized policy for verifying permissions when calling these services. Instead, the individual APIs that provide access to private data

or restricted actions must enforce the permissions associated with the critical resources they expose.

The service interface's implementing methods will explicitly invoke the permission validation mechanism when necessary to ensure that the calling app possesses all necessary permissions [30]. If the validation succeeds, execution proceeds normally. If not, the service immediately throws a `SecurityException`, thereby preventing access to the protected resources it is guarding.

Android offers various methods for checking permissions, such as `Context Compat.checkSelfPermission()` that allows an app to verify if it has been granted a specific permission or if it needs to request it from the user. Additionally, most Application Framework services use `ActivityManagerService. checkPermission()` to enforce permissions.

Ultimately, all requests are handled by the `PermissionManagerService`, which manages the process of granting and revoking permissions. The system component keeps track of all access rights associated with installed apps through the `checkUidPermission()` method.

CHAPTER 3

Related Works

Contents

3.1 Analysis of App Permission Usage 19
3.2 Studies on User Behavior Regarding Permissions . 21
3.3 User-Centered Tools for Permission Management . 23
3.4 Fine-Grained Permission Models 24
3.5 Dynamic approaches to contrasts Logic Bombs in Android Apps 27
3.6 Eye Tracking in Mobile Computing Privacy 28

3.1 Analysis of App Permission Usage

A prevalent issue is the presence of over-privileged apps that request excessive permissions. Tang *et al.* [31] found that out of the 10,710 apps they analyzed, as many as 76% exhibited this misbehavior. This number exceeds the previous findings of Felt *et al.* [30], who found that one-third of the 900 apps they considered

were over-privileged. Calciati *et al.* [32] pointed out that in 17% of cases, a new release of the same app introduces at least one more permission than the previous one, and only in 56% of cases is this permission actually used. This may explain the ever-increasing trend. Tang *et al.* [31] also discovered that third-party libraries integrated into apps are often responsible for using these over-claimed permissions, particularly the dangerous ones. Diamataris *et al.* [33] supports and strengthens this point with a study of more than 5,000 apps, which revealed that 65% of accesses to sensitive resources are made by external libraries. In 37% of cases, this data is used for profiling purposes. The ability to grant permissions, at least at the component level, is therefore critical.

Several studies have focused on understanding why certain apps require specific permissions. Wang *et al.* [34] proposed two approaches, based on static and dynamic analysis techniques, to infer the purpose of permission use from a predefined set of categories. An evaluation of 830 apps revealed that the dynamic analysis accurately determined the purpose of accessing the user's location in over 90% of cases. The authors also demonstrated the potential use of these categories to restrict resource access to specific functionalities but did not provide an implementation. Wei *et al.* [35] developed an NLP-based approach to identify links between the app description in the Play Store and the permissions requested. This highlights any inconsistencies, such as permissions that are not appropriate based on the features offered. However, the predictions were only accurate when a permission was highly correlated with the app category. For less relevant ones, the confidence was close to 50%, which is equivalent to a random guess. Wang *et al.* [36] showed that analyzing user reviews is more useful than relying solely on claimed app descriptions. The tool achieves an average accuracy and recall of nearly 90% in inferring 10 permissions from reviews. The descriptions only present the basic functionality of the app, while the users' comments reflect their actual user experience.

The current permission management mechanism has a significant limitation in that it grants permissions to the entire app rather than to specific features that require them. Hossen *et al.* [37] found that 46% of the 6,790 apps they analyzed used the same permission in multiple, different contexts. However, only 20% of the apps asked the user for this permission in more than one context. This means

that many apps use permissions that they would not explicitly prompt the user for if they were not already granted. In addition, 15% of apps ask for permissions in a context other than the one in which they are actually used. This emphasizes the importance of providing timely and well-contextualized notifications when apps access critical device resources and offering more precise control over permissions.

3.2 Studies on User Behavior Regarding Permissions

There has been extensive research on user behavior regarding the perception and management of permissions. Felt *et al.* [38] found that requiring permissions at install-time, before the user can use the app, does not allow for a fully informed decision. Users often accept all requests without consideration and lack an understanding of the meaning of permissions, which are only tied to the resources involved and do not indicate any associated risks. One of the open issues highlighted is the lack of context. It is unclear which app functionality corresponds to a particular install-time permission. The ask-on-first-use model was introduced precisely to address this limitation.

Wijesekera *et al.* [39] conducted a study on how users interact with the new run-time permissions. The findings suggest that certain permissions regulating access to harmless resources can be automatically granted to prevent warning fatigue. However, an explicit and contextualized request is necessary for permissions classified as dangerous. Participants expressed a willingness to deny access in one-third of cases, and 80% deemed at least one request as inappropriate. Reinfelde *et al.* [40] confirmed the positive reception of the new permission model by users. They found that users prefer run-time requests over install-time requests, which are more useful and effective. However, according to Scoccia *et al.* [41], permission-related issues are still prevalent. Their analysis of 4.3 million user reviews on the Play Store found that 8% of the collected apps received negative comments about permissions. In many cases, users complain that certain apps refuse to run without specific permissions or continue to request them with bad

timing. Additionally, if these requests are declined, users are often bombarded with frequent and repetitive prompts, leading them to accept to avoid further interruptions. Therefore, a more flexible approach is clearly necessary.

Cao *et al.* [42] investigated the factors influencing decisions regarding granting or denying permissions. 1,719 participants were surveyed immediately after being asked for permission in order to capture rationales, runtime expectations, comfort level, and willingness to grant temporary access. Out of the 36,000 decisions collected, study participants rejected around 17% of the requests. The study demonstrated that users are more likely to grant permission when provided with an explanation of why it is necessary. Conversely, they are more likely to deny a request when an app requests permission they did not anticipate needing. Baalous *et al.* [43] noted that the user's final decision is more influenced by the context in which the request is made than by the rationale message. Although clarity of context is important, it is crucial to note that any other purposes of use highlighted only in the rationale message may go completely unnoticed. Developers should follow guidelines for transparency and clarity when accessing resources and providing reasons for doing so. Shen *et al.* [44] remarked that the information provided to users in prompts is inadequate. Only 6% of the 4,636 participants in the study were able to correctly infer the scope of different permission groups from system messages, particularly the more coarse-grained ones that control different actions and sensitive resources. Only users who fully understand the meaning of a permission and the risks associated with its use grant it with due care. Palmerino *et al.* [45] proposed a change to the prompts shown to users, adding a less technical description of the permission and a list of other apps that have been granted the same permission on the device. This alternative model increased user confidence, despite the possibility of more informative and contextualized descriptions.

The effectiveness of privacy indicators has also been researched. To determine whether this mechanism improves awareness of sensitive resource usage, Guerra *et al.* [46] conducted a controlled experiment with 122 participants. The study found that these notifications were not helpful in cases where the context of resource use was implicit rather than explicit, which is when they would be most useful. In contrast, using a more visible notification, such as a toast, instead of

a dot on the status bar, was found to have a statistically significant correlation with user awareness of a resource being in use, particularly in non-explicit contexts.

3.3 User-Centered Tools for Permission Management

Several end-user tools have been proposed to enhance the native permission management mechanism from different perspectives. These tools often rely on creating a user privacy profile and automatically applying data access decisions. Liu *et al.* [47] conducted a study on the privacy settings of 4.8 million devices. They found that although individual users' preferences may appear to differ, it is possible to derive a small number of privacy profiles. Based on these profiles, they were able to predict with over 87% accuracy whether or not the user would grant permission to a certain app. A fully automated approach based on these profiles can respond to permission requests instead of the user. Other approaches learn from the user's previous choices without explicitly classifying them into profiles. Oglaza *et al.* [48] developed a personalized permission management system to establish policies for controlling access to resources on their device. The tool can automatically propose high-level policies based on the user's past preferences. For instance, it allows granting access to a user data group, which includes contacts and messages to all social apps, when it detects that this set of permissions is consistently given to apps in that same category.

One main criticism of these solutions is that users are often not privacy experts, which may lead to incorrect decisions. Rashidi *et al.* [49] developed a crowdsourcing-based framework for permission management. provides suggestions on granting access based on expert decisions. This allows even inexperienced users to make low-risk decisions with good accuracy. Gao *et al.* [50] proposed an approach based on text mining of app descriptions. Given the relationship between the functionalities and permissions of an app, users can be recommended to grant or deny requested permissions without relying on their historical decisions. Although these solutions help users make more informed and assisted choices, the

control options remain the same, as permissions are still managed at a coarse-grained level.

Fung *et al.* [51] designed multi-view prompts to provide users with the necessary information for informed decision-making. These prompts are tailored to the user's background and include assessing the risk associated with a given permission. However, the empirical evaluation showed that less experienced users are less interested in the new multi-view request model than in privacy-conscious ones. It would be appropriate to find solutions for the former. Additionally, the authors only developed a prototype of the tool and did not fully implement it.

3.4 Fine-Grained Permission Models

At the same time, research has also aimed to propose more adaptable tools that give users more control over how apps use their data. Before introducing the ask-on-first-use model, Nauman *et al.* [52] proposed a usage control model that enables users to specify the resources that a particular app can access through policies. This approach anticipated the ability to deny permission, as with the current model. Additionally, this tool offers the option to include specific conditions for automatically determining whether to grant consent. These constraints may include the user's location, the day of the week, or attributes associated with the app itself. However, writing these policies in XML format can be burdensome for users, especially inexperienced ones.

Additionally, the constraints do not refer directly to specific features or screens, which limits the ability to grant permissions at a more granular level than the entire app. Tiwari *et al.* [53] focused on addressing the first limitation by introducing the concept of context, which encapsulates information of various types that are used in specific circumstances. Using a user-friendly interface, users can create sandbox-like compartments and associate them with specific apps and permissions that they wish to grant them. However, this solution does not address the main issue of associating permissions with sub-components or features of an app.

Jeon *et al.* [54] defined additional sub-permissions for access to certain domains, individual address book contacts, single device status information, and specific

system settings that can be changed. Extending the list of permissions may appear to increase user control, but it is important to consider the potential drawbacks. As shown by Akhawe *et al.* [55], an increase in the number of prompts leads to a decrease in the user's attention span. This may result in the user accepting all prompts to continue using the app without further interruptions. This is why, in the current model, runtime permissions are requested at the group level rather than the individual operation level. In any case, managing permissions at the individual resource level, such as specific contact instead of the entire address book, can be a good mechanism.

Fu *et al.* [56] tackled the issue of sharing app permissions with third-party libraries. Their tool utilizes dynamic code instrumentation techniques to allow for separate permission management between the host app and external libraries, even when data is accessed in the background. While it does not address the problem of a permission being used in multiple contexts within the same app, this feature allows users to limit access to their data to only what is intended by the app developers. This is important because they are often unaware of the actions of third-party code integrated into their apps.

Scoccia *et al.* [57] focused precisely on the development of an approach for selectively granting specific features of an app certain permissions. The app undergoes a static analysis on a server before release to extract its components. The developer then defines the app's functionality in terms of title and description and maps it onto the extracted components. All API-sensitive calls are proxied and redirected to a permission management library added during repackaging. Upon first start-up, the user is presented with a list of the app's features and the permissions required, with the option to grant or deny them selectively. However, this approach requires additional effort on the part of the developer. They will not be able either to use code obfuscation techniques on pain of failure of the decompilation operation. Furthermore, this approach proves to be completely useless if the developer wishes to conceal blatantly malicious behavior, as they can freely map the use of a certain permission back to a lawful context.

In their early work on the automatic contextualization of data access requests, Huang *et al.* [58] utilized static analysis techniques to link the use of a resource with a particular top-level function, typically a method called after a user interaction.

Subsequent semantic analysis can determine whether the sensitive operation is related to user input or if there is a mismatch, preventing opaque access to the resource. Although the authors did not propose an end-user-facing tool, they did introduce the basic idea of tying permission requests to specific user actions on the UI. This concept was used to develop several permission models that allow for widget-level permission management. Li *et al.* [59] went a step further and proposed an approach based on both static and dynamic analysis techniques to map permissions to specific UI elements. At runtime, the UI hierarchy is navigated to find elements with which the user can interact. By instrumenting callback methods, the corresponding events can be captured. Then, the association between the sensitive APIs of the Application Framework and the permissions required to invoke them is used to derive this mapping. At runtime, users can see the permissions used by tapping on a particular widget. However, there is no way for them to block the use of a permission at this level. Duan *et al.* [60] addressed this issue, and also introduced the option to provide inaccurate or false data instead of denying permission, which can prevent app crashes. However, the proposed tool only detects contexts where a permission is first requested and then used. Still, it fails to identify cases where the permission is used without being requested, which is common when hiding data access from the user. Furthermore, it currently only offers fine-grained management of contact and location permissions. In the same field, Malviya *et al.* [29] employed static analysis to extract an inter-procedural CFG and trained supervised machine learning models to automatically classify the use of a permission by a single app feature as legitimate or not. They evaluated 49 licit and 40 malicious apps and found that their system achieves precision and recall close to 95%. However, the authors have not implemented a tool that end users can use. Additionally, assuming that all uses of permissions in benign apps are lawful for evaluating effectiveness is a strong assumption, as it is not always the case.

Finally, Du *et al.* [61] proposed an automated system for classifying data flows within an application as either positive or negative. In the first scenario, the application provides clear semantics regarding the flow of data or resource access on the user's device, while in the second scenario, transparency may be lacking. The system conducts a static analysis to link UI elements to data and control

dependency graphs, establishing a correlation between flow and context. Image recognition and natural language processing (NLP) algorithms are used to capture contextual semantics. An empirical evaluation was conducted to demonstrate the approach's ability to classify data accurately flows with a precision and recall rate of approximately 95%. Although the tool is not intended for end-users, this analysis can be used to manage permissions on a per-flow basis. Permissions can be granted automatically when the context is sufficiently explicit, and consent can be required otherwise.

3.5 Dynamic approaches to contrasts Logic Bombs in Android Apps

Over the past decade, numerous approaches have been proposed to automate malware detection. These approaches involve exploring static analysis techniques [62, 63, 64, 65, 66], dynamic execution [67], or a combination of both [68, 69, 70], as well as machine learning [71, 72]. Although effective on benchmarks, these techniques may fail to detect new attacks that use sophisticated evasion techniques. For example, attackers may employ code obfuscation [73] to bypass static analysis or hide malicious code behind triggering conditions during dynamic analysis.

Logic bombs can be used for various malicious activities, such as adware [74], Trojan [75], ransomware [76], spyware [77], and more [78]. As the trigger and the malicious code are independent of the core application code, logic bombs can easily be added to legitimate apps and repackaged for distribution [79, 80, 81, 82]. Therefore, detecting logic bombs is crucial, particularly in mobile devices containing critical personal information. Despite the challenges in detecting logic bombs, various approaches have been proposed in the literature. For instance, researchers have explored static-analysis-based heuristic or machine learning approaches [83] and dynamic-analysis-based approaches [63, 67, 84] to identify suspicious sensitive triggers. However, detecting logic bombs remains a problem [85], with both static and dynamic analyses often unable to detect such behaviors [86] [87].

Static analyses may be limited by many conditional statements in a given app code, making it challenging to identify suspicious sensitive triggers accurately. Moreover, a high rate of false positives suggests that a manual analysis should be conducted to prevent a legitimate app from being erroneously considered to contain a logic bomb [88].

TriggerScope [63] uses symbolic execution to detect logic bombs but is limited to certain trigger types and may not scale to large datasets. Dark Hazard [83] leverages a supervised learning approach with engineered features to identify sensitive triggers but flags up to 20% of apps, making it inefficient for isolating dangerous triggers. Unlike these approaches, our proposed in Chapter 8 focuses specifically on identifying suspicious, unexpected, and dangerous events and detecting logic bombs among them using dynamic analysis.

Difuzer [89] proposed an approach that relies on unsupervised learning techniques and specific trigger/behavior features to identify suspicious, hidden, sensitive operations. In contrast, our approach in Chapter 8 focuses on dynamic analysis and context-aware detection to prevent logic bombs triggered in dangerous contexts from being executed without the user's approval, allowing for targeted mitigation strategies. Additionally, our approach does not block app execution in case of a false positive but instead informs the user of sensitive resource usage, allowing for continued app use. In the event of a false positive, our system does not block the app's execution or prevent the user from acting. Instead, it informs the user that a sensitive resource will be used in that context, allowing the user to make an informed decision about whether to proceed or not. This capability is an improvement over existing techniques that rely solely on binary classification and often generate false positives, leading to unnecessary blocking of app execution.

3.6 Eye Tracking in Mobile Computing Privacy

The application of eye-tracking technology in understanding user interaction with privacy and security features on digital platforms is emerging as a crucial research direction. Studies by Furman *et al.* [90] and Egelman *et al.* [91] have employed eye tracking to evaluate how different information presentations affect user decisions in authentication scenarios. These studies reveal that while

additional information may increase reading time, it doesn't necessarily influence decision-making processes such as using Facebook Connect. The pioneering work by [92] demonstrated the broad applications of eye tracking across various fields, providing a foundation for its integration into privacy studies. Building upon this, research by [93] utilized eye tracking to probe the eye-mind connection, revealing the subconscious processes that guide visual attention and, by extension, privacy-related behaviors on digital platforms. Expertise in computer security also plays a significant role in user engagement with security features. Arianezhad *et al.* [94], monitoring eye movements, found that individuals with security knowledge spend more time looking at security indicators. Similarly, Whalen *et al.* [95] demonstrated the tendency for users to overlook security indicators without explicit prompting, underscoring the potential for inattentional blindness in digital security contexts. As eye-tracking devices become more accessible and integrated into everyday technology, active eye-tracking applications are being explored for security purposes. Miyamoto *et al.* [96] developed EyeBit, an eye-tracking system that encourages users to verify the URL before inputting sensitive data to combat phishing. Installing input fields only after confirming the user's gaze on the URL bar is an innovative approach to instilling secure online behaviors.

Despite the established utility of eye tracking, its application to privacy indicators (PIs) on mobile devices remains underexplored. Conventional studies have simulated user interactions with app prototypes and followed up with questionnaires to gauge resource usage awareness post-interaction [97]. Our research diverges from these methodologies by employing real-time eye tracking to capture immediate user responses to PIs. This direct measurement of attention allocation offers a nuanced understanding of user engagement with PIs and addresses the potential shortfalls of self-reported data.

The challenges of ensuring user attention to privacy prompts, as investigated by [98], highlight the importance of overcoming habituation. Our study aims to mitigate this through the objective analysis provided by eye tracking, contrasting previous approaches that lacked this real-time evaluative component. By doing so, we align with the recommendations by [99] and [100] for more personalized and contextually appropriate privacy controls, aiming to present a comprehensive

overview of how users perceive and process privacy notifications within their digital experiences.

Our research contributes to this dynamic field by investigating the real-time detection of Privacy Indicators using smartphone-embedded eye-tracking technology. We assess not only the attention drawn by these indicators but also their design and position relative to user gaze patterns during various interactive tasks. By doing so, we address the gap in understanding the immediate impact of PIs on user behavior and awareness in mobile computing.

Part II

Enhancing Android Security: A Dynamic Analysis with RPCDroid

"Data is the pollution problem of the information age, and protecting privacy is the environmental challenge."

Bruce Schneier

CHAPTER 4

RPCDroid: Runtime Identification of Permission Usage Contexts in Android Applications

Contents

4.1 Introduction 34

4.2 Design and Implementation 36

4.2.1 Used technologies 37

4.2.2 System architecture 38

4.3 Preliminary Evaluation 44

4.4 Final Remarks 47

4.1 Introduction

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 [7]. Android's permission management mechanism plays a crucial role in safeguarding user privacy. A recent empirical study by Scoccia *et al.* [101] 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, several improvements have been made to the authorization system, primarily aimed at increasing security and control over how and when applications access personal data. However, user awareness about when and how the permissions are exploited by each application is still lacking, and so is the flexibility of the current approach, contributing to bad practices like users grating permissions and ignoring the possible impact on their privacy [101]. 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 inherit 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 [12].

In this first 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 real applications potentially exploit the above-mentioned scenario. 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's execution, the tool generates two detailed logs, one containing all the user-generated events and the other with all the permissions accessed by the app. These logs allow us to identify which actions led to access to critical resources.

During a preliminary tool evaluation, 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 tool's effectiveness using 10 widespread and popular applications, with the aim of investigating whether real mobile applications have 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.

4.2 Design and Implementation

In this work, we want to assess the effectiveness of adopting the usage context as a means to discriminate between different kinds of access to the same sensitive resource. To this aim, we developed RPCDroid, a tool to monitor the execution of 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. It collects helpful information to identify resource usage contexts. Once the monitored application is started, RPCDroid Analyzer starts tracking any access to resources that require dangerous permission at the system level (e.g., the camera, microphone, storage, or location) through a dynamic injection mechanism based on hooking and callback techniques. These include all user actions performed at the interface level and 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 user interaction, the tool can be configured to execute random UI interactions on the apps and observe the permission requests. In both cases, RPCDroid automatically associates the observed permission requests and UI actions to categorize each resource's use 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 with details on its components.

4.2.1 Used technologies

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 [102]. 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 [103]. 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 [104].

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 [105].

Monkey exerciser

Monkey[1] 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 [106], 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 [107].

4.2.2 System architecture

RPCDroid was designed according to the principle of functional decomposition to facilitate the system development process and, subsequently, ensure a high degree of code maintainability. 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 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

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

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 generate 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 contains the code to be injected.
- Hooks defines 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 with. 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 with. 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 of the parameter) but also to influence its execution or to modify its return value.

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 classes, `ActivitySpecificCallback`

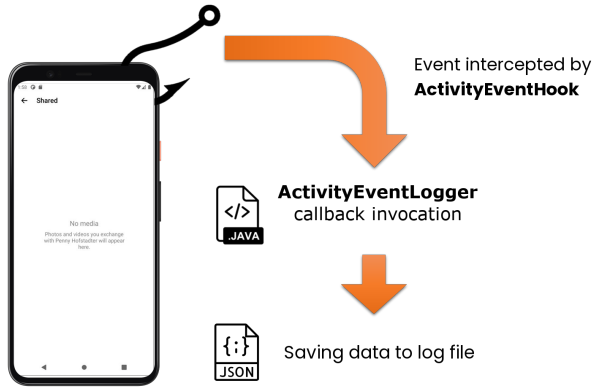


Figure 4.1: Process of *logging* an event related to an Activity

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 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 4.1) or a view (Figure 4.2) is reported in the 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.

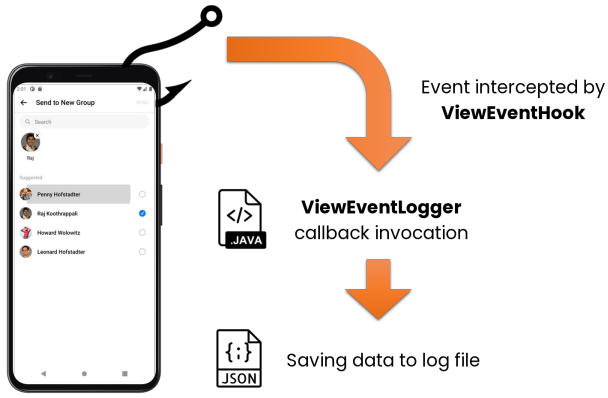


Figure 4.2: Process of *logging* an event related to a View

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`, 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 an *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.

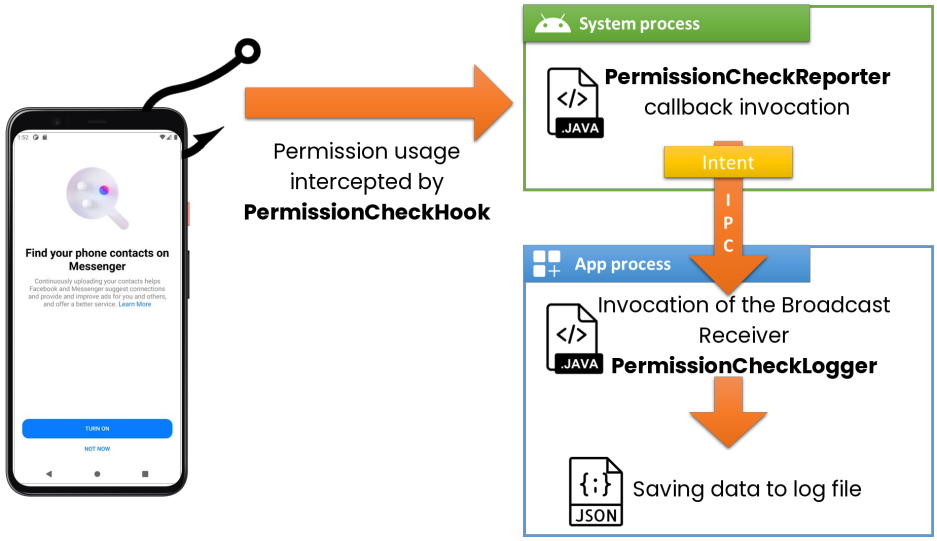


Figure 4.3: Logging process of permission usage

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 4.3).

For this reason, the entire *logging* process is delegated to a particular Broadcast Receiver[2], `PermissionCheckLogger`, which is automatically injected and registered by RPCDroid when a monitored application is launched, and the *callback* merely sends it an Intent[3] containing all the information about the intercepted request (such as the permission name and the *timestamp*).

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

[3]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*).

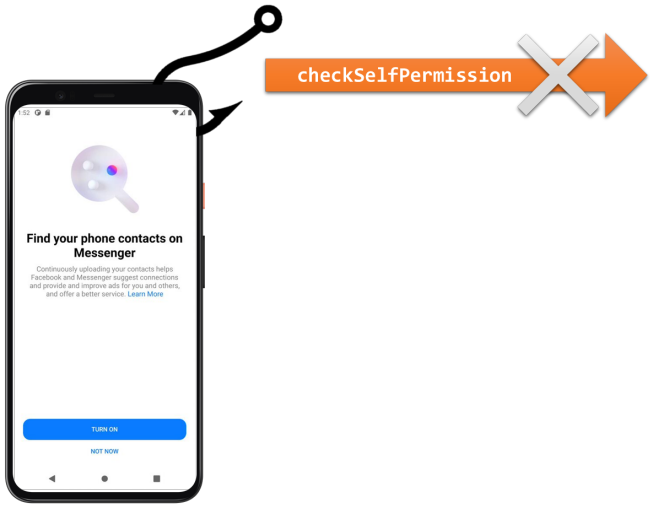


Figure 4.4: Checks made through `checkSelfPermission` are ignored to reduce false positives

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 [108], to exclude from the log these requests (see Figure 4.4).

Outputs

The output of the monitoring process for each app includes:

- One or more log files[4], 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 and appropriately organized into subfolders.

4.3 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 apps within the main Play Store. We excluded those apps 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[5].

In Table 4.1 we report the study results. 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

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

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

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 contexts. In particular, we do not want to have any access to a resource be considered a new context as this would reduce our approach to an "ask each time" rule. Still, 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 of accesses to four specific resources: 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 4.1 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.

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 behavior. Instead, we noticed that most of the apps 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. The precise location is another resource 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 4.1 is only a subset of all the possible access contexts. On the other hand, the use of pseudo-random events does not invalidate our results for all the apps showing multiple contexts per resource because all the contexts 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 sessions, were essential to mitigate such a bias.

It is important to note that mere access to the same resource in multiple contexts is not proof of malicious behavior (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 behavior. In Android 12 and later, Google introduced *Privacy Indicators*. Every time the camera or microphone is accessed, an LED is activated on the screen to inform the user when and where an app is accessing these resources. To assess if they are effective in alerting users about resource usage, we conducted two studies in Chapter 5 and Chapter 6. Even if this improves usage awareness, it is still limited to a few resources (camera, microphone, and location). Moreover, access to the resource is granted, potentially leading to privacy issues.

Table 4.1: 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

4.4 Final Remarks

In this first work, we presented RPCDroid, a tool for monitoring the execution of an Android mobile application that accesses specific device resources that require dangerous permissions. Such types of resources need to be explicitly authorized by the user. However, once 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 it difficult for the user to discriminate between the features that are allowed to access the permission and those that should be blocked.

To assess the tool's effectiveness in automatically discriminating between different kinds of access to the same sensitive resource, we conducted a preliminary study on 10 popular apps. The apps have been automatically run by using Monkey, a program that generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as several system-level events [107]. The results confirm that, in many cases, when the app accesses a sensitive resource, it does so in more than one context. Even if the most recent versions of Android improved awareness by enforcing the privacy protection mechanism through special LED indicators, nothing prevents the app from exploiting the acquired permission.

In this Thesis, we refined this tool, moving its focus to a more proactive approach, conducting a more exhaustive analysis, and testing it on malicious apps.

Part III

Advancements in Android's Permission Model: A Deep Dive into the Role and Impact of Privacy Indicators

"Security is always excessive
until it's not enough."

Robbie Sinclair

CHAPTER 5

An Empirical Study on the Effectiveness of Privacy Indicators

Contents

5.1 Introduction 52
5.2 Empirical study Design RQ_1 55
5.2.1 Context Selection 55
5.2.2 Experimental Procedure 57
5.2.3 Data Analysis 63
5.3 Results RQ_1 65
5.3.1 Results Analysis 66
5.3.2 Discussion 69
5.4 Evaluating an alternative solution RQ_2 73
5.4.1 Empirical Study Design 75
5.4.2 Empirical Study Results 75
5.5 Threats to Validity 77
5.6 Final Remarks 80
5.6.1 Data availability 81

5.1 Introduction

Modern mobile devices are shipped with increasingly advanced hardware and software components capable of integrating and extending the possibilities of the device. Managing access to these resources represents a technological challenge for OS vendors. Many of these components produce or manage sensitive information, which could be misused to profile users or even spy on them.

Examples of such privacy-sensitive hardware components are the camera, the gyroscope, the biometric sensors, the microphone, the GPS receiver, and the external storage, while software components that can represent a threat to privacy include the media library, the address book, the phone call, and SMS manager.

Many applications need to legitimately access one or more of these resources in order to provide advanced functionalities or to ensure a better user experience. Therefore, a mechanism to control access to the available resources must be provided.

The management of access permissions to sensitive resources of modern mobile devices is delegated to a particular component of the operating system whose task is to mediate the requests of each App according to the preferences specified by the user. For example, in Android to read the SMS within the cellphone SMS list, the App must obtain the READ_SMS permission.

Unfortunately, the decision to allow or deny access to a system resource cannot be entirely delegated to the operating system. Indeed, even if a generic clue can be found in the app category – *e.g.,* we expect that a voice recording app must be allowed to access the microphone and that in order to scan a QR code an app should be granted access to the camera – it is not possible in advance to establish if and when a specific app should be allowed to access a resource. Furthermore, it has been highlighted that even popular apps often use the same resource within different usage contexts [109]. For this reason, the choice is generally left to the end user.

Over the years, there have been many improvements to the authorization system, which has increased the perceived level of security, especially for what concerns the user privacy protection. The reasons why we still cannot rely on an effective mechanism to protect users from possible violations of their privacy are

various, including the need to offer a simple and effective solution, where the user is not continuously interrupted by the authorization system – often without even knowing the effects of one choice over another – which increases user fatigue; and the timeliness of controls, *i.e.,* user do not want to interrupt the regular use of the App, while maintaining temporal proximity between the access request and the actual use of a resource [110].

Starting from Android 12, new improvements have been introduced to the permission model regarding the use of resources and user awareness. In particular, Privacy Indicators (PI) have been introduced, which are notification indicators aimed to inform the user, through a colored dot on top of the screen, about the use of a device resource, such as the camera or the microphone (Figure 6.1).

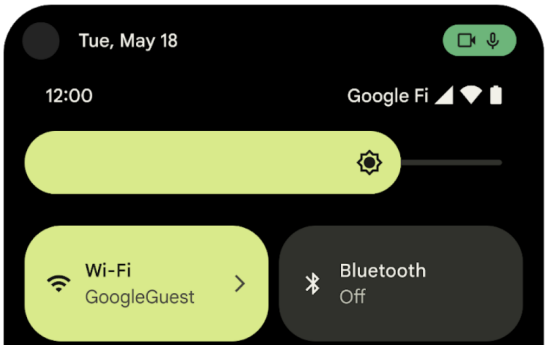


Figure 5.1: An example of Privacy Indicator in action on Android 12.

This mechanism could be a valid compromise between privacy awareness and non-intrusiveness, since the user is not continually interrupted by the system, but is nevertheless warned about the resource usage promptly. However, while ensuring an adequate user experience, there is a risk that users will not effectively identify and recognize such warnings.

Although there is evidence that attention can operate on or be drawn to unconscious stimuli, various studies question on awareness without attention in the case of multiple visual stimuli [111]. In most cases, objects or data relevant to the task can be identified in advance. During the actual visualization, the viewer's visual system must focus its attention on these objects and data in order to complete the task [112]. Thus, when participants are focused on the primary

task, a new and unexpected stimulus could remain unnoticed. This is known as *inattentional blindness* [113].

In this work, we investigate the effectiveness of PI on users' perception of resource utilization.

To this aim, we planned and conducted a controlled experiment with 122 participants to explore the effectiveness of PI in making users aware of resource usage in two contexts: *latent* and *explicit*. In *latent* contexts, resources – camera and microphone, in our study – are accessed even if they are not necessary for the task and the apps do not present any preview or animation on the device display while using a resource, except for the PI. In *explicit* contexts, apps actually need to access the resource to complete the task. Moreover, the expected behaviour is implemented when using a resource, such as camera preview or microphone icon animation. We designed four tasks in which developers were asked to perform actions on four popular Android apps, *i.e.,* WhatsApp, Facebook Messenger, Spotify, and Twitter. Each task was administered with two treatments, *i.e.,* visible or non-visible PI, leading to eight possible combinations. To run the experiment, we implemented eight interactive prototypes: such prototypes aim at simulating the Android OS and guiding the participants through the execution of the tasks while allowing us to fully control the PI visibility and other possible confounding variables (*e.g.,* notifications).

We performed statistical analysis, and our results show that the association between the presence of PI and the identification of resource usage is not statistically significant. Besides, the effect size is very low. This demonstrates that, while a slight increase in identification can be observed in some cases when PI is shown to the user, such a feature is still not entirely adequate to make users aware of camera and microphone usage. Motivated by these findings, we introduced an enhanced implementation of privacy indicators aiming to overcome the limitations identified with the standard PI. Thus, we extended our study to investigate if users' awareness of app resource usage can be increased. We used the same design we previously adopted while changing the treatments, which this time included (i) Android PI and (ii) our enhanced PI. This time, we involved 38 participants. Our results clearly show that users are significantly more aware of resource usage when using our enhanced PI. This shows promise for alternative strategies that could

further enhance such privacy features introduced in Android 12. To summarize, the main contributions of this work are:

- We investigate how effective Android privacy indicators are in making users aware of resource usage. Our results show that the currently used version of PI does not significantly impact the user's awareness;
- We assess the effectiveness of a more prominent and informative privacy indicator aimed at increasing the user's awareness of resource usage. Compared to Android PI, our results show that such an alternative visualization significantly improves the user's awareness.

Android maintainers can benefit from our work because they will obtain evidence about users' consciousness of privacy indicators and how to possibly improve such an important feature.

5.2 Empirical study Design RQ_1

The *goal* of our empirical study is to investigate whether PI can properly notify Android users about the use of a resource such as a camera or a microphone. Moreover, we aim to detect differences in identifying the use of the two resources between different contexts and whether one resource is more prominent than the other. The following research question guided our study:

RQ1 *How effective are Privacy Indicators to help users identifying when a resource is used by a mobile application?*

We conducted a controlled experiment with human participants to answer this RQ1.

5.2.1 Context Selection

The context of our study is composed of *objects*, *i.e.,* apps to use in a simulated Android environment, and *subjects*, *i.e.,* human participants representing common Android users.

As for the objects, our objective was to select four popular Android apps based on which we could create tasks that participants could complete in a few minutes. We chose two messaging apps (WhatsApp and Facebook Messenger), a music app (Spotify), and a social network app (Twitter). We selected these apps because they are among the top 30 most popular free apps in the Google Play Store[1]. More specifically, WhatsApp and Messenger are ranked first and second in the *Communication* category[2], Spotify is the most downloaded in the *Music And Audio* category[3], and Twitter is ranked fifth in the *Social* category[4]. Our choice of popular and trusted apps for the experiment was based on the assumption that users are more inclined to grant full resource usage permissions to these apps. Typically, users might restrict camera or microphone permissions for less trusted apps to one-time use. In contrast, they might provide full access to apps they regularly use and trust, such as those employed in our study. We assume that the role of privacy indicators is particularly relevant in the context of these frequently-used and trusted apps.

We involved a total of 149 subjects (participants) in our study. A subset of them (27) were involved in a pilot study we ran to test our experimental procedure (more on this later), while the remaining 122 were involved in our main experiment. Since the target population of our study is the whole population of Android users, we had no particular requirement in terms of skills or knowledge. We only ensured that all of them used an Android device as their primary phone. We made this choice because permission management among different OSs can be different and this could have been a bias for the experiment. Note that the investigator records the information obtained in such a manner that the identity of the human subjects cannot readily be ascertained, directly or through identifiers linked to the subjects.

[1] https://www.androidrank.org/android-most-popular-google-play-apps?price=free

[2] https://www.androidrank.org/android-most-popular-google-play-apps?category=COMMUNICATION

[3] https://www.androidrank.org/android-most-popular-google-play-apps?category=MUSIC_AND_AUDIO

[4] https://www.androidrank.org/android-most-popular-google-play-apps?category=SOCIAL

5.2.2 Experimental Procedure

We considered three independent variables. The first one represents the support of Privacy Indicators at OS-level (PI): such a variable allows us to simulate both the Android versions lower than 12 (without PI) and greater than or equal to 12 (with PI). The second variable is the Context Behavior (CB), *i.e.,* the way in which the app uses the resources (camera and microphone): an app might use a resource either in a legitimate way (explicit) to provide a feature to the end user (*i.e.,* taking a picture or registering a voice message) or in a latent way (non-explicit) if the resource is not strictly required, but the app uses it anyway. The last variable is the resource used (R): as previously mentioned, the PI are shown when one of the two resources is used, *i.e.,* camera and microphone. In our experiment, PI is the factor, with two treatments (*i.e., visible* — pre-Android 12 — and *not visible* — post-Android 12). CB and R are co-factors we consider to instantiate the tasks. We consider such co-factors to observe to what extent the treatments are effective in different contexts. The dependent variable is the correctness in the identification of cases in which an app uses a sensitive resource. Such a variable has two possible values: *true* (if the app uses a resource and the participant is aware of this) and *false* (otherwise). If the treatment in which PI are used works, we should observe a higher percentage of participants' awareness.

Task Definition and Group Assignment

Given the four selected apps, we defined four tasks by combining the values of the two co-factors (CB and R). Specifically, the tasks are the following:

T_1 [CB = Latent; R = Camera] The participant has to open the Twitter app and create a new tweet. A tweet composition window appears, with the message content already written, and the participant has to publish it by tapping the "Publish tweet" button. We simulate that the camera is used while the participant is composing the tweet.

T_2 [CB = Explicit; R = Camera] The participant has to open the WhatsApp app and look for a chat with unread messages. Then, the participant has to take a picture and send it as an attachment.

T_3 [CB = Explicit; R = Microphone] The participant has to open the Facebook Messenger app and search for a chat with a specific friend. The participant has to start a voice call with such a friend and close the call after a few seconds.

T_4 [CB = Latent; R = Microphone] The participant has to open the Spotify app and search for a specific playlist from the list. A list of songs appears, and the participant is asked to play a specific song. We simulate that the microphone is used while the participant is looking for the song from the list.



Figure 5.2: An example of prototypes created using Figma.

The first requirement for our experiment was to make sure that it was feasible to execute it by many participants. Therefore, we wanted them to use their own smartphones, without asking to install apps or modified Android versions. Also, most importantly, we wanted to have the opportunity to manipulate the PI variable for all of them to administer the two treatments. To meet such

requirements, we decided to create simulations of the Android OS that (i) guided the participants during the task execution, (ii) allowed us to explicitly show or hide PI, and (iii) could be accessed from any smartphone through the web browser, without specific requirements. We used Figma to create such simulations. Figma (similarly to others like Sketch and Adobe XD) is typically used to define high-fidelity prototypes of mobile apps, but we used it to prototype the entire Android OS (and the apps used for the respective tasks), by simulating it as closely as possible.

Figure 5.2 shows two screenshots of prototypes created with Figma. It is worth noting that they accurately reproduce the actual apps user interface and behaviour, also simulating the interactivity of these apps. For the implementation of the prototypes, we assumed that the permissions for camera and microphone had been granted by the user in a previous use of the app. Such a choice was made to avoid the system popup asking for permission granting, which would affect the participant's answer about using a resource, which is typical in an app's normal usage. We also decided to limit users' interactions on the prototypes, reducing the likelihood of participants getting stuck during the steps to be performed.

Table 5.1: Tasks and values assigned to the PI, CB, and R variables for each group.

Group 1	**PI**	**CB**	**R**
Task 1	visible	latent	camera
Task 2	not visible	explicit	camera
Task 3	visible	explicit	microphone
Task 4	not visible	latent	microphone

Group 2	**PI**	**CB**	**R**
Task 1	not visible	latent	camera
Task 2	visible	explicit	camera
Task 3	not visible	explicit	microphone
Task 4	visible	latent	microphone

We defined two groups of participants. We asked each group to complete the same four tasks in the same order. The only difference was the PI value for each task: In the first group, T1 and T3 were assigned with the PI visible treatment, while T2 and T4 with the PI not visible treatment; in the second group, we assigned complementary treatments (PI visible for T2 and T4 and PI not visible for T1 and T3). Therefore, in total, we prepared eight prototypes. We summarize the groups and the values of the three independent variables we consider in Table 5.1. Each participant was randomly assigned to one of the two groups.

Experimental Protocol

The experiment consisted of 5 phases. In the *preliminary phase*, we acquired basic information about the participants; in the *learning phase*, we instructed them on the PI feature introduced in Android 12; in the *task execution phase*, the participants were asked to complete one of the tasks; finally, in the *response acquisition phase*, we collected information that allows us to compute the dependent variable, while in the *matching phase* we allowed participants to review their response based on static screenshots of the app. We provide an overview of the phases in Figure 5.3 and we report below the details for each of them.

The experiment was conducted using two devices. Participants were required to use a laptop/tablet and a smartphone concurrently. On the laptop/tablet, participants interacted with a web application that guided them through the entire experiment, presenting the instructions, tasks steps to follow, and questionnaires. The tasks, tied to smartphone use and PI, were performed on the participants' personal smartphones. During the experiment, two of the authors and two collaborators monitored the participants to ensure they completed the tasks independently and correctly.

Preliminary phase. In this phase, we administered a pre-questionnaire to the participants through the web app. We aimed at acquiring both personal information (age, experience in terms of number of years they were using smartphones, instruction level, knowledge about smartphone operating systems, if they are specialized in computer engineering or computer science and their concern for privacy) and technical information (smartphone model, Android version installed).

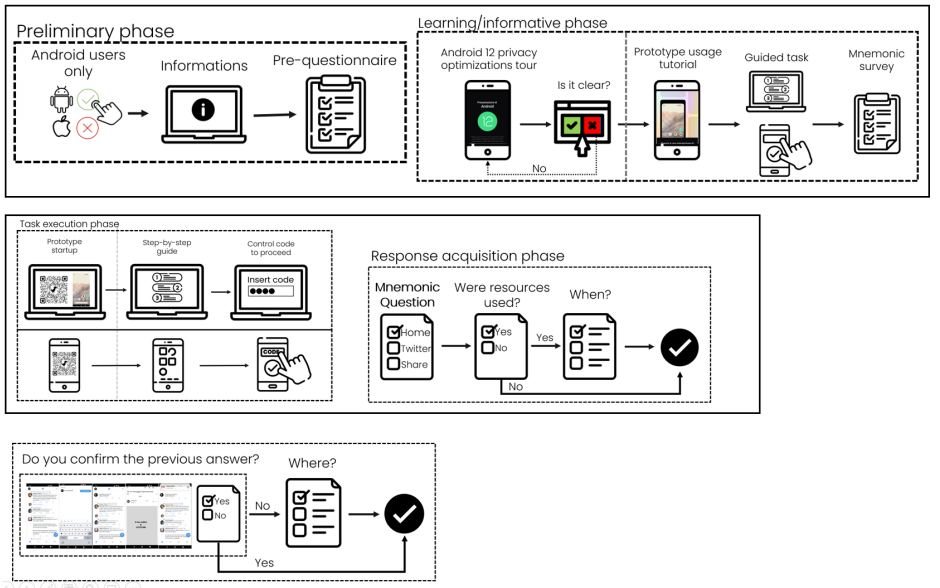


Figure 5.3: Experimental protocol.

Learning/informative phase. This phase aimed to ensure that all the participants were fully aware of the PI feature. We did not want to explicitly instruct the user about PI since there would have been a risk that they unnaturally focus on them during the tasks. Thus, we also instructed them about other Android 12 features (*e.g.,* privacy dashboard).

We first asked the participants to read from their laptop/tablet the Google official page that contains the features list with visual examples of them and asked to take time to navigate through the documentation. Then, we demonstrated such features through an interactive prototype (similar to the ones they would later use for the tasks), which was visualized directly on the users' smartphones, effectively simulating the behavior of Android 12. This provided the participants with a realistic experience of using the new features before proceeding with the tasks. In the end, we asked them if they fully understood those features, forcing them to start over in case of a negative answer.

After that, participants had to complete a guided task to get familiar with the prototypes with which they would later complete the tasks. We preliminarily

informed the participants on how the prototypes work (*e.g.,* not all the OS features are available) and we explicitly told them they could assume that microphone and camera permissions in the simulator had been granted for any of the apps used during the tasks. Then, we asked them to use their smartphone to access a tutorial with a guided task similar to the actual ones (next phase). During this phase, participants were also presented with a mnemonic survey consisting of three questions related to the actions performed during the guided task and the elements present in the UI. This survey was introduced to further familiarize participants with the environment and the tasks they would later encounter. In this way, we tried to reduce the influence of the learning effect. Indeed, in case we let the participants start completing the tasks, they would not have been fully aware of the process and, most of all, of the questions that would have been asked after the task. As a result, all the participants could perform less well on the first task.

Task execution phase. Participants were asked to complete the four tasks, with treatments assigned based on their groups. For each task, we showed on the web app a QR code that linked to the prototype implementing the task with the given treatment. Then, the web app (on the laptop/tablet) provided the instructions on the actions to perform, and participants executed them on their smartphones. The prototypes provided a verification code to the participants at the end of each task that they had to insert in the supporting web app on the laptop to proceed with the next phase. We did this to ensure participants completed the tasks before answering the questions.

Response acquisition phase. The web app showed a questionnaire at the end of every task, which included a control question and two questions about resource usage. The first one was designed to ensure that participants were not carelessly completing the tasks and were, instead, maintaining focus. Note that such questions were not used in our design (*i.e.,* we do not use them for data analysis), but rather to reinforce participants' attention during the tasks. These questions were task-specific and regarded the actions performed (*e.g.,* what was the name of the person to whom you sent a message?). As a second question, we asked whether the app used any resources (*i.e.,* camera or microphone). If the answer was affirmative, we asked to indicate at which point the resources were

used (*e.g.,* while writing the message). We later use the answers to the second and third questions to measure the dependent variable.

Matching phase (only for tasks PI = visible). In this last phase, we showed participants some screenshots of the tasks completed and we asked again the questions from the previous phase. One of the screenshots included the screen in which the PI was visible. This phase allows us to understand if there are differences between the answers provided only based on the information acquired while performing the tasks (*e.g.,* the participant did not pay enough attention to the PI because he/she was doing something else) and the ones provided while not performing a task. During this last phase, the user could confirm the answer given during the previous step by moving on, or change their answer by selecting from the proposed screenshots the one that is correct for them.

Pilot Study

Before running our experiment, we conducted a pilot study with 27 participants in an offline setting. Such a study was promoted from December 29, 2021, to January 14, 2022. We tested the previously-described experimental protocol and we checked whether the tasks were feasible, whether the instructions were sufficient and explicit, whether the prototypes worked on different devices and their usability, and whether users experienced problems or had doubts during execution. None of the users reported problems in understanding and completing the tasks.

5.2.3 Data Analysis

The values of the independent variables we are interested in are determined based on the task and the group. We compute the value of the dependent variable based on the answers provided in the response acquisition phase. Specifically, if the participants answered "yes" to the second question (*i.e.,* they noticed that the app used a resource) and if they correctly identified the moment the resource was used (third question), the value of the dependent variable was *true*, otherwise it was *false*.

To answer our RQ1, *i.e.,* to determine whether there is a statistically significant association between the presence of PI and the identification of resource use, we use a statistical independence test. Such tests are used to determine if there is a significant relationship between two categorical variables. Specifically, we use Fisher's exact test [114]. The null hypothesis (H_0) is that there is no association between the fact that the OS provides PI (variable PI) and the awareness of the use of resources (dependent variable). Conversely, the alternative hypothesis (H_1) is that the presence of PI helps Android users in identifying resource usage. We reject the null hypothesis if the p-value is lower than or equal to 0.05.

We run such an analysis in several scenarios:

- We considered only the tasks where the CB was latent (T1 and T4), to understand if the PI helps in possible malicious usage of the resources.

- We consider only the tasks where the use of the resources is explicit (T2 and T3), to understand if the PI still helps when the user probably already knows that the resource is used (*e.g.,* through the preview of the camera).

It is worth noting that there is no way participants in Group 1 and Group 2 could notice resource usage in Task 4 and Task 1 since they were *latent* and the PI were hidden. Thus, if such participants reported that they identified the resource usage, we excluded them from our study since this was a sign that they did not perform the task with sufficient attention, they provided random answers, or they could have not understood what they were asked to do.

In addition, we analyzed to what extent the resource (variable R) influenced the identification of PI. Since we run multiple comparisons, we use Benjamini & Hochberg method [115] to adjust the p-values.

Finally, we report in how many cases the correctness achieved in the *matching phase* (in which we showed pictures of the tasks performed with the PI visible) is higher than the one achieved after simply using the app. If, after this phase, the percentage improves, we can conclude that PI can capture the users' attention, but the focus on a particular task can make them miss the visual stimulus.

5.3 Results RQ_1

Figure 5.4 summarizes the demographics of the sample involved in our study. Since, as previously reported, we randomly assigned each participant to one of the two groups, there is a small difference in terms of number of participants in them (63 in Group 1 and 57 in Group 2). The two groups are generally balanced in

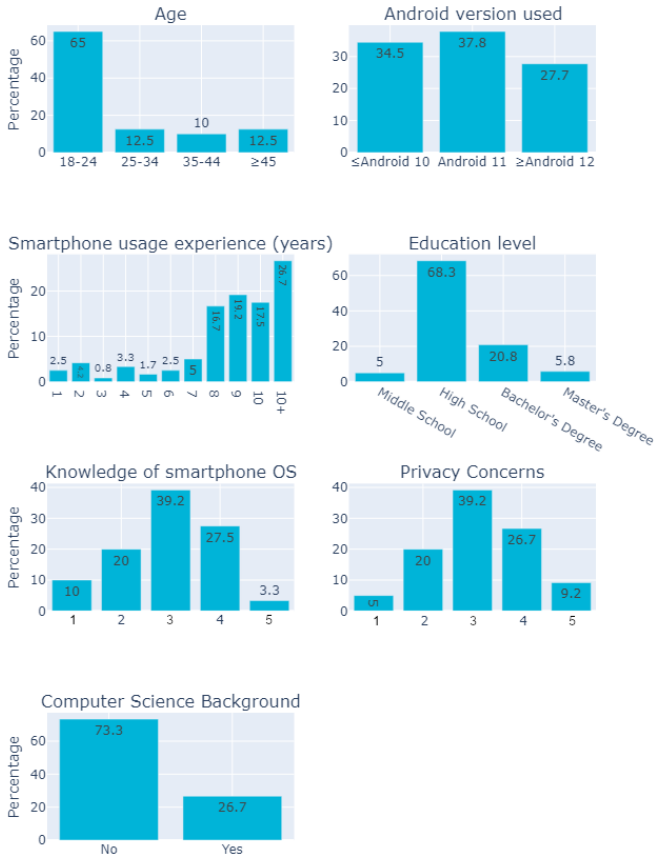


Figure 5.4: Demographic users information.

terms of all the demographics we acquired. A quarter of the participants reported experience (education or job) in Computer Science (17 in Group 1 and 15 in

Group 2). Concerning the age, we can observe that the majority of participants (∼65%) are in the range 18-24 years old (78, *i.e.,* 43 in Group 1 and 35 in Group 2), while we have ∼12.5% participants in the range 25-35 (15, *i.e.,* 7 in Group 1 and 8 in Group 2), ∼10% in the range 35-44 (12, *i.e.,* 4 in Group 1 and 8 in Group 2), and 12.5% that are 45 years old or more (*i.e.,* 9 in Group 1 and 6 in Group 2). Both groups have similar proportions of participants with a middle school education. However, Group 1 has a slightly higher proportion of participants with a high school diploma (73%) than Group 2 (63.2%), while Group 2 has a higher representation of participants with a Bachelor's degree (28.1%) compared to Group 1 (14.3%). Lastly, a small percentage of participants in both groups hold a master's degree. Figure 5.4 also shows descriptive data on smartphone usage in the general population. Approximately most of those respondents used smartphones for more than eight years.

Regarding the Android versions used, Android 12 or higher versions are used by 26.9% of Group 1 and 28.1% of Group 2, while Android 11 is used by 36.5% and 38.6% of Groups 1 and 2, respectively. Finally, 36.5% of Group 1 and 33.3% of Group 2 use Android versions earlier than 11. In other words, a significant portion of our sample daily use Android versions equipped with Privacy Indicators (Android 12 or later). Concerning privacy, low concern levels were rare in both groups, with just 1.6% of Group 1 and 8.8% of Group 2 selecting the minimum score. Most participants expressed moderate privacy concerns, with the majority in both groups selecting 3 on the scale. Despite these minor variations, both groups shared similar profiles in terms of smartphone knowledge and privacy awareness, though slight differences were apparent at the scale's extremes.

Applying the exclusion criterion previously discussed, we identified and excluded 2 participants from the final analysis. These individuals reported identifying latent resource use in tasks where privacy indicators were invisible by design.

5.3.1 Results Analysis

We analyze the correlation of factors in explicit and latent contexts separately. In Table 5.2 we report the percentage of participants who correctly identified the use of a resource in *latent* and *explicit* tasks, when PI are visible and hidden. Table 5.3 reports the same results divided by the type of resource involved (camera

or microphone). As for the *explicit* context, we observe that the rate of correct identifications of the resource usage is only slightly higher when PI are visible (100 vs. 93). In this case, there is not a statistically significant association between resource identification and the use of PI (p-value = 0.3291). This means that we can not conclude that PI are useful for signaling the resource usage when the usage is explicit. When computing the effect size, we observe that the contribution of PI is *very small* (odds-ratio = 1.449) [116].

Something generally similar happens in the *latent* context, in which the rate of correct identifications of the resource usage is, again, slightly higher when PI are visible (6 vs. 0). For the *latent* context, the observed effect of PI is *marginally* significant (p = 0.0586). The odds-ratio is not a useful measure of effect size in this case: the lack of identification of resource usage when the PI is not visible is 0 by construction since there is no way participants could notice the resource usage, and this leads to infinite odds-ratio even when a single individual correctly identifies the resources. However, only 5% of the participants were able to identify the resource usage in this scenario. This means that the PI would not be effective for the large majority of Android users.

Table 5.3 highlights differences in the usage identification rate for the camera and microphone. To analyze this phenomenon more in-depth, we checked if the type of resource used leads to a significant difference in the identification, regardless of the context or the presence of PI. In this case, we obtained a p-value = 0.0054 using Fisher's test, indicating that the difference is, indeed, significant. In particular, the participants identified more easily the use of the camera than the use of the microphone. This behavior is evident in all tasks, in both latent and explicit contexts, with PI visible and hidden, except for tasks where the behavior was latent, and PI was not visible (where it is impossible to identify a resource). In the explicit context and with PI visible, an 18.3 percentage points difference can be observed between camera identification compared to microphone identification. A higher difference (30.7 percentage points) is noticeable for hidden PI in the explicit context, where the camera usage is identified more frequently. A difference of 2.8 percentage points can be noted in the latent context with visible PI.

Table 5.2: Difference of resource identification between latent and explicit tasks in presence or absence of PI.

% Identified	PI visible	PI not visible
Latent	5%	0%
Explicit	83.3%	77.5%

Table 5.3: Summary table with the number of participants who correctly identified the use of the resource (camera or microphone) reported in the single row and the respective behavior (latent or explicit), when PI were visible and not visible.

Context	Resource	*PI visible*	*PI non visible*	Task
Explicit	***Camera***	53/57 *(92.9%)*	58/63 *(92.1%)*	**2**
	Microphone	47/63 *(74.6%)*	35/57 *(61.4%)*	**3**
Latent	***Camera***	4/63 *(6.3%)*	0/57 *(0%)*	**1**
	Microphone	2/57 *(3.5%)*	0/63 *(0%)*	**4**

Matching Phase

This section presents the analysis results conducted on the responses provided during the *matching phase*, taking into account only those participants who had not identified the resource used during the *response acquisition phase*. In Task 1, only 6 out of 59 participants (that previously had not identified the use of the resource) changed their minds by correctly selecting the screen related to the resource being used, even when the PI was visible, 2 of which are Android 12 users. Concerning Task 2, 2 out of 4 participants correctly selected the screen when the PI was shown. In Task 3, 2 out of 16 participants responded by correctly identifying the screen with the PI, and none of them used Android 12. In Task 4, 4 out of 55 participants correctly selected the screen with the PI, one of them is an Android 12 user. Table 5.4 summarizes the results of this analysis.

In conclusion, concerning the **RQ1** our results suggest that PI play a marginal role in enabling users to identify resource use.

Table 5.4: The number of participants that, during the *response acquisition phase*, did not identify the resource and instead, during the *matching phase*, changed their response correctly.

Tasks	***# identified in the matching phase***
1	6/59
2	2/4
3	2/16
4	4/55

5.3.2 Discussion

The goal of our study is to understand to what extent PI enhances users' awareness of the use of privacy-related resources in different contexts. Accordingly, it was critical to understand if users actually notice that a resource was used because of the PI.

The analysis performed allowed us to answer RQ1. The main goal of PI is to inform the user whenever an application makes use of a runtime permission to access a sensitive resource, like the microphone or the camera. Regarding this, we noticed that the contribution of PI to the identification of resources usage is not statistically significant. As a matter of fact, in spite of an increment in the total number of correctly identified cases, PI do not adequately enhance the users' awareness of when and where an application accesses a specific sensitive resource.

Concerning the tasks in which the access to the sensitive resource is explicit, we can notice a high identification rate independently of the PI. This is likely due to the fact that users have other elements to understand that the resource was accessed (*e.g.,* the camera preview on the screen). Considering the tasks in which camera was used, more than 90% of the users are able to identify when the application accessed the camera just by means of contextual information, *i.e.,* the task they are performing and what the application showed on the screen. It is worth noting that in a few cases (4 out of 57 participants with PI enabled and 5 out of 63 participants without PI enabled) the subjects did not correctly identify the use of the camera, despite the onscreen preview. This is probably due to the fact that, during the experimentation, the preview showed a default animation,

since we used a simulation of the OS instead of the real OS. This could have misled some of the subjects. Overall, the difference between the results in which the PI were visible and those in which they were not visible is negligible (0.8 percentage points improvement on camera).

Considering the task where the application made an explicit use of the microphone, the difference between the two treatments is higher. In fact, when PI were visible 47 out of 63 participants (about 74.6%) were able to identify the fact that the microphone was used, as compared to 35 out of 57 participants (about 61.4%) that noticed the use of microphone when PI were not visible. It is worth noting that, in this scenario, the identification rate is lower in both cases. This is likely due to the fact that the visual feedback of the microphone enabling (during the voice call, the icon to turn the microphone off and on is visible) is less evident than the camera preview (even if the preview did not show the real camera, but a predefined video).

Even if we did not find a statistically significant correlation between the presence or absence of the PI and the identification of the use of the microphone, we noted an improvement of more than 13 percentage points in the identification when PI were visible. Overall, our results suggest that the PI enhance the user awareness of the resource usage, but only very slightly.

To further assess this thesis we discuss what happens when the use of the same kind of resource is made within a latent context, *i.e.,* when there is not a clear reason to access the resource and there is no visual feedback to the user to inform that the app is accessing it, except for the PI, when shown. In particular, we removed any information that could help them to understand the use of the resource, *i.e.,* the camera preview and the behavior that was related to the microphone activation. We expect that this scenario is more adequate to test the effectiveness of the PI, as the participants cannot guess whether the app is using the camera or the microphone by reasoning about the task or noticing the resource activation, but can be informed by the system only by means of the PI. It is worth noting that this is not an unreal scenario as it is likely what happens when the user is victim of a spyware and other type of malware applications. This is exactly where we expect the PI helps the user identify the malicious behaviour.

By construction, none of the participants not having the PI feature enabled was able to notice that the camera and the microphone have been used during the tasks. However, even when PI were visible, only 4 out of 63 participants (6.3%) correctly identified the use of the camera while 59 (93.7%) did not. Similarly, 2 out of 57 participants (3.5%) correctly identified when the microphone was used, while 55 (96.5%) failed. Note that, in this case, the contribution of PI to the awareness of resource usage is even lower than the result achieved in the explicit context. This is counter-intuitive, as we expected at least the same improvement.

To conclude, despite an improvement can be noted, it is clear that PI, in their current implementation, are still unsatisfactory as a solution to make users aware of the resources accessed during the use of Android apps.

It is worth noting that most of the participants do not use Android 12 or later as their primary mobile OS, and that such a feature probably requires time to be effectively mastered by the users. Thus, we have conducted a further analysis restricting the observations to Android 12 or later users only, to confirm or contradict our findings. Furthermore, we are interested only in tasks where the context was latent, since we noted that when the context is explicit users are able to identify the use of sensitive resources in any case. Among the 33 participants that claimed to use regularly Android 12 and later, 17 were involved in Task 1 and 16 in Task 4. Only 3 of the participants involved in Task 1 and 1 participant involved in Task 4 noticed that the camera and the microphone have been used during the tasks. While this result is slightly better, the message is the same: the large majority of participants (87%) fail to identify the use of sensitive resources.

It is possible to observe that the results are not more encouraging after the *matching phase*. During this phase, the user is not distracted by the execution of the task and statically observes screens related to already executed tasks. Note that the screens show the PI when enabled. Concerning the tasks where the use of resources was explicit, the improvement for the camera task was 3.5 percentage points (2 users of the 4 who initially did not identify the camera noticed the PI), while for the microphone task the improvement was 3.2 percentage points (2 users out of 16). Note that the results were quite good for explicit contexts, thus we expected a limited improvement during the *matching phase*. As expected, the most noticeable improvement was in the tasks making a latent use of resources. In

particular, concerning the camera task it is possible to observe an improvement of 9.5 percentage points (6 users out of 59 who initially did not identify the camera), while in the microphone task the improvement was 7 percentage points (4 users out of 55). This suggests that when the PI is shown statically, participants are substantially more able to identify the resource usage. Unfortunately, when the use of a resource is latent, the user awareness is still very limited. In fact, when the PI was visible, 53 users did not notice that the camera was used during the task (51 for the microphone).

We deliberately concentrated on evaluating the use of the camera and microphone, as these are currently the only resources for which Android PI are used. By doing so, we ensured that our experiment design mirrored real-world scenarios, allowing us to assess the effectiveness of PI under conditions that Android users regularly encounter. Our results in a latent scenario, where the resource use isn't immediately apparent, could potentially apply to other resources that don't provide clear visual feedback when accessed, such as contacts or data storage. However, we consider this a preliminary interpretation and suggest that further research is necessary to conclusively extend our findings to these resources.

Moreover, we conducted an analysis to investigate how various factors, such as perception of privacy, familiarity with smartphone operating systems, years of smartphone usage, level of education, and background in computer science, may impact the ability to identify the PI in tasks involving latent resource usage, where PI is the only means for participants to determine resource usage.

To achieve this goal, we used logistic regression analysis. Our independent variables included demographics such as privacy perception level, knowledge of smartphone operating systems, years of smartphone usage, level of education, and having a computer science background. The dependent variable was the identification of resource usage.

The logistic regression analysis indicated that none of these variables significantly affected the probability of participants noticing the PI. Specifically, privacy perception level ($p=0.569$), knowledge of smartphone operating systems ($p=0.513$), years of smartphone usage ($p=0.747$), level of education ($p=0.782$), and having a computer science background ($p=0.975$) all had p-values exceeding

the 5% significance level. These results suggest that the null hypothesis, stating these variables do not influence the outcome, cannot be rejected.

For example, our results suggest that subjects with a high level of education are not more likely to notice the PI than subjects with a lower level of education. Similarly, having a background in Computer Science did not increase the chances of noticing the PI compared to not having a computer science background.

Interestingly, the only statistically significant factor in our model was the constant (p=0.022), implying a significant effect on the likelihood of responding "Identified" or "Not identified" to questions about resource usage in the task that wasn't explained by the variables in our model. Furthermore, the model's pseudo R-square of 0.03125 indicates our chosen variables explain only a small fraction of the variability in PI detection. This absence of significant predictors suggests that PI awareness, and thereby the latent behavior associated with the usage of the resource it represents, might be intrinsic to how it is implemented.

The analysis results are summarized in Table 5.5.

Table 5.5: Logistic Regression Results

Variable	**Coefficient**	**Std. Error**	**z**	**p-value**
Constant	-5.0907	2.216	-2.297	0.022
Privacy Perception Level	0.2692	0.473	0.569	0.569
Knowledge of Smartphone OS	0.3357	0.513	0.654	0.513
Years of Smartphone Usage	0.0393	0.122	0.322	0.747
Level of Education	-0.1368	0.495	-0.277	0.782
Having a Computer Science Background	0.0287	0.935	0.031	0.975

5.4 Evaluating an alternative solution RQ_2

There are many reasons why PI may not be effective in supporting awareness of the use of privacy-related resources. Two of the contributing factors could be the visibility of the indicators and the association between PI and the fact that a sensitive resource is used. In order to evaluate whether this conjecture is correct and the user awareness can be enhanced, we introduce an alternative version of PI that is clearly more visible and that highlighted the resource accessed by the app. Our solution (named POPUP) is a non-blocking (allows the user to

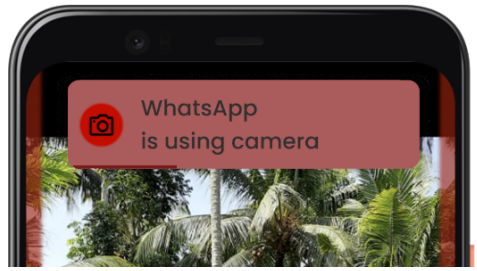


Figure 5.5: An example of our enhanced privacy indicator.

continue using the application, rather than blocking its execution) and informative notification implemented through a toast component similar to push notifications. This component gradually becomes transparent over 5 seconds until it disappears completely, initially covering part of the screen but later allowing the user to see more of what is underneath.

The POPUP has three visual enhancements with respect to standard PI (see Figure 5.5):

- **Icon**: a red icon representing the resource accessed (e.g., a camera or microphone icon).
- **Description**: a textual information indicating the application that is accessing the resource.
- **Animation**: the POPUP includes a red progress bar that represents how long the textual notification will be totally visible until it disappears. Moreover, POPUP includes two red bands appearing on the screen's sides. These bands alternate between widening and narrowing, providing a peripheral alert that may capture users' attention more effectively even when the textual notification is hidden.

The POPUP was designed to have a limited impact on the user experience with the application. Indeed, it is non-blocking (similar to the default Android implementation of PI) and disappears automatically after 5 seconds so that the user can ignore it. Nonetheless, it provides enough cues to make the user aware of the resource accessed by the app.

5.4.1 Empirical Study Design

The goal was to understand if our solution is more effective than the default one provided, at the moment, in Android. Such a study is guided by the following research question:

RQ2 *Does an enhanced indicator improve user ability to identifying when a resource is used by a mobile application compared to default Android PI?*

We entirely re-used the design of the previous experiment (*e.g.,* the apps and the tasks). In this case, we involved **38 Android users**. Some of them participated in the first experiment and expressed an interest in being involved again in this study, while others were volunteers from a public event. The two treatments, in this case, were (i) default PI, and (ii) POPUP PI. Since all the participants were exposed to both default PI and POPUP PI (in different tasks), we needed to re-think the instructional tutorial presented to the participants. In the previous experiment, it only presented default PI, while in this case, we wanted to present our new solution as well. Thus, we presented both of them, explicitly reporting that they could encounter either and that the meaning was exactly the same.

As for data analysis, we employed a Fisher's exact test, with the null hypothesis (H_0) that *"Enhanced indicators do not significantly improve user ability to identifying when a resource is used by a mobile application compared to default Android PI,"* and the alternative hypothesis (H_1) that *"Enhanced indicators significantly improve user ability to identifying when a resource is used by a mobile application compared to standard PI."*

5.4.2 Empirical Study Results

Figure 5.6 visualizes the demographic distribution of the participants, while we present in Table 5.6 the results of our experiment.

For the *explicit* context, the obtained p-value is **0.0125**, a value considerably below the 0.05 threshold. This suggests a significant correlation between the use of enhanced indicators and higher user awareness of resource usage. Note that, in this case, statistical significance has been achieved with a much smaller number of participants, thus suggesting a larger effect of the treatment.

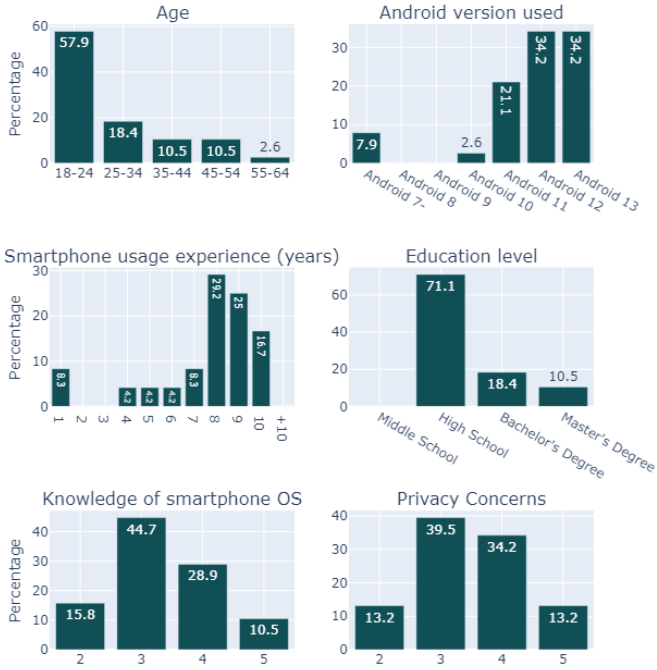


Figure 5.6: Demographic users information in our solution

Indeed, the odds-ratio is 7.194 (*i.e., large* [116]), suggesting that users are much more likely to recognize the resource usage when enhanced indicators are used compared to default Android PI. This result is even more clear in the *latent* context, for which the p-value is extremely low (**1.939e-07**) and the odds-ratio, again, *large* (20).

These results clearly show that a more prominent indication of resource usage significantly helps users.

Table 5.6: Summary of the experiment results

Context	Resource	*PI visible*	*POPUP*	Task
Explicit	***Camera***	15/19 *(78.95%)*	19/19 *(100%)*	**2**
	Microphone	13/19 *(68.42%)*	17/19 *(89.47%)*	**3**
Latent	***Camera***	0/19 *(0%)*	16/19 *(84.21%)*	**1**
	Microphone	4/19 *(21.05%)*	11/19 *(58.89%)*	**4**

5.5 Threats to Validity

Threats to **internal validity** concern confounding factors that could influence the results. Participants were randomly assigned to one of the two groups, countering selection bias by making groups comparable at the start of the study. We assigned each participant 4 tasks; it is reasonable to think they get used to complete them by noticing how previous tasks work, increasing the learning effect. We changed the treatment between tasks assigned to the participants (based on the variables reported in Section 6.2.2) to reduce the learning effect. To reduce the risk of possible random responses from participants we included control questions to sustain participant focus and disqualified individuals who inaccurately identified latent resource usage when PI were not visible. Only two participants out of the initial 122, less than 2%, were excluded based on this criterion. This minimal exclusion rate does not significantly compromise the overall validity and reliability of our experiment. The total percentage of correct responses to control questions across all participants turned out to be **94.17%**. This high percentage suggests that participants generally demonstrated a strong understanding of the tasks and were sufficiently focused.

One of the possible risks is that the results depend on the specific task in two groups and they may not be balanced since it is possible to observe differences in the results of individual tasks. For that reason, we performed an analysis with Fisher's test based on each task within both groups. Results ruled out this possibility for all tasks ($p_{Task1} = 0.30$, $p_{Task2} = 1$, $p_{Task3} = 0.30$, $p_{Task4} = 0.30$).

In Section 5.3.2 we discuss about the improvement given by PI to participants' awareness of resource usage between latent and explicit tasks, noting that the improvement is lower in latent tasks than in explicit tasks. This result could

be counter-intuitive and could indicate that the aforementioned results are not strictly related to PI. We mitigated this thread through the matching phase where it is possible to observe that the results do not improve significantly.

To detect errors and improve the experimental design, we conducted a pilot study with 27 participants. To avoid as many problems as possible related to social interaction, we have selected pilot study participants outside the university setting to minimize possibles contacts between them and actual study participants. In addition, to limit interactions among participants within the university, during the final study participants were grouped in sessions and monitored by university staff. Moreover, we prevented, as much as possible, any contact between participants from different sessions.

Another threat to validity is that 72.3% of users do not run Android 12 or later versions on their own devices, so they may not be aware of PI. For this reason, at the beginning of the experiment, we explained the new privacy-related features introduced in Android 12, with particular reference to PI. In addition, we created a prototype where participants were shown a tour of the new features, interactively demonstrating how PI works. The tour was mandatory and was rerun in case the participants did not declare they fully understood the new feature introduced in Android 12. However, the risk that more experienced users on the latest version of Android would obtain different results exists. To this aim, we have conducted a further analysis restricting the study to regular Android 12 and later users only. Even in this case, 87% of them failed to identify the use of sensitive resources in latent contexts. It is important to note that this is a qualitative analysis conducted on a limited sample of the participants, so it is recommended that a replication of this study will be conducted in the future involving subjects with a higher experience on Android 12 and later to confirm or contradict our findings.

We decided to limit the interactions that users can do on the prototypes; in this way, it is tough to get stuck during the steps to be performed, informing participants during the tutorial that by doing an action that is not allowed, the prototype will highlight with a blue container any of the allowed interaction. Despite this is not the actual usage scenario, we believe that the restricted

interaction do not significantly impact on the user's understanding of whether the app uses the camera or the microphone or not.

We did not explicitly disclose the goal of the study to the participants to avoid creating bias, but the purpose could be evident to our participants at various moments, which may have influenced their conduct. For example, before each task, we notified the user that specific task represented a smartphone running Android 12 and we clarified that all permissions were accepted by default and would not be asked again. In this way, we might have suggested that our goal was related to resource usage, but this was necessary because the PI is only available since Android 12. Moreover, we wanted to make participants aware of the fact that no popup request for permission would be presented during the tasks. Note that making any request explicit would have biased the results of the study. In fact, within a task having a latent access to the microphone, the presence of the popup asking for permission to use it would have made the task explicit, making the PI useless. We wanted to focus the participants' attention only on the PI, avoiding other identifiers of resource use; moreover, the request to use a permission can occur only once during app use (unless the user agrees only for this one time), so we did not affect normal behavior.

Threats to **external validity** concern the generalization of the experiment and our findings. Our sample has the risk of not being representative of the entire population of Android users, as the study was conducted in a University setting. However, the participants have different characteristics from each other, as identified during the demographic questionnaire.

There are characteristics of our study design, like any empirical investigation, that may restrict the generalizability of the results. Our focus on popular and trusted apps assumes that users are more likely to grant them full access to sensitive resources. However, this might not hold true for lesser-known or non-trusted apps, where users may be more cautious. Therefore, our findings might not generalize effectively to these scenarios, warranting further research.

We performed our experiment in a highly standardized and somewhat artificial environment, and our findings could not reflect user behavior in actual apps usage. We followed recommendations for experimental studies proposed by Aguinis in [117] to reduce this problem. For example, our participants were given a thorough

explanation (both orally and in writing) of what they were supposed to do. The individual goals to be achieved in the tasks were clearly explained. In addition, they were given a tutorial on how the prototypes to use work. All participants were able to complete the tutorial, obtaining the check-code to enter in order to proceed (as in the individual tasks) without any problems. These measures, combined with the use of their own smartphone and realistic prototypes experiencing interface and action to perform identically to real apps, ensure a high level of immersion for the participants, which, as previous work has shown, leads to the highest possible generalizability of the study results [117] [118] [119].

5.6 Final Remarks

In this work, we conducted a preliminary study on the effectiveness of the PI feature released in Android since version 12 to inform the user whenever an app accesses the camera or the microphone of the mobile device.

We conducted a controlled experiment involving 122 participants, who were asked to complete a series of tasks on a series of prototypes that mimic Android 12 with the PI feature enabled and disabled. Some of the tasks explicitly made use of the camera or microphone, while other tasks used them in a latent manner. We wanted to assess how effective are PI to help users identify the use of a resource (camera or microphone) in different scenarios.

Our results show no significant correlation between the presence or absence of the PI and the detection of the resource used, suggesting that most Android users do not adequately notice their presence. Interestingly, our results indicate that the ability to identify the PI in tasks involving latent resource usage appears to be independent of factors like privacy perception level, knowledge of smartphone operating systems, years of smartphone usage, level of education, and having a Computer Science background. This conclusion is based on the p-values obtained from the logistic regression analysis explained at the end of Section 5.3.2, all of which exceeded the 5% significance level. However, further research with control groups is needed to support these findings conclusively.

We, therefore, evaluated an alternative solution focused on two of the aspects that we believe could contribute to the reduced effectiveness of PI, namely their

visibility and their ability to adequately inform the user that the app has accessed a specific resource. The results of this second study demonstrated that, by acting on the way PI are implemented, a significant improvement in their effectiveness can be achieved.

Our work was focused only on two specific privacy-related resources, although we can imagine that Android will use a similar approach for other runtime permissions in the future. We suppose that it will be even less effective to extend the approach to include other runtime permissions, (*e.g.,* ACCESS_FINE_LOCATION, INTERNET_CONNECTION, WRITE/READ_EXTERNAL_STORAGE) since these are even less explicit in their use. Moreover, the risk exists that the user could be further confused by several indicators repeatedly shown in the toolbar and would barely remember their meaning or even stop paying enough attention to the LED displayed on top of the screen. On the other hand, focusing only on these two resources is limiting since, although they are among the most sensitive permissions, many other threats to the privacy are related to different runtime permissions. Therefore, part of our future agenda will include more in-depth studies to understand whether users tend to identify access to other sensitive resources and whether we can further generalize the obtained results.

5.6.1 Data availability

The data that support the findings of this study are openly available at `https://dibt.unimol.it/report/privacy-indicators/`, derived from users' responses to questionnaires available at `https://dibt.unimol.it/report/privacy-indicators/#survey`. Simulations of the four tasks can be observed at `https://dibt.unimol.it/report/privacy-indicators/#tasks`, with explanation of their behavior and visible PI. In addition, we have also made available prototypes that can be used directly on personal smartphones; they can be accessed by scanning the QR codes at `https://dibt.unimol.it/report/privacy-indicators/#prototypes`.

CHAPTER 6

Visual Attention and Privacy Indicators in Android: Insights from Eye Tracking

Contents

6.1 Introduction 83
6.2 Design of the Study 85
6.2.1 Context Selection 86
6.2.2 Experimental Procedure 87
6.2.3 Pilot Study 93
6.3 Technical Implementation of the Used System . . . 93
6.3.1 Integration of Eye-Tracking with WebGazer.js 93
6.3.2 Privacy Indicator Detection Mechanism 94
6.4 Results 95
6.4.1 Discussion 98
6.5 Final Remarks 99
6.5.1 Data availability 100

Building on our initial findings in Chapter 5, we have started using eye-tracking technology to investigate the complex relationship between user attention and the effectiveness of PIs. This method allows us to observe how users react to PIs in real-time and obtain a more detailed perspective on their interaction with these features. Our objective was to bridge the gap between the empirical data we gathered and the nuanced understanding of how users perceive and respond to PIs. By integrating eye-tracking, we aimed to uncover the subtle behaviors that indicate user awareness and response to PIs, providing a more comprehensive picture of their impact on privacy awareness. This study is not only an extension but a crucial enhancement of our understanding of PIs. It enables us to pinpoint the exact moments when users notice, or fail to notice, these indicators. This approach provides empirical evidence of the PIs' visibility and effectiveness in real-world scenarios, moving beyond self-reported data and assumptions. Through this detailed investigation, we validated our initial findings and gained new insights on how to enhance the design and implementation of Privacy Indicators. Our goal is to improve Android's privacy features to meet the nuanced needs and behaviors of its extensive user base, ensuring that these indicators are not just present but are genuinely effective in safeguarding user privacy.

6.1 Introduction

Android, the undisputed leader in the mobile device world, has become an integral part of our everyday lives, with its ubiquitous operating system deeply entrenched into the fabric of our digital experiences. The platform's widespread adoption brings to the fore critical questions about user privacy and the efficacy of mechanisms designed to protect it. Android's permission model, a framework established to regulate app access to sensitive resources like cameras and microphones, is central to this discussion. However, users' growing trust in applications increasingly challenges this system's effectiveness, often leading to a casual approach to granting permissions.

The inherent complexities of Android's permission model and user behavior underscore the need for robust privacy safeguards. This is where Privacy Indicators

(PIs) come into play. Implemented as visual cues within the operating system, PIs aim to alert users when sensitive resources such as camera and microphone are accessed (Figure 6.1). Yet, questions linger about their actual impact and visibility in everyday use, especially when permissions are exploited in non-contextualized or malicious ways.

PIs became a part of the Android operating system starting from Android 12. They are a critical update in Android's ongoing evolution to enhance user awareness and control over privacy. This study seeks to evaluate the practical impact of this significant feature in real-world scenarios, providing insights into its effectiveness and areas for potential improvement.

Our study seeks to illuminate this crucial aspect by employing eye-tracking technology, a method traditionally reserved for fields such as marketing and medical research, now repurposed to scrutinize user interactions with PIs. This innovative approach allows for an objective, real-time analysis of how users notice and process PIs amidst various interactive tasks on their smartphones.

In this context, we conducted a controlled experiment involving 29 participants to empirically investigate user interaction with Privacy Indicators on Android devices. These participants, drawn from diverse backgrounds, interacted with a series of tasks designed to simulate real-world smartphone usage. Through eye-tracking, our objective was to monitor how and when these users noticed the Privacy Indicators during different types of interactions, especially in scenarios where resource use might not be anticipated.

Through a comprehensive experimental setup, we aim to probe the depths of user attention and response to PIs, assessing their prominence and effectiveness in real-world scenarios. This investigation is particularly timely given the increasing sophistication of apps in seeking permissions under benign guises, only to exploit them for less scrupulous purposes, as explored in the research by [97].

Our experiment revealed critical insights into user engagement with PIs. The findings suggest that the classic PIs in their current form are often unnoticed, especially in high-engagement tasks, highlighting a potential vulnerability in privacy protection. This underscores the need for more intuitive and noticeable privacy indicators that can effectively alert users to resource access in real-time.



Figure 6.1: An example of Privacy Indicator in action on Android 12.

Our research endeavors to contribute significantly to the discourse on mobile privacy by bridging the gap between technological capability and user behavior. The insights gleaned from this study are expected to inform the design of more intuitive and effective privacy features, steering the mobile computing landscape toward a safer, more privacy-conscious future.

6.2 Design of the Study

The primary *aim* of our study is to understand user perception of Privacy Indicators (PIs) using eye tracking, given their critical role as defined by Google in signaling live resource use. Specifically, we focus on how users detect and react to PIs that denote unauthorized access to sensitive resources such as cameras or microphones by potentially malicious applications. Additionally, the study explores user responsiveness to two innovative PI designs across various interactive contexts. As the sole alert mechanism for sensitive resource usage, these indicators

present a unique opportunity to measure their conspicuousness and the immediacy of user awareness facilitated by them. The research was driven by the following question:

RQ1: *Do the app characteristics influence the effectiveness of Privacy Indicators (PIs)?*

A controlled experiment involving human subjects was designed to address this RQ_1.

6.2.1 Context Selection

The context for our study was crafted to mirror real-world usage where Privacy Indicators (PIs) play a pivotal role in alerting users to the live use of sensitive device resources like cameras or microphones. Unlike simulations of app environments, our experiment utilizes a web application that integrates eye-tracking technology to create a controlled setting that closely replicates user interactions with actual device notifications. The subjects comprised a group of 29 university students, selected for their commonality with the general Android user population. The web application was designed as the interactive 'object' of the study, in an Android environment where classic and alternative PIs were triggered unexpectedly. This consisted of two main categories of tasks: low spatio-temporal attention tasks (quizzes) and high spatio-temporal attention tasks (games). Including these diverse tasks allowed us to evaluate the PIs effectiveness across a spectrum of user engagement levels, providing insights into how attention to PIs might vary with user cognitive load. Unlike previous studies that simulated popular applications, we designed a singular application tailored to the experiment's requirements, ensuring a consistent platform for all participants. Before initiating the experimental tasks, participants provided detailed information about their personal devices, including vendor, model, and operating system, to ensure comprehensive data relevance and to prepare the experimental setup adequately. We excluded devices solely running Apple's iOS to maintain focus on Android's user interface, which could influence the study's outcomes due to cross-platform differences. Within our application, we

designed the display of *Classic* privacy indicators, as presently incorporated in the latest Android versions, to trigger without user anticipation. This setup allowed us to explore user perceptions in a setting that mimicked potential malicious application behavior. To better understand if implementing more visible PIs can affect the achieved results, we also introduced two innovative PI designs—*edge* and *disk*—each exhibited twice to the user under various task conditions.

6.2.2 Experimental Procedure

We explored three distinct PIs (Figure 6.2), each with unique visual characteristics meant to signal the user about the real-time usage of sensitive resources. Here we detail these indicators, which were the focal point of our investigation, describing their design rationale and expected impact on user experience and privacy awareness:

- **Classic PI:** This is the current implementation of PI in the latest versions of Android. This indicator typically appears as a small icon at the top right edge of the screen, generally within the status bar, indicating the use of either the camera or microphone. In our experiment, this PI was designed to emulate this typical behavior closely. It appeared as a predefined icon – a simple camera or microphone symbol – to signal the operation of these resources. The icon was intended to be discreet yet noticeable enough to inform the users without causing significant distraction from their primary tasks on the device.

- **Disk PI:** This is conceptualized as a dynamic visual cue to enhance the visibility of the resource usage notification. It consists of an outer ring that maintains a constant size while the inner circle size changes, creating a pulsating effect. The color of the disk alternates between green and yellow, providing an additional visual signal of the camera or microphone in use. This design intended to make the classic indicator more visible without radically altering its form or position, presenting a potentially more eye-catching and, thus, effective notification method.

- **Edge PI:** This is an innovative alternative inspired by the edge lighting feature found in certain Samsung devices. It is represented as a small, colored slider that animates horizontally across the top edge of the screen, from left to right, and fades out as it moves along its path. The animation includes a fading trail to subtly draw the user's attention, aiming to determine whether motion and transient visual changes could alert users more effectively. The choice to confine the edge effect to the top of the screen was a deliberate design constraint, considering the limitations of measuring visibility for indicators that occupy the full screen on a mobile device.

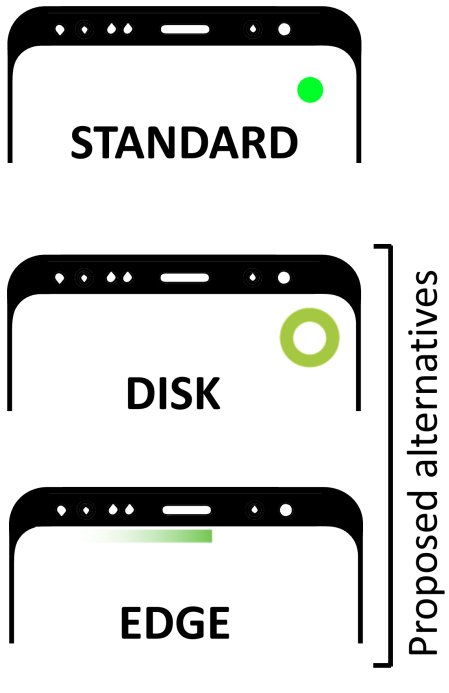


Figure 6.2: Comparative Display of Classic Privacy Indicators and Proposed Alternatives

Our study's experimental procedure was a structured investigation into how users perceive and interact with Privacy Indicators (PIs) while engaging in distinct tasks within a specialized web application enriched with eye-tracking technology.



Figure 6.3: Sequential Diagram of the Controlled Experiment Process

This bespoke platform was designed to record precise user reactions to classic and alternative PIs under conditions that closely mimic real-world application usage.

The study was conducted in a dedicated university classroom designed to minimize distractions and optimize eye-tracking accuracy. Specific measures, such as intense uniform lighting and a monochromatic backdrop, were taken to ensure high-quality data capture and minimize potential error sources. The participants were instructed on maintaining a stable posture and were briefed about the calibration process and the tasks they would undertake during the experiment. The indicators were displayed briefly at the top of the screen while users were engaged in the designed activities, capturing their attention and response patterns using eye-tracking metrics.

At the beginning, participants were familiarized with the experimental protocol. The experimental procedure was delineated as follows (Figure 6.3):

- **Initiation:** users start with reading the instructions and granting necessary permissions.
- **Calibration:** The task proceeds with the calibration interface, engaging with the system to train the eye tracker.
- **Task Engagement:** Users perform the assigned tasks, during which PIs can be presented.
- **Interaction Logging:** All user interactions with the tasks and PIs are logged alongside eye-tracking data.

- **Data Submission:** Upon completing each task, data are submitted to the server.

After giving participants the instructions and ensuring their consciousness about the experiment, they started the calibration phase to adapt the eye-tracking system to their individual visual and interactive patterns. Each participant was asked to follow and touch specific points on their screen, which were highlighted to attract focus (Figure 6.4). The calibration phase was a recurring step in the experiment, serving as a gateway between tasks to ensure the continued accuracy of the eye-tracking data. The calibration process was meticulously designed to require a minimum of 70% accuracy. This threshold was established as an optimal balance between precision and usability. Indeed, posing a higher threshold would result in frequent recalibrations, disrupting the experiment flow, while lower values would compromise data integrity. In case participants failed to achieve 70% accuracy, the system prompted a recalibration until the required precision level was reached. After the initial calibration phase, continuous drift correction is applied throughout task execution, refining the calibration with each interface touch to progressively enhance the eye-tracking data accuracy.

After the calibration phase, participants engaged in a series of six tasks, alternating between quizzes and games to challenge their attention and cognitive load (Figure 6.5). These tasks were intended to explore the effectiveness of PI notifications within apps that significantly differ in terms of user engagement: low spatio-temporal attention during quizzes and high spatio-temporal attention during interactive gaming scenarios. The sequence of tasks was carefully orchestrated as follows:

Q_1 [**Classic PI**] Participants began with a quiz designed to assess their focus on content and subsequent noticeability of the *Classic* PI.

G_1 [**Edge PI**] The next task was an interactive game intended to examine how the alternative *Edge* PI performed in an environment requiring high user interaction.

Q_2 [**Disk PI**] Another quiz session followed, this time using the *Disk* PI, to test its visibility in a low spatio-temporal attention context.

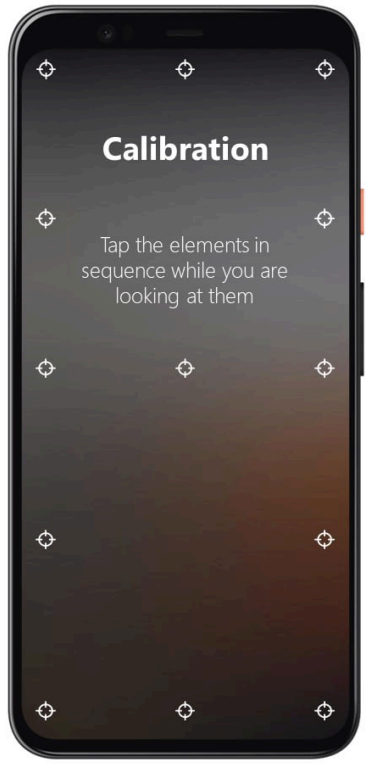


Figure 6.4: Calibration process screen displaying markers for eye-tracking accuracy improvement

G_2 [**Classic PI**] The sequence returned to a gaming task with the *Classic* PI to evaluate consistency across task types.

Q_3 [**Edge PI**] A repeat of the quiz format provided another data point for the *Edge* PI, reassessing its effectiveness.

G_3 [**Disk PI**] Finally, the *Disk* PI was revisited during a gaming task to complete the set of experimental conditions.

To simulate a realistic usage scenario, PIs and their alternatives were displayed for exactly eight seconds, randomly materializing within a three-second interval following the onset of a corresponding task. This randomness introduced an element of unpredictability, mimicking real-world notifications that often occur without prior user expectation.

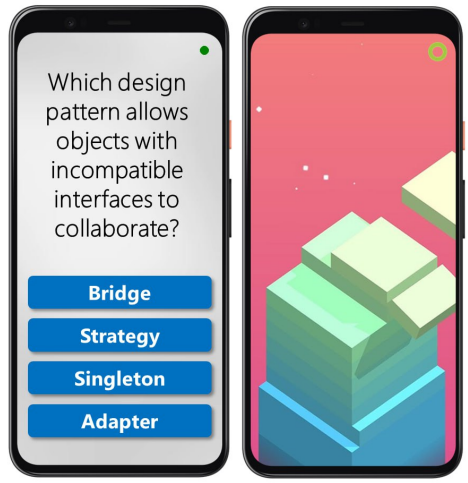


Figure 6.5: Two of the experiment tasks requiring low spatio-temporal attention (*Quiz*) on the left and high spatio-temporal attention (*Game*) on the right

User actions and eye-tracking data throughout the tasks were logged to a remote server in real-time. This logging included timestamps of each PI display, gaze fixation, duration, and the user's interactions with the task at hand. Such data are collected and subsequently analyzed to understand whether the PIs

were noticed and how the interaction context and cognitive load influence their visibility and the user's attention engagement.

6.2.3 Pilot Study

Before running our experiment, we conducted a pilot study with 8 participants in an offline setting. Such a study was promoted from March 22, 2023, to April 13, 2023. We tested the previously described experimental protocol, and we checked whether the tasks were feasible, whether the instructions were sufficient and explicit, whether the web app worked on different devices and their usability, and whether users experienced problems or had doubts during execution. None of the users reported problems in understanding and completing the tasks.

6.3 Technical Implementation of the Used System

In this section, we describe the development of a web application purpose-built for conducting an experimental evaluation of Privacy Indicators (PIs) and two novel alternatives. The application leveraged the Angular framework for front-end user interaction and WebGazer.js for eye-tracking capabilities. The web app served as a testbed to assess the current PI design's effectiveness and gauge user response to newly proposed PI designs under real-world conditions. Angular was chosen for its scalability and robustness, ideal for creating a dynamic web application capable of supporting the complex functionalities required by the experiment. The application was structured to handle various tasks and conditions systematically, guiding participants through the experiment's flow while capturing their interaction data with high fidelity. The web application's design facilitated a responsive and intuitive user experience, essential for maintaining participant engagement throughout the experiment.

6.3.1 Integration of Eye-Tracking with WebGazer.js

WebGazer.js [120] provides a versatile eye-tracking solution adaptable to different experimental scenarios. The library was integrated into the Angular application to enable real-time gaze tracking using participants' webcams, with a

specific focus on the accuracy and responsiveness necessary for tracking quick eye movements.

Calibration process

The eye-tracking calibration process was a multi-step procedure designed to adapt to the user's unique physiology and environmental conditions. Calibration accuracy was enhanced through iterative user engagement with on-screen elements, which refined the predictive model of WebGazer.js in real-time. The web application was designed to provide visual cues that guide users through calibration without overwhelming or distracting from the core tasks. We used an improved version of Webgazer.js that was calibrated continuously throughout the session. This ongoing calibration, known as drift correction, is dynamically adjusted based on user interactions, specifically their touch inputs. Authors in [121] paper provided a foundational basis for understanding how gaze behavior can be effectively tracked and adjusted in real-time, enhancing the accuracy and reliability of eye-tracking data, especially in dynamic user interaction scenarios. This continuous calibration method is crucial for maintaining the precision of gaze tracking throughout the session, ensuring that the data collected reflects the actual user behavior and interactions. This was achieved through minimalistic design choices and using Angular's dynamic rendering capabilities to update the interface based on the progress of the calibration process.

6.3.2 Privacy Indicator Detection Mechanism

The 'alert zone' setup involved defining the perimeter within the application view where user attention was considered focused on the PI. This logic was encapsulated within the *setPrivacyIndicator()* function, which dynamically adjusted the sensitivity based on the screen's real estate and the user's distance from the device. A dedicated Angular service was implemented to handle the real-time streaming of gaze data provided by WebGazer.js. The service utilized an event-driven model to capture and process gaze coordinates at pre-defined intervals, ensuring the system's responsiveness and the accuracy of the collected data. All gaze data associated with PI detection were timestamped and recorded within the user's

session without storing any video feed. This approach was designed to respect user privacy and comply with ethical standards for research involving human subjects.

6.4 Results

We analyzed the eye-tracking data collected during the experiment to investigate the effectiveness of different PIs in attracting users' focus during tasks requiring different levels of attention. We introduced two independent variables: the animation extent of the PI (*Classic*, *Edge*, and *Disk*) and the level of concentration required for the task (*Quiz* and *Game*). The dependent variable concerns the eye movement's location and speed as a proxy of attention and awareness of the PI. Specifically, we determined the amount of time spent looking at the PI by measuring the total duration of fixations within a 250-pixel radius around it. We also defined unintended or false fixations as a total cumulative fixation duration of less than 150 ms and excluded them from subsequent analyses. These criteria were adopted to account for the noise and variability introduced by our smartphone-based eye-tracking approach. Out of 174 observations, 3 were discarded based on these criteria, while a further 37 were excluded because the participant completed the task too quickly to achieve the required level of attention, and often before the PI was displayed.

To answer our RQ, we conducted a two-way mixed-effects ANOVA, which tests for the presence of a statistically significant effect of the independent variables, singly or in interaction, on the dependent variable. Given a significant effect, we performed post-hoc comparisons to understand which specific type (or interaction of types) of task and/or PI successfully affected PI awareness. Specifically, we used Bonferroni-corrected unpaired t-tests for pairwise comparisons between different conditions. Importantly, the advantage of this mixed model over alternative solutions is that it takes into account the effect of between-participant variability. Furthermore, the interaction between PI and task allows us to investigate whether a PI might have a specific effect only on one particular task condition and not on the other. To sum up, the null hypothesis (H_0) is that there is no association between the type of task and PI and the awareness of the PI. Conversely, the alternative

hypothesis (H_1) is that the presence of PI is noticed differently according to its visual properties and the characteristics of the attentional context. We reject the null hypothesis if the p-value is lower than or equal to 0.05.

In Table 6.1 and Figure 6.6 we report the average time spent looking at the PI in the different conditions. These data show a strong effect of task type on PI awareness. Almost all participants didn't fixate on the PI when the task required a strong attentional focus on a different screen location (i.e., the *Game* condition). This effect appears to be irrespective of PI animation, although the *Disk* was marginally more effective than the others. Since PIs were overlooked in the *Game* condition, the effect of the different animations can be appreciated only in the *Quiz* condition. *Disk* fixations lasted longer than those of the *Classic* PI. Interestingly, the *Edge* PI was almost not noticed in this condition as well. We attribute this result to the smaller size of the stimulus and its location on the outer periphery of the screen rather than to its animation type.

		Indicator		
		Classic	*Edge*	*Disk*
Task	*Quiz*	965.81 ms	11.08 ms	2114.40 ms
	Game	0.0 ms	0.0 ms	224.50 ms

Table 6.1: Average time spent looking at the PI in the different conditions.

The statistical findings are presented in Table 2. The ANOVA corroborated our observations, as we identified significant ($p < 0.05$) main effects of task and PI type on the interaction. This outcome indicates that our manipulations were effective in influencing PI awareness. Given the lack of fixations in the *Game* setting, post-doc comparisons focused on the *Quiz* one. Unsurprisingly, all PIs had a significant effect, with the *Disk* being the most effective, and the *Edge* being the least effective.

Overall, these results indicate that the PIs were noticed almost exclusively in the *Classic* and *Disk* form during the *Quiz* task. To further investigate the attentional dynamics of these conditions, we measured the time elapsed between the onset of the PI and the first fixation within the PI area. These saccadic reaction times were shorter for the *Disk* PI (mean = 1799; std = 381) compared

to the *Classic* one (mean = 2244; std = 513), suggesting that our animation provides a mean for a faster attentional capture. However, an unpaired t-test revealed no significant difference between the reaction times to the two PIs.

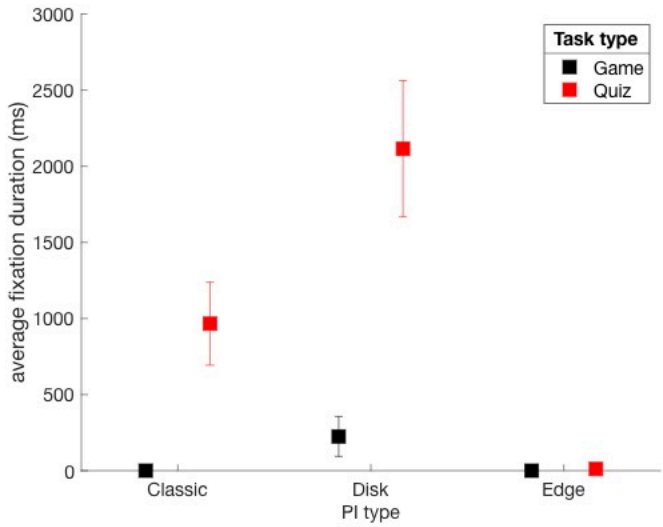


Figure 6.6: Average fixation durations on Privacy Indicators (*Classic*, *Edge*, *Disk*) during low (*Quiz*) and high (*Game*) spatio-temporal attention tasks, highlighting the differential impact of task type on user's attention to PIs.

Source	*Sum Sq.*	*d.f.*	*Mean Sq.*	*F*	*Prob>F*
Type	30163156	2	15081578	9.939948	9.71E-05
Task	29526910	1	29526910	19.46056	2.16E-05
Type:Task	19506429	2	9753215	6.428137	0.002187
Error	1.79E-07	0	0		
Total	2.84E+08	133			

Table 6.2: Statistical results

6.4.1 Discussion

In this section, we discuss the eye-tracking experiment's outcomes regarding the effectiveness of PIs in different task contexts.

The data collected indicate that the cognitive load required by a specific task notably impacts the likelihood of the user detecting a PI. This was particularly evident in the *Game* setting, which is characterized by a high degree of spatio-temporal focus, and where both Android's native PI, as well as alternative animations, were consistently overlooked. These findings suggest inattentional blindness, where users caught up in demanding tasks may fail to notice secondary stimuli, such as the PIs in this study. Consequently, a key design goal is to develop PIs that can successfully capture the user's attention without interfering with their activity.

There is also evidence that the characteristics of each PI have a direct impact on its visibility, as our analysis revealed that the *Disk* was marginally more effective at capturing user attention than the other PIs examined in this study. This suggests that specific visual features, such as variations in color or motion, may be critical in increasing their detectability, and confirms the need for extensive research to determine the optimal design parameters in different usage contexts.

The implications of our findings regarding the effectiveness of PIs have a significant impact on the privacy of Android users. The *Classic* PI, as implemented, fails to capture the user's attention consistently during tasks requiring high levels of concentration, so malicious apps could exploit this flaw to conceal privacy-damaging actions during periods of intense user engagement. Therefore, there is an urgent need to design PIs that are more noticeable and can intelligently adapt to the current level of cognitive load, ensuring that users are alerted to privacy risks even when completely absorbed in their activity. This aims to safeguard user privacy using instinctive and non-disruptive methods, avoiding the danger of desensitization that can occur with frequent alerts.

While the study provides novel insights into PI effectiveness, it also presents limitations, including the experiment's controlled environment and the participants' specific demographic, which may not fully reflect the diversity of real-world application users. The experimental evaluation primarily involved university students, who are generally young and well-educated. This demographic, while

reasonable for initial testing, does not encompass all possible user groups, particularly those of different ages, educational backgrounds, and digital literacy levels. Recognizing this, the need to conduct field studies becomes more evident, as such studies could provide a more comprehensive understanding of the effectiveness of privacy indicators across a broader and more diverse population. This would help to reinforce these findings in everyday contexts and ensure the applicability of our proposals to a wider range of users.

6.5 Final Remarks

Our study contributes significantly to understanding the effectiveness of Privacy Indicators (PIs) in mobile applications. Our findings suggest that the current implementation of the *Classic* PI in Android is inadequate for consistently capturing user attention, particularly in situations that demand high cognitive engagement. This raises concerns about user privacy, as these indicators are the primary alert mechanism against unauthorized resource usage like camera or microphone access by potentially malicious applications.

Exploring alternative PIs, such as *Disk* and *Edge*, revealed that animation and visibility play crucial roles in attracting user attention. The *Disk* PI, with its dynamic animation, demonstrated a higher efficacy in catching the user's gaze in low-attention scenarios compared to the *Classic* and *Edge* PIs. However, the effectiveness of these indicators in high-attention tasks remains limited, indicating a need for more innovative approaches.

Future research should focus on developing more effective PIs that can alert users even during high-concentration tasks. Exploring dynamic and context-sensitive PIs that adapt their visibility based on the user's current activity may prove beneficial. Additionally, investigating user's cognitive load and attention span in relation to PI visibility could offer deeper insights into designing more effective privacy indicators. Further studies could also explore the integration of auditory or haptic feedback as supplementary or alternative alert mechanisms.

An intriguing direction for these studies would be to examine how privacy indicators directly linked to specific in-app actions, such as submitting responses in a quiz or completing a level in a game, might influence users' awareness and

attitudes towards privacy. Such investigations could deepen our understanding of the interplay between privacy notifications and user experience in mobile applications, enriching the ongoing dialogue on digital privacy. The ultimate goal is to strike a balance between ensuring user privacy and maintaining a non-intrusive user experience.

6.5.1 Data availability

The data supporting the findings of this study, derived from observations collected as participants performed the tasks, are publicly available [1].

[1] https://github.com/rmilanese99/privacy-indicators-eye-tracking/

Part IV

Advancing Android Security through Comprehensive Analysis of Context-Aware Permission Model: From SCoPe Implementation to LLM-Based Security Assessments in Android

"The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards – and even then I have my doubts."

Gene Spafford

CHAPTER 7

SCoPe: a fine-grained, context-aware permission model for Android

Contents

7.1 Introduction 105

7.2 SCoPe: Tool Design and Technical Implementation 110

7.2.1 Employed technologies 110

7.2.2 System architecture 112

7.2.3 System components 113

7.3 From Manual to Automated: A Preliminary Study on Transitioning Context Detection from RPCDroid to SCoPe 138

7.4 Smarter Testing for Android Apps: Moving Beyond Randomness with AI 142

7.4.1 Experimental Study Design RQ_1 143

7.4.2 Experimental Procedure 144

7.4.3 Analysis Results 149

7.4.4 Final remarks 158
7.5 Evaluating the Efficacy of SCoPe Automated Context Identification Against Manual Analysis in Android Apps 159
7.5.1 Experimental Design 159
7.5.2 Results 161
7.5.3 Conclusion 162

7.1 Introduction

In Chapter 4, we introduced RPCDroid, a tool that exposes critical vulnerabilities in Android's permission system by monitoring how applications access sensitive device resources. Our findings confirmed a common behavior in Android apps. After granting permissions, apps often access user data across multiple contexts, making it difficult for users to manage permissions and protect their privacy. For example, a granted camera permission might be used for both capturing profile pictures and live video streaming, even though users might only expect access to the former. There is a need for a more context-sensitive approach to permission management that accounts for the circumstances in which an app requests resource access, as the current system has a flaw.

Building upon these insights, Chapter 5 evaluated the Privacy Indicator (PI) feature introduced in Android 12, designed to enhance user awareness when sensitive resources like the camera or microphone are accessed. However, our research indicated that these indicators often go unnoticed, failing to significantly influence users' ability to detect when their resources are being accessed. This finding highlights a critical disconnect between the design of privacy alerts and user engagement, suggesting that current implementations do not sufficiently capture user attention. We then conducted an extensive study in Chapter 6 with an eye-tracking system. The results from this study reveal that the current implementation of the *Classic* PI, despite its purpose, fails to consistently capture user attention, particularly in high cognitive engagement scenarios. This poses a significant concern for user privacy, as these indicators are meant to serve as the primary defense against unauthorized access by potentially malicious applications.

Conversely, the *Disk* PI, characterized by its dynamic animation, showed a higher efficacy in drawing user gaze in low-attention scenarios. However, it still falls short during tasks that require high concentration, indicating that even well-designed visual alerts are not sufficient on their own. The proven efficacy of informative pop-up notifications in Chapter 5 demonstrates the critical need for proactive, text-based information that can educate and alert users in real-time, thereby significantly enhancing their understanding and management of privacy risks.

This chapter synthesizes the findings from the earlier chapters to propose a new permission model that incorporates real-time, context-aware information and dynamic privacy notifications. Our goal is to pioneer a paradigm where privacy protections are not only robust and user-centric but also intuitive and minimally invasive. This approach aims to rectify the inadequacies of the current system by providing a more personalized and proactive privacy management framework. The proposed system will not only address the gaps identified in the existing models but also set a new standard for privacy and security in the digital era, ensuring that users can trust and manage the applications they rely on every day.

For a permission-based security model to be sound, it should be able to deliver both *effectiveness* and *efficiency* of controls, ensuring timely notices and choices without over-burdening the user by constantly interrupting their activity [12]. At the outset, the approach taken for Android's authorization mechanism almost exclusively favored the latter efficiency at the expense of the former. Another major problem with this early implementation is that decisions have to be made prematurely and out of context, sometimes before using a feature that requires any permissions, or even the app itself [15]. In this way, the user doesn't have the opportunity to properly understand the purposes of accessing their data and the risks involved, so they can't make a fully informed decision [14]. Despite the additional contextual information available for decision-making, it is still too inflexible to grant permissions on a *per-app* basis [10]. While it makes sense to give a messaging app access to the microphone when recording a voice message, this opens the door for the app, and any third-party libraries that may be included, to use the same resource indefinitely [17], even across different contexts, and potentially misuse it for malicious or at least opaque purposes [10].

Our aim is to ensure that requests are well contextualized, as users can make more informed decisions according to what they are doing with the app at the time [12], and that there is always the option to deny access or even revoke consent at a later date [16].

A naive way of tackling the problem would be to simply ask for explicit consent every time an app seeks access to sensitive data, but this would mean constantly bombarding the user with invasive requests that interrupt their activity [12], with the risk of *warning fatigue* completely undermining the accuracy of their decisions, as they may stop paying attention to the prompts or even decide to permanently grant access to stop being bothered [20].

For this reason, the various approaches proposed over time to improve Android's permission model have focused on achieving the best possible balance between control granularity and system usability by managing privileges on a *per-feature* rather than *per-app* basis, deducing privacy preferences across different contexts and automatically applying them in new scenarios, to provide a powerful but effortless data protection mechanism. Unfortunately, all the existing approaches focus only on a small set of permissions rather than taking into account all dangerous permissions. Nevertheless, it is unpractical to develop a fully automated solution that does not require additional effort on the developer's part, prior analysis of the apps installed on the user's device, or even their repackaging. Therefore, this work aimed to implement a new and more flexible permission model for Android that infers the usage context of a sensitive resource directly at run-time, starting from the trace of the user's actions on the UI, and provides notices and choices for the individual features of an app, rather than for the entire app as a whole.

This Chapter describes the design and implementation of SCoPe *(Sensitive Context-aware Permission Enforcement Framework)*, a tool based on dynamic analysis techniques that, when installed on a rooted physical device or Android emulator as an Xposed module, allows access privileges to personal data and sensitive device resources to be managed at a much finer granularity than allowed by Android's native permission model.

Specifically, SCoPe not only monitors the use of these resources by intercepting calls to the centralized permission control mechanism but also tracks all user

actions on GUI elements and the events associated with their life cycle so that it can correlate them and extract the usage contexts.

This allows permissions to be assigned to individual features rather than the app as a whole, improving both user awareness through more informed and timely choices and, most importantly, the ability to control how their data is used and processed.

This chapter describes the evolution and validation of SCoPe, a proactive tool developed to enhance user privacy through precise permission management on Android devices. Building upon the foundational work of RPCdroid, which primarily served as an analytical tool, SCoPe has been re-engineered not just to analyze but to actively manage app permissions based on automatically identified usage contexts. This was achieved by developing a novel contextualization methodology that leverages user interactions with the graphical user interface, such as screen views and button presses, alongside the permissions accessed during these interactions. Furthermore, SCoPe memorizes user choices for each app, thus personalizing and enhancing the user experience by adapting to individual privacy preferences.

The remainder of our chapter is organized as follows. Initially, the chapter delves into the detailed architecture and functionality of SCoPe, illustrating how it transcends its predecessor, RPCdroid, by integrating a proactive component that presents users with timely pop-up notifications related to permission usage in recognized contexts. The first study was conducted to evaluate SCoPe's effectiveness involved comparing its ability to identify multiple permission usages within individual apps against the baseline established by RPCdroid. This comparative analysis aimed to validate the operational enhancements brought about by the new implementation. Subsequently, details about the integration with ARES + Monkey to improve input generation coverage are presented. In the end, results from automated context identification are presented and manually validated.

The second study primarily aimed at analyzing app executions, which subsequently enabled us to create a valuable dataset available to the community. This advancement involved transitioning from the Monkey tool, previously utilized in RPCdroid, to ARES, a state-of-the-art automated input generation tool. We

further enhanced the capabilities of ARES by integrating Monkey's randomness and leveraging frameworks like Cosmo and Jacoco within ARES to deepen our analytical reach into app behaviors. The study conducted a rigorous testing protocol on 100 open-source apps using various input generation algorithms—Monkey, ARES, and a combination of Monkey+ARES—across 50-minute sessions. This approach allowed us to compare the coverage achieved with these automated tools against that of manual execution by an expert, aiming to extensively explore and stress-test the apps. This meticulous analysis facilitated the creation of an extensive dataset, now available to the broader research community

After confirming that ARES, enhanced with Monkey, was more effective, this combination was integrated into SCoPe for further studies. The third study tested the efficacy of our system's automatic contextualization in identifying diverse or identical permission usage contexts. This phase involved running a subset of the 100 apps used in the previous study, alongside an extended analysis of 64 commercial apps from various categories in the Google Play Store. Two authors manually inspected the collected logs to determine the accuracy of the contextualization. The findings showed an 84% accuracy rate in automatic contextualization, affirming that SCoPe could serve as an effective, user-centered, proactive permission model that significantly contributes to safeguarding user privacy.

7.2 SCoPe: Tool Design and Technical Implementation

In this section, following a brief introduction to the dynamic instrumentation framework employed, the overall architecture of SCoPe, a context-aware permission model for Android, is presented. We then move on to a more detailed description of its various components and the implementation choices that guided its development and organization into packages.

7.2.1 Employed technologies

Xposed [122] is a widely used *dynamic instrumentation framework* for rooted Android devices that allows custom code to be injected directly into any process, be it an external app or the system itself, at run-time without the need to install custom versions of Android or repackage apps.

Specifically, Xposed allows the *hooking* of any Java method, whether declared by an external application or by the system itself, to monitor, modify, or completely replace it [105]. In any case, these injected instructions will also be executed within the original process, just as if nothing had changed [102].

As shown in Figure 7.1, in Android all processes are created by a `fork()` system call from a single parent process, the `Zygote`, which, once created at device startup, loads all the libraries and resources needed to run [103], which are then inherited by all child processes [123].

Xposed takes advantage of this very mechanism to inject its libraries into each process, modifying the code that is executed by the `Zygote` when it starts up. In particular, these startup instructions are contained in the `/system/bin/app_process` file, which is replaced with a modified version that also loads the `XposedBridge.jar` package into memory, which is therefore embedded in each app process [105].

Each time a hooked method is called, the `handleHookedMethod()` from this library is called first, which in turn calls two callback methods, `beforeHookedMethod()` and `afterHookedMethod()`, which, as the name suggests, are executed immediately before and after the hooked method [123].

```
public class Tutorial implements IXposedHookLoadPackage {
    public void handleLoadPackage(
            final LoadPackageParam lpparam)
            throws Throwable {
        if (!lpparam.packageName
                .equals("com.android.systemui"))
            return;

        findAndHookMethod(
            "com.android.systemui.statusbar.policy.Clock",
            lpparam.classLoader, "updateClock",
            new XC_MethodHook() {
                @Override
                protected void afterHookedMethod(
                        MethodHookParam param)
                        throws Throwable {
                    TextView tv = (TextView)
    param.thisObject;
                    String text = tv.getText().toString();
                    tv.setText(text + " :)");
                    tv.setTextColor(Color.RED);
                }
        });
    }
}
```

Listing 7.1: Example of an XPOSED module that defines hooks and callbacks for a system method

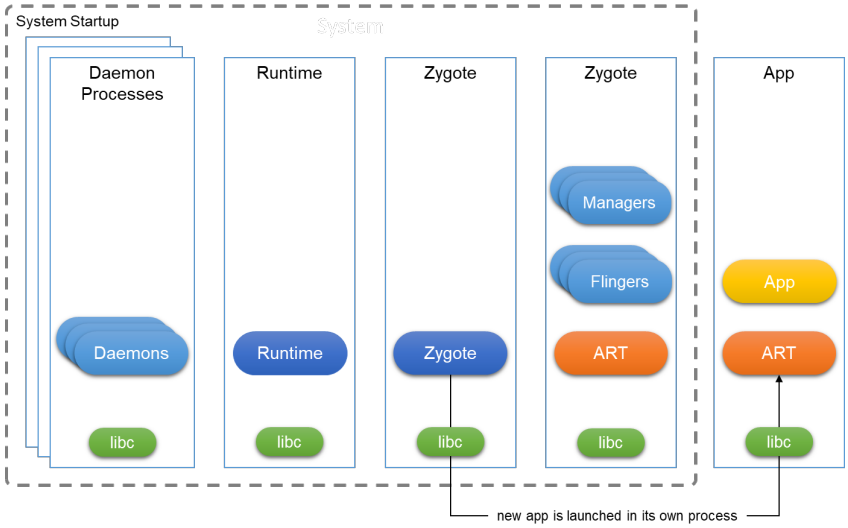


Figure 7.1: Schematic of the application launch process from the `Zygote`

As shown in the example in Listing 7.1, both the methods to be hooked and the callbacks to be called are defined in so-called XPOSED modules, which are developed and then installed on the device as normal apps [105], but which can also use the APIs provided by the `XposedBridge` library [122].

7.2.2 System architecture

SCOPE has been designed with a *repository* architecture directly inspired by Android's centralized permissions management system, shown at a high level in Figure 7.2.

In fact, a centralized `PermissionManager` is launched within the system process and performs very similar functions to the native `PermissionManagerService`, completely replacing it when it comes to checking and managing permissions.

Exploiting the `Binder` IPC mechanism, this central service communicates directly with various *bound services*, called `ContextManager`, which are instead registered at the level of individual apps, and are responsible for tracing the

actions performed by the user on the UI and then extracting the different contexts of use of protected resources from these traces.

Permissions are managed based on these contexts, rather than at the level of the entire app, as is the case with the current mechanism, for more granular control. When the `PermissionManagerService.checkUidPermission()` method is called, whether directly by an app or by an Application Framework service, the SCoPe `PermissionManager` retrieves the current context of the requesting app based on its UID. If the user has already made a decision for that specific *context-permission* pair, it is retrieved and enforced if necessary, otherwise, they are prompted, and the response is saved for later use.

7.2.3 System components

To facilitate the system development process and subsequently ensure a high degree of code maintainability, the components of SCoPe are organised into Java packages, as shown in the package UML diagram in Figure 7.3, according to the principle of *functional decomposition*. In fact, each subsystem is responsible for implementing specific features and then providing certain services to the other parts:

- The `context` package contains all the logic that is executed within the app processes to track and then extract context from user input.
- The `permission` package, on the other hand, contains all the logic executed within the system process to manage permissions according to these contexts, track user decisions, and enforce them.
- The `service` package contains the hooks and services needed to load and register a custom service into any app, which can then be bound to by remote processes via the `Binder` IPC protocol.
- The `config` package contains the services that load and manage the configuration files, including the list of apps to monitor and all saved decisions.
- The `hook` package contains interfaces and base classes for defining method and constructor hooks, while `util` contains helper classes.

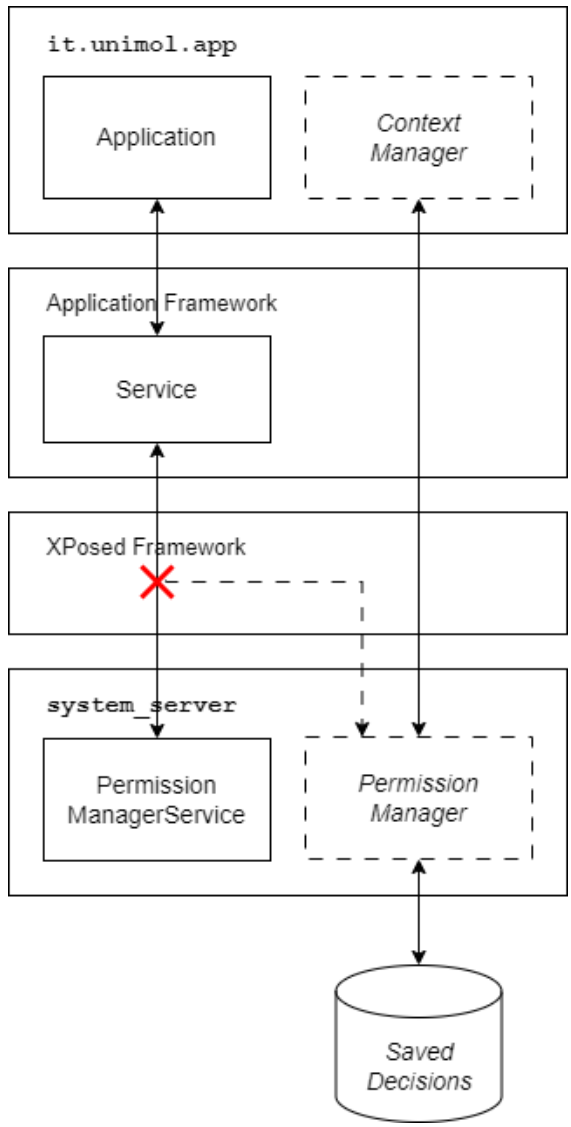


Figure 7.2: The overall system architecture. The dashed components are injected with Xposed into the processes of the monitored app or system

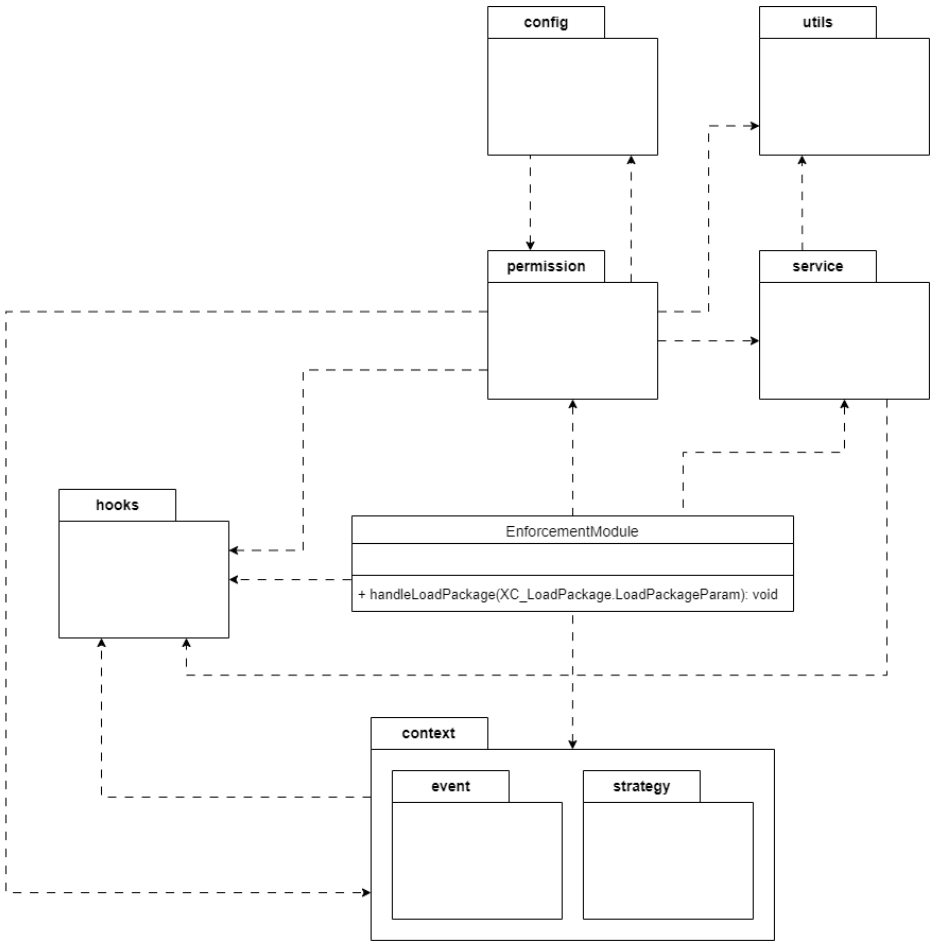


Figure 7.3: Package UML diagram with the packages in which SCoPe is organised

All these different components are then bound together and orchestrated by the `EnforceModule` class:

- Once the `android` system package is loaded into memory, the module initialises the centralised `PermissionManager`. This in turns calls the `ConfigService` running in the SCoPe process to load the list of apps to monitor and bind to their `ContextService`. The module also registers a hook for the `ActivityManagerService.getAppStartMode Locked()` method, so that the `ConfigService` can also run in the background, as shown in Listing 7.2.
- Instead, when the `com.android.permissioncontroller` package, which is related to displaying permission request prompts, is loaded, and the module first calls the `PermissionManager.disablePackageName PermissionCache()` method, so that the hooked `PermissionManager Service.checkUidPermission()` is called on each check, then initialises the `PermissionCheckHijacker` so that the native permission request dialogues are not displayed, but only the ones managed by SCoPe, as shown in Listing 7.3.
- Finally, when another app starts, the module initialises the `Context Manager` and registers the `ContextService`, then again disables response caching for permission checks by calling `PermissionManager. disablePermissionCache()`, so that they always reach the centralised mechanism, as shown in Listing 7.4.

Hook definition

The `MethodHook` and `ConstructorHook` classes help simplify the definition of hooks for Java methods and constructors, respectively, by completely wrapping the underlying Xposed `XC_MethodHook` instance and allowing its life-cycle to be managed using the `hook()` and `unhook()` methods defined in the `IHook` interface, which they implement.

Hooking a method requires specifying its simple name, the canonical name of the declaring class, and the callback method to execute. It is also possible

```
public class EnforcementModule
        implements IXposedHookLoadPackage {

    [...]

    @Override
    public void handleLoadPackage(
            XC_LoadPackage.LoadPackageParam lPParam)
            throws Throwable {

        if (lpp.packageName.equals(SYSTEM_PACKAGE)) {

            // Initialize the PermissionManager
            PermissionManager manager =
                PermissionManager.getInstance(lPParam);

            /* Allow bound services to be launched also
               in background for the E-RPCDroid process */
            new MethodHook(
                "com.android.server.am.
                    ActivityManagerService",
                "getAppStartModeLocked",
                true, new XC_MethodHook() {

                    @Override
                    protected void beforeHookedMethod(
                        MethodHookParam param) {

                        String packageName =
                            param.args[PACKAGE_NAME_ARG];

                        if (packageName.equals(
                                    MODULE_PACKAGE))
                            param.setResult(0);
                    }
                }).hook(lPParam);
        }

        [...]
    }
}
```

Listing 7.2: Section of the SCoPe module that is executed when the system process starts

```
public class EnforcementModule
        implements IXposedHookLoadPackage {

    [...]

    @Override
    public void handleLoadPackage(
            XC_LoadPackage.LoadPackageParam lPParam)
            throws Throwable {

        [...]

        } else if (lpp.packageName.equals(
                            CONTROLLER_PACKAGE)) {

            /* Disable permission checks caching, so that
               requests always reach the PermissionManager
               Service */
            Class<?> pm = XposedHelpers.findClass(
                "android.permission.PermissionManager",
                lPParam.classLoader);
            XposedHelpers.callStaticMethod(
                pm, "disablePackageNamePermissionCache");

            // Initialize the PermissionRequestHijacker
            PermissionRequestHijacker.getInstance()
                                     .initialize(lPParam);
        }

        [...]
    }
}
```

Listing 7.3: Section of the SCoPe module that is executed when the `permissioncontroller` process starts

```
public class EnforcementModule
        implements IXposedHookLoadPackage {

    [...]

    @Override
    public void handleLoadPackage(
            XC_LoadPackage.LoadPackageParam lPParam)
            throws Throwable {

        [...]

        } else {

            // Initialize the ContextManager
            ContextManager manager =
                ContextManager.getInstance();
            manager.traceMainEvents(lPParam);

            /* Register and load the bound service that
               allows the ContextManager to be called
               also from remote processes */
            ServiceLoader loader =
                ServiceLoader.getInstance(lPParam);
            loader.addService(ContextService.class);

            /* Disable permission checks caching, so that
               requests always reach the PermissionManager
               Service */
            Class<?> pm = XposedHelpers.findClass(
                "android.permission.PermissionManager",
                lPParam.classLoader);
            XposedHelpers.callStaticMethod(
                pm, "disablePermissionCache");
        }
    }
}
```

Listing 7.4: Section of the SCoPe module that is executed when any other app is launched

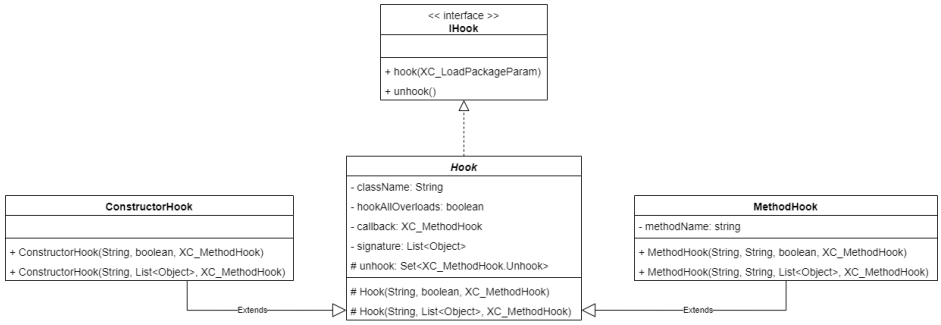


Figure 7.4: Class diagram with the main classes and interfaces of the `hook` package

to specify the signature of a particular overload, as a list of classes, or set the `hookAllOverloads` flag to true to hook all of them. For constructors, things are similar, except that since they have no name, only the class they belong to has to be stated.

In this way, defining a hook simply consists of instantiating one of these two classes, followed later by calling the `hook()` method to make it effective, as shown previously in Listing 7.2.

If the same method needs to be hooked into multiple components or processes, it is also possible to declare a new class that extends one of the two base types, as in Listing 7.5, to avoid duplication of code by increasing reuse and modularity.

Service injection

Although the SCoPe Xposed module is the same for the system, the `com.android.permissioncontroller` package, and all other apps, the copies running in each of these processes are actually completely separate, so variable values are not directly shared between copies and there is no other built-in communication mechanism.

For the central `PermissionManager`, running in the system process, to call the `ContextManager` instances running in the processes of the different apps instead,

```
public class ApplicationCreateHook extends MethodHook {

    private static final String CLASS_NAME =
        "android.app.Application";
    private static final String METHOD_NAME =
        "onCreate";

    // The hooked method has no parameters
    private static final List<Object> METHOD_SIGNATURE =
        null;

    public ApplicationCreateHook(XC_MethodHook callback)
            throws EmptyFieldException,
                   InvalidFieldException {

        /* Call the constructor of the extended class
           MethodHook */
        super(ApplicationCreateHook.CLASS_NAME,
                ApplicationCreateHook.METHOD_NAME,
                ApplicationCreateHook.METHOD_SIGNATURE,
                callback);
    }
}
```

Listing 7.5: Definition of a class that extends the base `MethodHook` type to hook the same method into multiple separate components

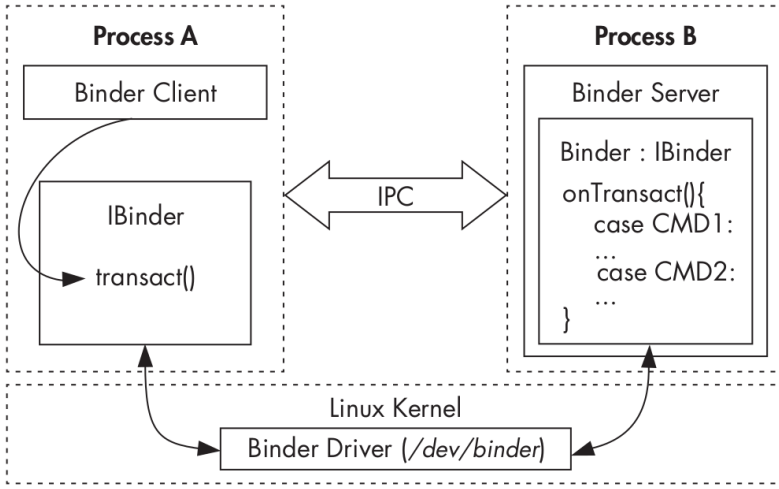


Figure 7.5: Inter-process communication through the `Binder` infrastructure

so that permission requests can be associated with their individual features, it is necessary to exploit Android's IPC mechanism.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest

    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="it.unimol.app">

    <application>

        [...]

        <service
            android:name="it.unimol.app.BookletService"
            android:exported="true" />

    </application>

    [...]
</manifest>
```

Listing 7.6: Declaration of an exported service in the Manifest file

`Service` components can be used by an app not only to perform a long-running task in the background, but also to expose some of its functionality to others. In the latter case, it is referred to as a bound service, which can be seen as a server in a client-server interface, allowing a persistent inter-process connection to be established through the `Binder` infrastructure to interact with it [124], as shown in Figure 7.5.

As with all other components of an app, services must be declared in the Manifest by adding a `<service>` element as a child of the `<application>` element. In addition to specifying the fully qualified name of the class that defines the service, the `android:exported` value must be set to `true` for external applications to bind to it [125], as in the example in Listing 7.6. If an attempt is made to start an undeclared service, an exception is thrown, and the operation fails.

For this reason, even with Xposed, injecting a service and starting it directly inside a process is impossible. The SCoPe components that deal with circumventing this protection mechanism are all contained in the `service` package, as shown in Figure 7.6.

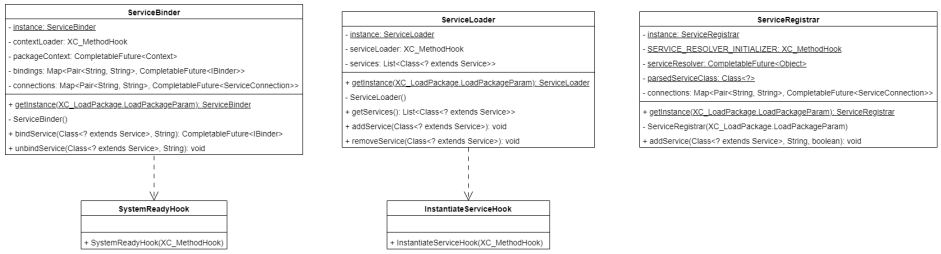


Figure 7.6: Class diagram with the classes of the `service` package

Specifically, the `ServiceRegistrar` class manually registers a service within any package, as if it had been declared in its Manifest. In fact, when an app is installed, the `PackageManagerService` parses it, then calls the `ComponentResolver.addAll Components()` method, which registers all the components, including activities and services, defined within it. This `SCoPe` class provides the `addService()` interface, the implementation of which is shown in Listing 7.7. Thanks to XPOSED, this implementation internally calls the `ComponentResolver.addService()` method, simulating the addition of the service to be injected into the Manifest.

Although a custom service can be started this way, a second tweak is required to correctly load its code into a process other than the package that defines it.

The `ServiceLoader` class, through an `InstantiateServiceHook` that allows the `AppComponentFactory.instantiateService()` method to be modified, actually sets the correct class loader[1] for an injected service, which is not the one of the target package, but that of the package in which it is implemented, whenever it is spawned, as shown in Listing 7.8.

Finally, the `ServiceBinder` class simplifies the process of binding to remote services by wrapping Android's IPC protocol and automatically handling these connections, exposing the `bindService()` and `unbindService()` interfaces to clients.

[1]A class loader is an object responsible for loading Java classes. Given the binary name of a class, a class loader should attempt to locate or generate data that constitutes its definition.

```
public class ServiceRegistrar {

    [...]

    public void addService(
                    Class<? extends Service> serviceClass,
                    String packageName,
                    boolean exported) {

        [...]

        /* Parses the passed Service class into a
           ParsedService instance */
        Object parsedService =
                XposedHelpers.newInstance(
                    ServiceRegistrar.parsedServiceClass);

        XposedHelpers.setObjectField(
                parsedService, "name",
                serviceClass.getCanonicalName());

        XposedHelpers.setObjectField(
                parsedService, "packageName",
                packageName);
        XposedHelpers.setObjectField(
                parsedService, "processName",
                packageName);

        XposedHelpers.setBooleanField(
                parsedService, "exported",
                exported);

        /* Registers manually the injected service into
           the given package by calling the addService
           method of the ComponentResolver class */
        ServiceRegistrar
                .serviceResolver
                .whenComplete((s, t) ->
                    XposedHelpers.callMethod(
                        s, "addService", parsedService));
    }
}
```

Listing 7.7: Extract of the `addService()` method implemented in the `ServiceRegistrar` class

```
public class ServiceLoader {

    [...]

    private final XC_MethodHook serviceLoader =
        new XC_MethodHook() {

        @Override
            protected void beforeHookedMethod(
                            MethodHookParam param)
                        throws Throwable {
                super.beforeHookedMethod(param);

                /* Retrieves the name of the service to
                   be instantiated */
                String serviceName = (String)
                    param.args[SERVICE_NAME_ARG_INDEX];

                if (serviceName == null)
                    return;

                /* If the service is managed by SCoPe,
                   then its class loader is set */
                services.stream()
                        .filter(s ->
                            serviceName.equals(
                                s.getCanonicalName()))
                        .findFirst()
                        .ifPresent(s ->
                            param.args[
                               CLASS_LOADER_ARG_INDEX] =
                           s.getClassLoader());
        }
    };

    [...]

    private void initialize(
        XC_LoadPackage.LoadPackageParam lPParam) {

        new InstantiateServiceHook(this.serviceLoader)
            .hook(lPParam);
    }
}
```

Listing 7.8: Extract of the `InstantiateServiceHook` callback implemented in the `ServiceLoader` class

Context extraction

The entire context extraction process, as mentioned above, takes place within the individual processes of the audited apps, and as shown in the UML diagram in Figure 7.7, all of this logic is implemented in the classes of the `context` package.

Specifically, the `ActivityEventHook` and `ViewEventHook` classes, both of which extend the `IHook` interface of the `hook` package, define the hooks for all `Activity` life-cycle methods, such as `onStart` when a new screen is displayed, or for `View` callback methods, such as `onClick` when the user taps on a UI element.

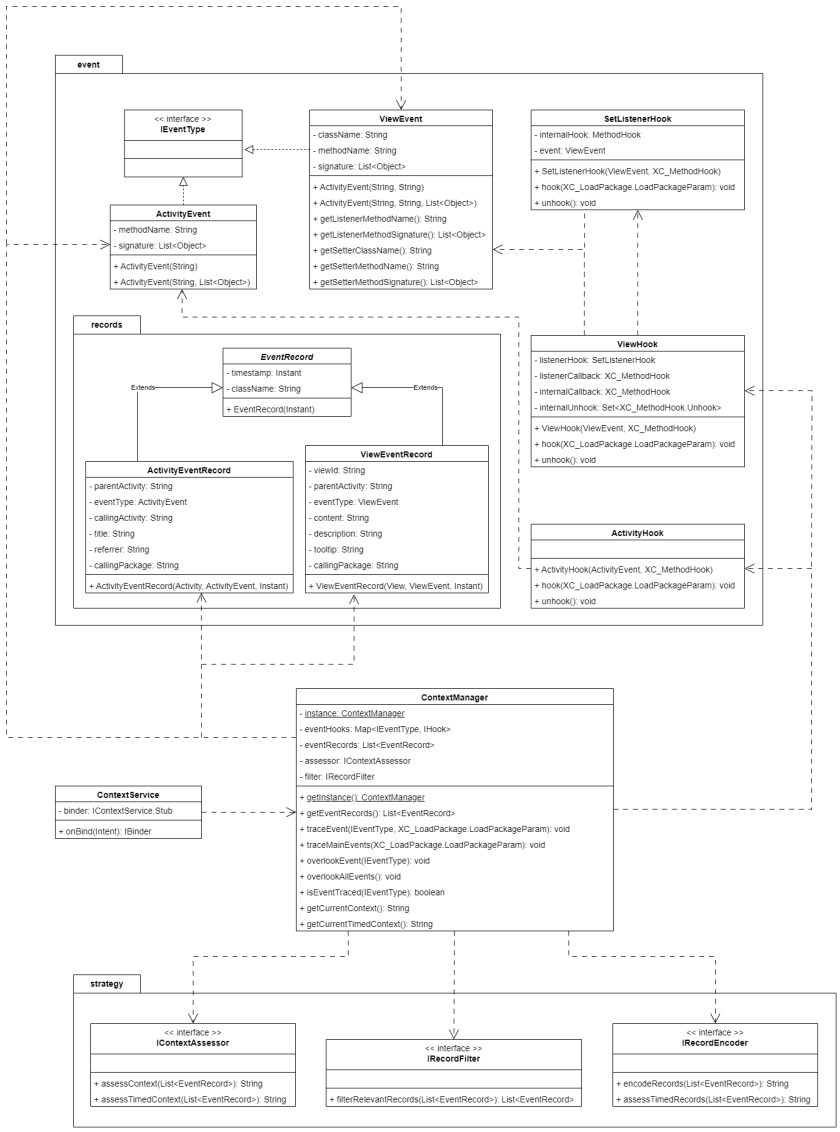


Figure 7.7: Class diagram with the main classes and interfaces of the `context` package

Inside the `ContextManager` class, these hooks are implemented and activated to monitor all actions the user takes while using the app. As shown in Listing 7.9, all the logic necessary to keep track of these events is implemented in the respective callback methods, which add an `EventRecord` containing key information whenever they occur:

- For each `ActivityEvent`, the `Activity` class name, its title, the calling `Activity` and the `referer` field of the `Intent` with which it was launched, if any, are recorded, along with the type of event that occurred and the timestamp.
- For each `ViewEvent`, the unique ID of the `View`, its content, description, tool-tip, if any, and the class name of the `Activity` to which it belongs are stored, as well as the type of event that occurred and the timestamp as in the previous case.

When the `getCurrentContext()` method, reported in Listing 7.10, is called, these records are then processed by three components whose interfaces are all defined within the `strategy` sub-package:

- The `IRecordFilter` interface declares the `filterRelevantRecords()` method, which takes a list of `EventRecord` instances as input and returns a reduced version that contains only those records that are useful for context extraction, to reduce memory footprint.
- The `IRecordEncoder` interface declares the methods `encodeRecords()` and `encodeTimedRecords()` which, given a list of `EventRecord` instances, return a compact representation of them. The latter method, unlike the former, also encodes their timestamps.
- The `IContextAssessor` interface declares the methods `assessContext()` and `assessTimedContext()`, which, given a list of `EventRecord` instances, manage to extract the app context, whether or not timestamps are taken into account.

For each of these interfaces, SCoPe provides a standard implementation:

```
public class ContextManager {

    [...]

    private void hookViewEvent(
        ViewEvent eventType,
        XC_LoadPackage.LoadPackageParam lPParam) {

        ViewHook eventHook =
            new ViewHook(eventType, new XC_MethodHook() {

            @Override
            protected void beforeHookedMethod(
                            MethodHookParam param)
                        throws Throwable {
                super.beforeHookedMethod(param);

                View view = (View)
                    param.args[ViewHook.VIEW_ARG_INDEX];

                ViewEventRecord record =
                       new ViewEventRecord(view,
                                           eventType,
                                           Instant.now());

                addEventRecord(record);
            }
        });

        eventHook.hook(lPParam);

        this.eventHooks.put(eventType, eventHook);
    }

    [...]
}
```

Listing 7.9: Callback method that is executed each time the user performs an action on a `View` in the app

```
1  public class ContextManager {
2
3      [...]
4
5      private String getCurrentContext(boolean keepTimestamp)
6              throws InvalidOperationException {
7          if (this.assessor == null)
8              throw new InvalidOperationException(
9                  "the context assessor has not been set yet");
10
11         // Sorts the records by their timestamp
12         this.sortEventRecords();
13
14         /* Calls an implementation of IRecordFilter so
15            that no longer useful records are removed */
16         this.trimEventRecords();
17
18         /* Calls an implementation of IContextAssessor so
19            that the current context is extracted and then
20            returned */
21         if (keepTimestamp)
22             return this.assessor
23                     .assessTimedContext(this.eventRecords);
24         else
25             return this.assessor
26                     .assessContext(this.eventRecords);
27     }
28 }
```

Listing 7.10: Extract of the `getCurrentContext()` method implemented in the `ContextManager` class

```
interface IContextService {

    String getCurrentContext();
    String getCurrentTimedContext();
}
```

Listing 7.11: Definition of the `IContextService` interface using the AIDL

- `TimedMethodFilter`, which implements `IRecordFilter`, removes all records that are older than a specified time threshold, which is 1 minute by default.

- `DefaultRecordEncoder`, which implements `IRecordEncoder`, computes the MD5 hash of each record, then concatenates them and recomputes the hash of that string so that there are always identically-length representations.

- `SimpleContextAssessor`, which implements `IContextAssessor`, retrieves the most recent event in order of time and returns the encoding of the corresponding record.

It is, of course, possible to define different implementations and to use them in the `ContextManager` in place of the default ones.

The `IContextService` interface, defined using the AIDL as shown in the above Listing, exposes both the `getCurrentContext()` and `getCurrentTimedContext()` methods.

The `ContextService` class, which implements the stub automatically generated by the AIDL compiler as shown in Listing 7.12, merely acts as a wrapper around the `ContextManager`, which is normally inaccessible to external processes, by directly calling its appropriate methods. The binding process takes place in the `onBind()` method, which returns the interface that clients can use to interact with the service.

This service is then injected and started within the processes of all monitored apps, as already shown in the SCoPe module excerpt in Listing 7.4, so that the central permission management mechanism can access the context and associate access requests with it.

```
public class ContextService extends Service {

    private final IContextService.Stub binder =
        new IContextService.Stub() {

        @Override
        public String getCurrentContext()
                throws RemoteException {
            try {
                return ContextManager
                            .getInstance()
                             .getCurrentContext();
            } catch (InvalidOperationException e) {
                throw new RemoteException(e.getMessage());
            }
        }

        @Override
        public String getCurrentTimedContext()
                throws RemoteException {
            try {
                return ContextManager
                            .getInstance()
                            .getCurrentTimedContext();
            } catch (InvalidOperationException e) {
                throw new RemoteException(e.getMessage());
            }
        }
    };

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return this.binder;
    }
}
```

Listing 7.12: Implementation of the `IContextService` interface in the `ContextService` class

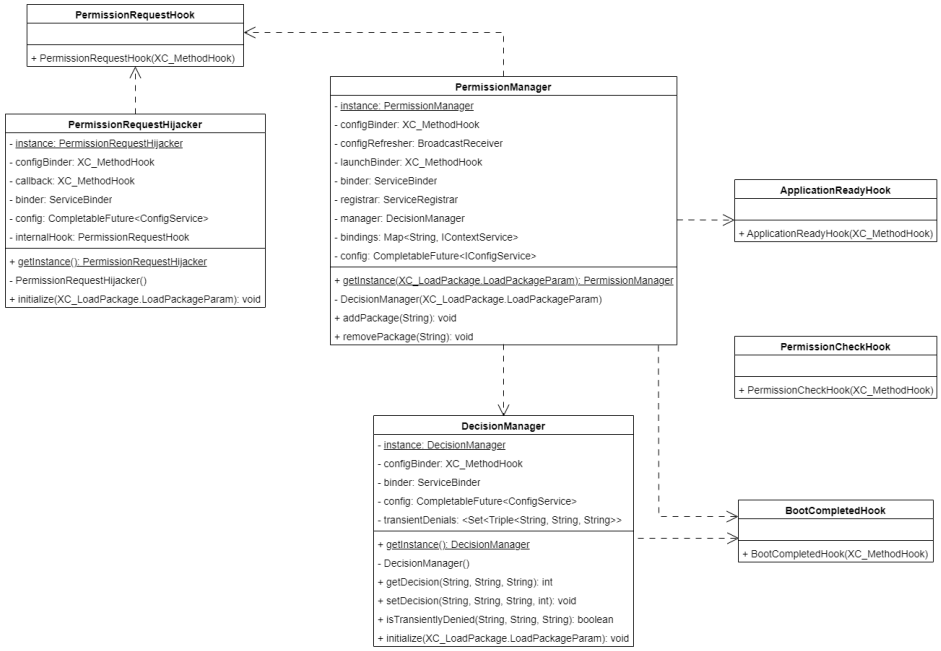


Figure 7.8: Class diagram with the main classes of the `permission` package

Permission management

While the context extraction activity is distributed across different app processes, the permission checking and enforcement activity occurs entirely in the system process.

The above class diagram shows all the components that perform this task are located in the `permission` package.

In particular, the `PermissionCheckHook` class declares a hook for the `PermissionManagerService.checkUidPermission()` method, which, is the core of Android's permission validation mechanism and is called by all the other checking APIs. This makes it possible to intercept explicit permission requests from applications, as well as checks performed internally within the Application Framework as a result of the enforcement process.

This hook is then instantiated and made effective within the `PermissionManager` class, which handles the entire process of checking, requesting and, indirectly, enforcing permissions, and is loaded into the system process by the SCoPe module, as already reported in the extract from Listing 7.2.

Then, when the `PermissionManagerService.checkUidPermission()` method is called, SCoPe takes over the request, retrieves the context of the corresponding app via the associated `ContextService`, to which it is connected via the `Binder` IPC infrastructure, and checks whether the user has already been asked for that particular context-permission pair. This last task is handled by the `DecisionManager`, which also manages the persistence of choices via the `ConfigService`.

If the user has not yet been prompted, a dialogue box like the one shown in Figure 7.9 will appear, giving the user the option to allow or deny access rights to that specific app feature, then the decision is saved.

Finally, according to the user's preferences, either `PackageManager.PERMISSION_GRANTED` or `PackageManager.PERMISSION_DENIED` is set as the return value of the `PermissionManagerService.checkUidPermission()` method. In this way, permissions are enforced by relying on the standard Android mechanisms, and no further action is required.

Native permission request dialogues are never displayed, even when explicitly called up by apps, as the `PermissionRequestHijacker` component hooks the `GrantPermissionsActivity.onCreate()` life-cycle method and prevents its execution.

```
public class PermissionManager {

    [...]

    private void initialize(
        XC_LoadPackage.LoadPackageParam lPParam) {

        [...]

        new PermissionCheckHook(new XC_MethodHook() {

            @Override
            protected void beforeHookedMethod(
                            MethodHookParam param)
                        throws Throwable {
                super.beforeHookedMethod(param);

                int uid = (int)
                    param.args[PACKAGE_UID_ARG_INDEX];

                String packageName =
                    PackageUtils.getPackageName(uid);
                String permission = (String)
                    param.args[PERMISSION_ARG_INDEX];

                String context = bindings
                                    .get(packageName)
                                    .getCurrentContext();

                if (decision == NO_USER_DECISION) {

                    /* Request permission from the user,
                       then save the context-decision
                       pair */

                }

                /* Sets the return value for the checkUid
                   permission() method, so that decisions
                   are also enforced */
                if (decision == PERMISSION_GRANTED) {
                    param.setResult(
                       PackageManager.PERMISSION_GRANTED);
                } else {
                    param.setResult(
                       PackageManager.PERMISSION_DENIED);
                }
    }
}
```

Listing 7.13: Extract of the `PermissionCheckHook` callback implemented in the `PermissionManager` class

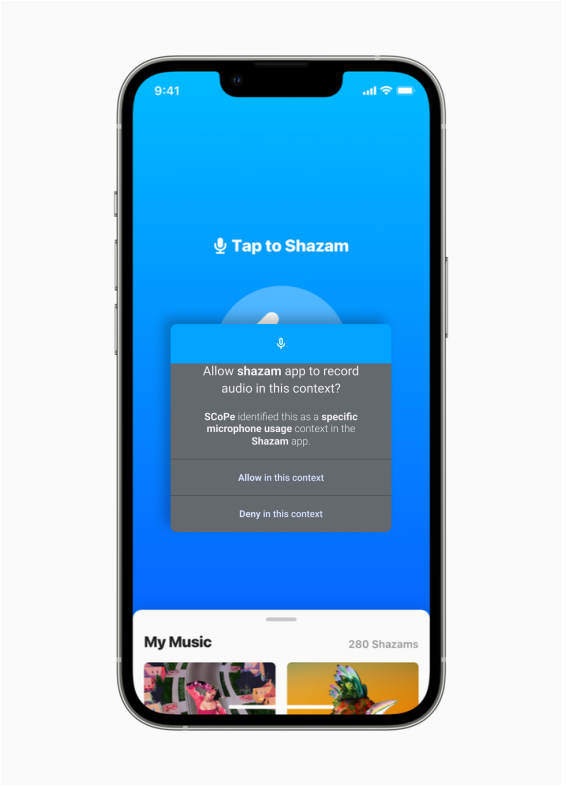


Figure 7.9: Example of SCoPe permission request dialogue in a specific identified context (after touching *Tap to Shazam*).

7.3 From Manual to Automated: A Preliminary Study on Transitioning Context Detection from RPCDroid to SCoPe

This section assesses SCoPe's ability to both extract the context of the use of personal data and protected device resources and to enforce user choices. Specifically, the new approach is compared to the already exposed in Chapter 4 RPCDroid, a dynamic analysis tool that generates traces of permission usage and is used to detect these contexts manually.

The first step was to evaluate the ability of SCoPe to automatically extract the context of using protected resources within apps. The study involved 10 apps that could be downloaded for free from the Play Store, listed in Table 7.1, and that had previously been analyzed with `RPCDroid` [126], a dynamic analysis tool that can track user actions and permission requests, but without automatically inferring their context of use, which was done manually from these traces in the study.

Methodology

Each app was installed on a rooted emulator with a specially modified version of the SCoPe module enabled, and then automatically executed with Monkey, using the same seed as in the previous study, so that the same pseudo-random actions were performed on the UI.

This particular version of SCoPe does not show any dialogue, just an informative, non-blocking toast that simply logs context-permission pairs, so the execution is exactly the same as with RPCDroid.

The SCoPe log files, an example of which is given in Listing 7.14, were then analyzed to determine the number of unique contexts in which each permission was used, so that they could be directly compared with the results of the RPCDroid study. In all cases where the number of identified contexts differed, a manual analysis was carried out to determine the causes of these differences.

Name	Package	Version
Amazon Shopping	com.amazon.mShop.android.shopping	24.6.0.100
eBay	com.ebay.mobile	6.48.0.1
Facebook	com.facebook.katana	353.0.0.0.4
Google Earth	com.google.earth	9.151.0.2
Instagram	com.instagram.android	223.0.0.0.18
Shazam	com.shazam.android	12.6.0
Spotify	com.spotify.music	2.0.45
Sticker Maker	com.marsvard.stickermakerforwhatsapp	0.0.2-82
Google Translate	com.google.android.apps.translate	6.14.0.05.35
VLC for Android	org.videolan.vlc	3.4.3

Table 7.1: List of apps against which the ability of SCoPe to extract use context has been evaluated

```
{
  "package" : [ "com.shazam.android" ],
  "decisions" : {
    "(android.permission.ACCESS_COARSE_LOCATION,
      4507f0ad168d08ced53b096101a915d5)" : 0,
    "(android.permission.RECORD_AUDIO,
      ee69f821ccad6b8dc77b8c68a40ca850)" : 0,
    "(android.permission.RECORD_AUDIO,
      d5bf882111d2dbea78bbc0332d636ee3)" : 0,
    "(android.permission.RECORD_AUDIO,
      4507f0ad168d08ced53b096101a915d5)" : 0,
    "(android.permission.ACCESS_COARSE_LOCATION,
      433b15b5a7944f49e2737bf8780b136a)" : 0,
    "(android.permission.ACCESS_COARSE_LOCATION,
      d5bf882111d2dbea78bbc0332d636ee3)" : 0,
    "(android.permission.ACCESS_COARSE_LOCATION,
      f6bf1eb702a82fd029371e984b687fe4)" : 0,
  }
}
```

Listing 7.14: Example of SCoPe log file for the Shazam app

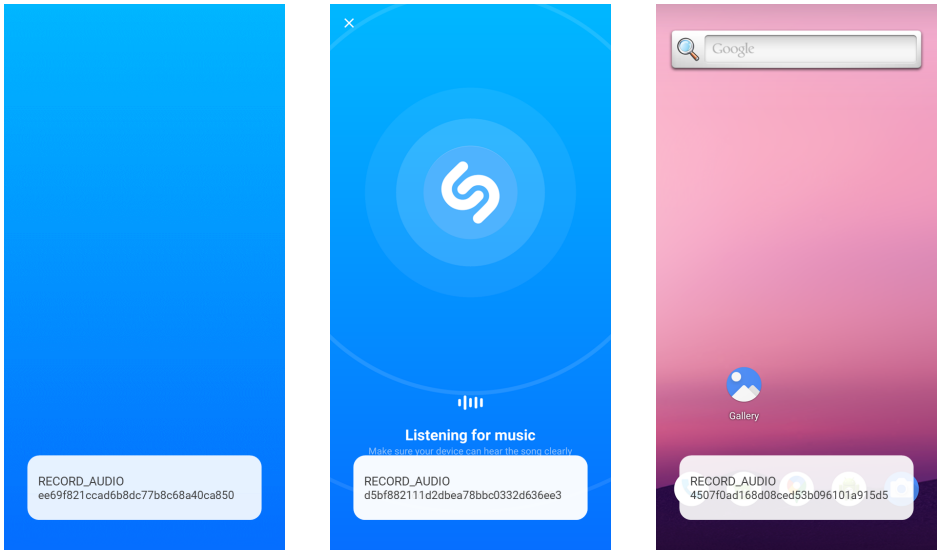
App	# Camera	# Location	# Microphone
Amazon Shopping	3 / 3	0 / 0	0 / 0
eBay	5 / 3	0 / 0	0 / 0
Facebook	2 / 2	22 / 19	0 / 0
Google Earth	0 / 0	6 / 5	0 / 0
Instagram	0 / 0	13 / 11	0 / 0
Shazam	0 / 0	4 / 6	3 / 1
Spotify	0 / 0	0 / 0	0 / 0
Sticker Maker	1 / 1	0 / 0	0 / 0
Google Translate	3 / 6	0 / 0	5 / 4
VLC for Android	0 / 0	0 / 0	0 / 0

Table 7.2: Number of identified usage contexts (SCoPe / RPCDroid) for each app and resource type

Results

Table 7.2 shows the results of the context evaluation performed automatically by SCoPe and manually with RPCDroid. As can be seen, in most cases the figures are slightly different between the two approaches.

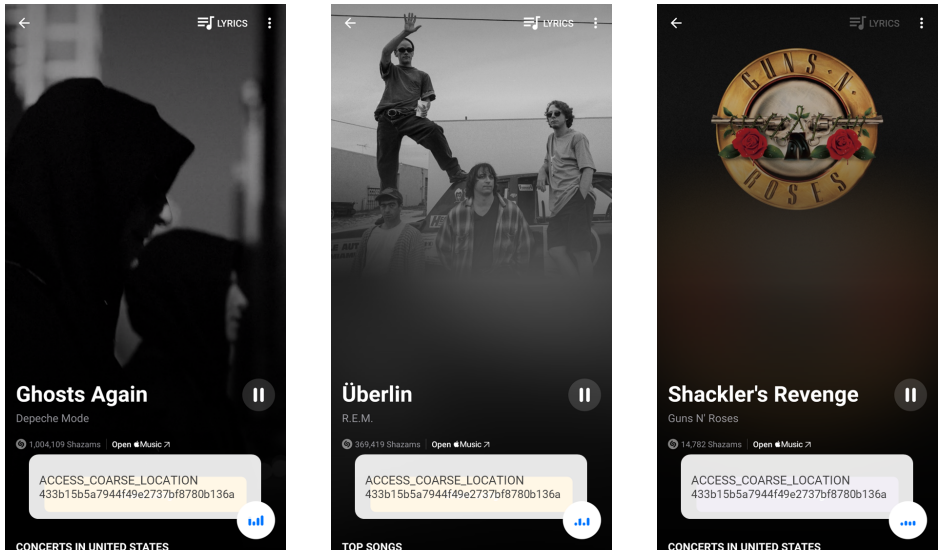
Subsequent manual analysis showed that in most cases SCoPe can characterize all the contexts identified with RPCDroid, and that the additional ones returned correspond to different app features. For example, in Shazam, SCoPe correctly detects the use of the microphone when the app is launched, when the song recognition functionality is active, and when the app is placed in the background, as shown in Figure 7.10, whereas with RPCDroid all these scenarios have



(a) (b) (c)

Figure 7.10: Different microphone access contexts identified by SCoPe in Shazam

been brought back into a single context. The only exception is eBay, where the additional contexts actually refer to an already visited feature, the only difference being that it was called from different UI elements. This penalizes the efficiency of the proposed solution, as it requires two approvals that are actually superfluous.



(a) (b) (c)

Figure 7.11: Location access contexts considered as different from RPCDroid, but merged by SCoPe

On the contrary, when the number of identified contexts was lower than that of RPCDroid, it was found that the excluded ones were duplicates. Again, with Shazam in mind, the contexts associated with accessing the user's location that have been merged are all related to the display of a song's details in the library. By considering the activity itself as a single feature, SCoPe can correctly infer the context, whereas RPCDroid considered a different context for each song displayed, which is wrong.

7.4 Smarter Testing for Android Apps: Moving Beyond Randomness with AI

The landscape of Android application GUI testing is undergoing a transformative shift, with the conventional pseudo-random input generation methods,

epitomized by tools such as Monkey, being challenged by the burgeoning capabilities of artificial intelligence (AI). This chapter delves into the intricacies of enhancing the input generation process by integrating a state-of-the-art Deep Reinforcement Learning (DRL) algorithm, namely SAC, to refine the exploration of mobile applications beyond the limitations of pseudo-random actions. While tools like Monkey offer an essential form of app testing by simulating user actions, their inherent randomness often fails to capture the nuances of real-user interactions. Consequently, such methods may overlook critical functionalities or edge cases within an application. This section explores the transition from this randomness to a more deliberate and user-centric interaction pattern through the integration of SAC, a cutting-edge DRL algorithm that facilitates a strategic and nuanced approach to input generation. In our quest to refine the art of input generation—often referred to as *fuzzing* in the software testing domain-we conducted an extensive survey of the existing automatic input generation tools for Android applications. Our exploration culminated in identifying ARES [19], a tool representing the actual state of the art of current advancements in this field. By adopting ARES, we aim to transcend the confines of pseudo-random input generation, tackling a realm where actions are not merely random but are guided by the learning capabilities of artificial neural networks. We selected 100 open-source applications from the F-Droid repository and conducted a combined testing approach. We incorporated Monkey within the ARES framework alongside SAC, allowing both algorithms to work together while re-running the applications. To validate the automated analysis, a human user conducted a manual exploration of each app, mirroring the 50-minute duration of the automated tests. The user's objective was to explore the app functionalities as deeply as possible, mimicking real-world user interactions. We meticulously measured the impact of all approaches (Monkey, SAC, combined Monkey & SAC, and manual exploration) on code coverage, measuring the percentage of the app's code exercised during testing by each approach.

7.4.1 Experimental Study Design RQ_1

This experimental study's primary *goal* is to evaluate the effectiveness of ARES, a Deep Reinforcement Learning-based input generation tool, in improving code

coverage during Android application testing with SCoPe. The study contrasts ARES's performance with that of the traditional Monkey tool, a hybrid approach combining Monkey and ARES, and a comprehensive manual testing strategy that aims to explore as many app scenarios as possible. The following research question guided our study:

RQ1 *How does the integration of ARES as an input generation technique affect the coverage of SCoPe in Android application testing compared to the previous Monkey approach and compared with a manual execution?*

7.4.2 Experimental Procedure

The design aims to examine the capabilities of each fuzzing methodology—Monkey, ARES, hybrid, and manual—in achieving extensive code coverage. It leverages COSMO for detailed code instrumentation, providing an in-depth understanding of each method's effectiveness in navigating and uncovering the complexities of Android applications.

Robust Measurement with COSMO: COSMO [127] is a tool that allows to automatically instrument a Gradle-based application and generate code coverage reports. Code coverage is a measure useful to describe the degree to which the source code of a program is executed when a particular test suite runs. Employing COSMO ensures a granular code coverage analysis. This tool not only instruments the code but also integrates coverage data with the source, offering detailed and actionable insight into the testing process's effectiveness. COSMO enables measuring how an application has been inspected, generating a coverage report after its execution. COSMO achieves this by instrumenting the application's source code before the usage session. Instrumentation is a process that involves inserting calls to special methods called instrumentation in the source code. These calls record information about the code's execution flow, such as which methods have been called and which lines of code have been executed. Additionally, COSMO can operate as a black-box tool, instrumenting an app starting from the compiled app. This allows COSMO to modify the source code of the Android app automatically, providing a comprehensive analysis of code

coverage. In addition, we used an instrumentation framework named JaCoCo for Android to generate a code coverage report. Specifically, COSMO prepares the application code so that it is ready for use with the aforementioned framework. The process involves:

1. **Modifying the "AndroidManifest.xml" file:** The manifest file describes essential information such as components, permissions, and app features. Once located, the tool declares a broadcast receiver named EndCoverageBroadcast as an app component. This receiver allows the application to receive intents named `intent.END_COVERAGE`.

2. **Creating the EndCoverageBroadcast.java file:** A file named *EndCoverageBroadcast.java* is created in the project directory. This file implements the EndCoverageBroadcast broadcast receiver, previously added to AndroidManifest.xml. When activated by a developer, the intent marked as `intent.END_COVERAGE` collects the application's code coverage during the test phase. The code coverage file has a *.ec* extension.

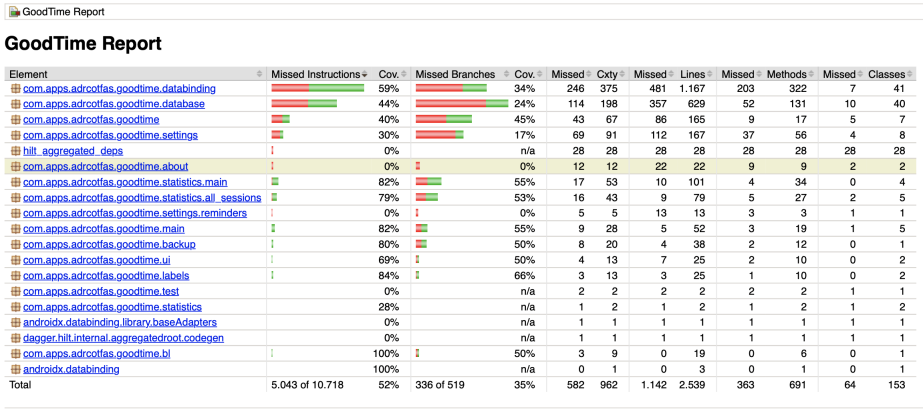
3. **Adding dependencies to the build.gradle task:** The *build.gradle* file is an advanced development toolkit for automating and managing the build process, enabling the definition of flexible and custom build configurations. Dependencies are added to the build.gradle task. Moreover, the tool inserts a new task, *jacoco-instrumenter-coverage.gradle*, which generates the final coverage report from the **.ec* files generated during the application's test phase.

Then, **JaCoCo** is used to measure the code coverage of Java applications. It helps answer the question: "*How much of the source code is executed during an application's usage session?*" Besides measuring code coverage, JaCoCo also generates detailed visual reports (Figure 7.12) that clearly show which parts of the code have been tested and which require further testing. Specifically, it introduces:

- **Red diamond:** indicates that no branches were exercised during the test phase.

- **Yellow diamond:** indicates that some branches were not exercised.

Figure 7.12: Example *"file.html"* generated at the end of the app analysis



GoodTime Report

GoodTime Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.apps.adrcotfas.goodtime.databinding		59%		34%	246	375	481	1.167	203	322	7	41
com.apps.adrcotfas.goodtime.database		44%		24%	114	198	357	629	52	131	10	40
com.apps.adrcotfas.goodtime		40%		45%	43	67	86	165	9	17	5	7
com.apps.adrcotfas.goodtime.settings		30%		17%	69	91	112	167	37	56	4	8
hilt_aggregated_deps		0%		n/a	28	28	28	28	28	28	28	28
com.apps.adrcotfas.goodtime.about		0%		0%	12	12	22	22	9	9	2	2
com.apps.adrcotfas.goodtime.statistics.main		82%		55%	17	53	10	101	4	34	0	4
com.apps.adrcotfas.goodtime.statistics.all_sessions		79%		53%	16	43	9	79	5	27	2	5
com.apps.adrcotfas.goodtime.settings.reminders		0%		0%	5	5	13	13	3	3	1	1
com.apps.adrcotfas.goodtime.main		82%		55%	9	28	5	52	3	19	1	5
com.apps.adrcotfas.goodtime.backup		80%		50%	8	20	4	38	2	12	0	1
com.apps.adrcotfas.goodtime.ui		69%		50%	4	13	7	25	2	10	0	2
com.apps.adrcotfas.goodtime.labels		84%		66%	3	13	3	25	1	10	0	2
com.apps.adrcotfas.goodtime.test		0%		n/a	2	2	2	2	2	2	1	1
com.apps.adrcotfas.goodtime.statistics		28%		n/a	1	2	1	2	1	2	1	2
androidx.databinding.library.baseAdapters		0%		n/a	1	1	1	1	1	1	1	1
dagger.hilt.internal.aggregatedroot.codegen		0%		n/a	1	1	1	1	1	1	1	1
com.apps.adrcotfas.goodtime.bl		100%		50%	3	9	0	19	0	6	0	1
androidx.databinding		100%		n/a	0	1	0	3	0	1	0	1
Total	5.043 of 10.718	52%	336 of 519	35%	582	962	1.142	2.539	363	691	64	153

- **Green diamond:** indicates that all branches were exercised during testing.

This is crucial for developers as it helps identify critical areas of the application and points where bugs, errors, or potential vulnerabilities may hide. Additionally, JaCoCo introduces three important metrics, each concerning:

- **Line coverage:** shows the percentage of code executed based on the number of instructions called.
- **Branch coverage:** shows the amount of analyzed branches (e.g., if-else statements).
- **Cyclomatic complexity:** indicates the number of paths needed to cover all possible code paths through linear combination. If there is no if-else instruction in the code, the cyclomatic complexity will be 1, as only one path is required to inspect the code.

The experiment necessitates the configuration of **ARES** in order to work with **SCoPe**, which respectively implements the **SAC** Deep RL algorithm and the *pseudo-random* generation algorithm **Monkey**.

The study begins with the execution of **SCoPe**, which, after a total of *8000* events (about *50* minutes of session), generates an *.ec file* within a specific folder on the system.

```
end_coverage = "adb shell am broadcast -p #{package} -a
    intent.END_COVERAGE"
system(end_coverage, out: $outlog, err: $errlog)
# Create the full path based on the package name
folder_path = "path/to/pc/folder/#{package}"

# Create the target folder if it doesn't exist
Dir.mkdir(folder_path) unless File.directory?(folder_path)

# Execute the adb pull command with the updated path
pull_command = "adb pull
    /sdcard/Android/data/#{package}/files/coverage.ec #{folder_path}"
system(pull_command, out: $outlog, err: $errlog)
puts '[*] Performing cleanup'
```

Listing 7.15: Modifying SCoPe to generate code coverage at the end of a usage session

The subsequent analysis is performed using the *Deep RL* tool, **ARES**. After configuring the tool, terminal execution is initiated, and the testing phase starts with the following command:

```
python3 parallel_exec.py --instr_jacoco --list_devices "avd-name0
    ..." --appium_ports "4270 4277 ..." --android_ports "5554 5556
    ..." --path "apps" --timer 55 --emu headless --platform_version 1
    1.0 --iterations 10 --algo SAC --timesteps 1000 --trials_per_app
    3 --save_policy
```

The most crucial flag to consider for generating coverage files directly from the tool is `--instr_jacoco`, which develops a series of *.ec files* in a dedicated folder dynamically created in the ARES repository.

If one opts for a manual session of the application, exploring all its features by hand, the following terminal command should be executed at the end:

```
adb shell am broadcast -p <package.name> -a intent.END_COVERAGE
```

This command signals the end of code coverage data collection while the application is still running.

To generate the final coverage report:

1. **Retrieve the coverage files:** These are stored on the device at **/sdcard/Android/data/<package.name>/files/**. To transfer them to a

PC, execute the following terminal command: bash adb pull /sdcard/Android/data/<package.name>/files/coverage.ec path/to/dir

2. **Modify the jacoco-instrumenter-coverage.gradle file:** Replace the **path/to/coverage** in this file with the directory path where the coverage files are located. This gradle file resides in the same directory as the application's build.gradle.

3. **Generate the HTML report:** In the terminal, navigate to the instrumented application's folder and execute **./gradlew jacocoInstrumenterReport**. The HTML report is then generated in *build/reports/jacoco/jacocoInstrumenterReport/* directory of the apk.

The meticulous **configuration of the emulator** within Android Studio is pivotal for ensuring that the testing environment mirrors the operational conditions of an average Android device as closely as possible. Selecting specific parameters such as the allocated RAM, screen resolution, and the version of the Android operating system is instrumental in creating a realistic testing milieu. This setup is critical for the fidelity of the test results, as it reflects a user's typical experience on a contemporary Android device.

Moreover, the advantage of using an emulator is underscored in the context of testing flexibility. Unlike the complex and often cumbersome process of unlocking and rooting a physical device, the emulator streamlines this necessity, offering a straightforward path to obtaining root access. This accessibility is paramount for conducting in-depth, system-level testing, which is essential for the comprehensive evaluation of the application under test.

In the quest for consistency and reproducibility of the experiments, the **choice of the device** and the **Android version** for the emulator are executed precisely. Opting for the *Pixel 4 XL* running *Android 11.0* as the system image exemplifies a commitment to simulating a widely-used Android environment. This careful selection is intended to ensure that the experimental results are not only valid but also resonate with the prevailing user experiences, thereby enhancing the applicability and relevance of the study.

A critical phase in the setup is the **instrumentation of the APKs**, a process facilitated by the COSMO tool. This step involves inserting additional

code into the applications, enabling an exhaustive tracking of code coverage during subsequent testing. The initial stage of this process entails selecting appropriate open-source applications from the F-Droid repository, ensuring they meet COSMO's compatibility criteria, particularly concerning the Gradle version.

Once the applications are vetted and COSMO is configured, the source code instrumentation is conducted via a command-line interface. This preparation is vital for transforming the APKs into entities capable of providing a granular view of their interaction under various testing conditions, whether through automated tools like ARES and RPCDroid or through manual testing.

The final step in the preparation process is the **generation of the instrumented APKs**, for which Android Studio is employed, leveraging its user-friendly IDE environment. This phase involves importing the instrumented source code into Android Studio and proceeding with the APK generation, a process that is both straightforward and efficient in this context.

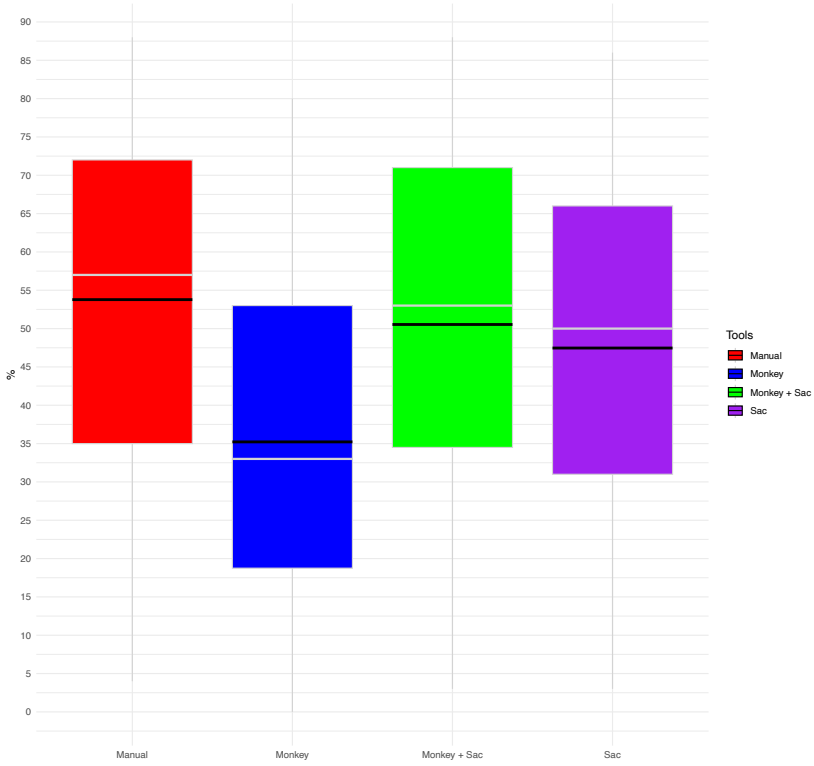
Utilizing Android Studio in this capacity simplifies the APK generation process, facilitating a smooth transition to the testing phase. Post-generation, the applications are primed for rigorous analysis, ready to be subjected to various testing methodologies aimed at uncovering their behaviors and interactions, thereby contributing to the overarching goal of enhancing Android application testing methodologies.

7.4.3 Analysis Results

The culmination of our dataset analysis has yielded convincing outcomes, illustrating that the use of automated tools can closely approximate manual testing efficacy. Specifically, the average coverage achieved through manual execution of apps hovers around *58%*. This benchmark means that manual execution, being the "optimal" reference, sets a standard against which the tool-induced outcomes are compared, aiming to closely match or exceed this threshold.

The Graph visually displays evaluations based on the methodology used. (Figure 7.13).

Figure 7.13: Boxplot illustrating various coverage levels categorized by the corresponding methodology



In the Graph 7.13, each boxplot contains two lines representing the *median* (gray line) and the *mean* (black line), respectively. The darker line illustrates the average code coverage trajectory across applications in relation to the executed tools, facilitating an accurate estimation of their performance.

- At the *bottom rank*, we observe an average *code-coverage* of *35.22%* during the application executions using the **Monkey** tool. Initially, Monkey might seem less efficient due to its reliance on *pseudo-random* inputs, which may not always align with the application's functional areas. However, it's crucial to acknowledge that such interactions can still contribute valuable insights into the application's overall performance analysis.

- In *second position* is the average *code-coverage* obtained using the **SAC** algorithm within the instrumented applications, achieving a mean of *48%*. One might anticipate that a Deep Reinforcement Learning algorithm would outperform a pseudo-random approach, yet a comprehensive evaluation of their dynamics is essential to fully appreciate their utility in application performance analysis.

- The *top position* showcases a combination of **Monkey + SAC**, where the integration of Monkey's sessions enhances the mean coverage to approximately *51%*. This increment indicates that Monkey's actions have contributed a *3 percentage point* increase over the performance achieved solely by SAC, underscoring the potential of pseudo-random actions to access functionalities or paths that might elude an AI algorithm.

In Table 7.3 are reported coverage values of executed apps. An observation regarding the automatic fuzzing tools is that, in some instances (42 out of 100), the combined approach (*Monkey + SAC*) rivals the code coverage of *manual execution*. Delving into the "index.html" files generated post-analysis, we sought to discern the reasons behind these instances where manual execution was outperformed, examining the code segments evaluated by the tools against those overlooked in manual execution.

Table 7.3: This table compares code coverage achieved by different algorithms for all 100 applications from the dataset. The value in *green* represents the highest code coverage achieved for each application. The equal coverage values are highlighted in *orange*.

Package name	MONKEY	SAC	MONKEY+SAC	MANUAL
AirMessage	0,03	0,09	0,1	0,04
Epistolaire	0,78	0,78	0,78	0,6
FlashAlert	0,51	0,51	0,51	0,5
Red	0,62	0,68	0,68	0,63
SimpleSmsReport	0,33	0,38	0,38	0,39

Table 7.3 continued from previous page

Package name	MONKEY	SAC	MONKEY+SAC	MANUAL
YetAnotherCallBlocker	0,38	0,38	0,5	0,5
Buses	0,18	0,19	0,2	0,3
DNS66	0,35	0,72	0,72	0,63
LavSeeker	0,07	0,65	0,65	0,78
TowerCollector	0,24	0,32	0,36	0,28
Block6	0,01	0,53	0,53	0,67
nextPush	0,56	0,68	0,7	0,75
Ulogger	0,23	0,38	0,4	0,55
Forkyz	0,35	0,47	0,48	0,5
RTranslator	0	0,03	0,03	0,1
valetudo	0,6	0,78	0,78	0,8
AirGuard	0,26	0,44	0,45	0,57
Toc2	0,57	0,76	0,8	0,87
Ojo	0,44	0,47	0,47	0,72
BlockPuzzle	0,01	0,04	0,05	0,35
Snapdrop	0,12	0,5	0,5	0,63
GpsdRelay	0,68	0,76	0,76	0,68
NextCloud Maps Client	0,67	0,75	0,78	0,8
Mastodon	0,07	0,14	0,16	0,15
Acastus	0,22	0,54	0,55	0,57
BikeComputer	0,04	0,58	0,58	0,72
BikeSharingHub	0,47	0,52	0,62	0,53
Brouter	0,02	0,06	0,06	0,17
Finder	0,1	0,13	0,15	0,23
Spp Terminal	0,53	0,8	0,85	0,88

Table 7.3 continued from previous page

Package name	MONKEY	SAC	MONKEY+SAC	MANUAL
FreeOTP	0,15	0,38	0,42	0,5
GeoNotes	0,44	0,43	0,6	0,72
Gpscockpit	0,63	0,75	0,78	0,85
GpsLogger	0,33	0,39	0,42	0,57
Bluepass	0,45	0,56	0,6	0,65
HardWhere	0,05	0,08	0,08	0,15
Infinity-for-Reddit	0,15	0,26	0,27	0,57
KeyLogger	0,25	0,44	0,52	0,72
KISS	0,35	0,42	0,43	0,62
NextCloud Service	0,45	0,47	0,5	0,57
Bustorino	0,15	0,24	0,27	0,63
MyGeo	0,33	0,64	0,64	0,8
Burger Party	0,25	0,53	0,55	0,7
Passegares	0,15	0,34	0,34	0,45
Satstat	0,25	0,34	0,43	0,3
SpiritPreise	0,2	0,18	0,24	0,28
UrlChecker	0,76	0,83	0,85	0,87
VilleChecker	0,35	0,47	0,51	0,45
WhereYouGo	0,13	0,23	0,24	0,27
Calls	0,6	0,78	0,78	0,6
Lexica	0,63	0,61	0,71	0,54
Gridle	0,66	0,67	0,74	0,7
Birday	0,27	0,43	0,43	0,45
BirthDayDroid	0,12	0,18	0,18	0,17
ShareX	0,37	0,43	0,46	0,75

Table 7.3 continued from previous page

Package name	MONKEY	SAC	MONKEY+SAC	MANUAL
ToDOAgenda	0,34	0,28	0,35	0,35
Lametric	0,44	0,5	0,51	0,57
Deepl	0,6	0,78	0,78	0,78
Grocery	0,3	0,61	0,67	0,63
GetFlow	0,35	0,51	0,55	0,47
Wasgeht	0,44	0,58	0,59	0,75
Sensor Server	0,3	0,54	0,54	0,5
Warpclock	0,22	0,25	0,25	0,21
DebateKeeper	0,34	0,82	0,82	0,87
ToDoo	0,31	0,3	0,35	0,28
CalendarImportExport	0,14	0,14	0,14	0,14
CleanTimer	0,75	0,44	0,75	0,72
Amarok Rider	0,63	0,8	0,81	0,75
Child Monitor	0,55	0,78	0,8	0,83
FourthTools	0,76	0,69	0,78	0,76
Better Internet Tiles	0,25	0,53	0,57	0,55
GoodTime	0,19	0,32	0,33	0,52
BG2Bangle	0,45	0,57	0,6	0,77
Just24Hours	0,06	0,11	0,11	0,08
Crossword	0,33	0,42	0,42	0,5
DarkModeWallpaper	0,8	0,76	0,8	0,8
Meditation Assistant	0,24	0,36	0,38	0,21
Vaches de France	0,65	0,83	0,83	0,87
Mifare Classic Tool	0,45	0,67	0,7	0,82
Octal Suntime	0,24	0,25	0,25	0,23

Table 7.3 continued from previous page

Package name	MONKEY	SAC	MONKEY+SAC	MANUAL
Open Time Limit	0,06	0,03	0,06	0,07
Dcnnt	0,47	0,49	0,55	0,7
Wlan Scanner	0,23	0,25	0,25	0,28
Editor	0,25	0,43	0,45	0,5
ShiftCall	0,18	0,57	0,71	0,54
SleepTimer	0,62	0,6	0,67	0,84
Snap Streak	0,53	0,55	0,58	0,51
Super Productivity	0,58	0,75	0,75	0,58
Time Clock	0,37	0,7	0,72	0,7
Kwik EFIS	0,52	0,54	0,56	0,68
Thor	0,06	0,12	0,15	0,2
WallME Wallpaper	0,08	0,08	0,08	0,1
Rethink	0,23	0,54	0,57	0,65
WorldWeather	0,24	0,3	0,31	0,28
ZipXtract	0,6	0,53	0,6	0,6
Profile	0,21	0,81	0,81	0,75
TaxiCat	0,69	0,86	0,88	0,84
ChipDefense	0,6	0,75	0,78	0,75
FindMyDevice	0,15	0,26	0,29	0,3
SSTVEncoder	0,51	0,52	0,71	0,67

A significant case involved the **Clean Timer** application, version 1.1. As visible in Figure 7.14, the tools detected a 'swipe' feature activated by a left-to-right gesture, but this instruction was missed in manual execution, improving the code coverage achieved by Monkey (in *yellow* in Table 7.3). Such findings

highlight the capabilities of automated tools in identifying interactive elements that might not be immediately apparent to human testers.

```
280.            @Override
281.            public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY) {
282.                final int SWIPE_MIN_DISTANCE = 40;
283.                final int SWIPE_THRESHOLD_VELOCITY = 100;
284.
285.                if(e1.getX() - e2.getX() > SWIPE_MIN_DISTANCE && Math.abs(velocityX) > SWIPE_THRESHOLD_VELOCITY) {
286.                    return false; // Right to left
287.                }  else if (e2.getX() - e1.getX() > SWIPE_MIN_DISTANCE && Math.abs(velocityX) > SWIPE_THRESHOLD_VELOCITY) {
288.                    return false; // Left to right
289.                }
290.
291.                if(e1.getY() - e2.getY() > SWIPE_MIN_DISTANCE && Math.abs(velocityY) > SWIPE_THRESHOLD_VELOCITY) {
292.                    startActivity(new Intent(MainActivity.this, stats.class));
293.                    return false; // Bottom to top
294.                }  else if (e2.getY() - e1.getY() > SWIPE_MIN_DISTANCE && Math.abs(velocityY) > SWIPE_THRESHOLD_VELOCITY) {
295.                    startActivity(new Intent(MainActivity.this, stats.class));
296.                    return false; // Top to bottom
297.                }
298.                return true;
299.            }
300.        }
```

Figure 7.14: Code snippet from the **Clean Timer** application analyzed in a manual session. The red section indicates the unexecuted *swipe* functionality.

These insights underscore the complementary nature of random tools and ARES, suggesting an improvement in approaches where both methodologies converge to enhance code coverage and application understanding.

In the end, this analysis of open-source applications underscores the challenges in achieving 100% code coverage. While reaching this benchmark may seem daunting, it's essential to recognize that these constraints are often rooted in the application's source code rather than the limitations of the testing tools employed.

The potential obstacles encountered in these 100 apps in maximizing code coverage include:

- **Easter Eggs:** These are hidden features that users might stumble upon during their interaction with the app, typically accessed through specific sequences of actions or interactions within the UI (see Figure 7.15). Such undocumented features pose a significant challenge not only for human testers but also for automated analysis tools, as their discovery relies on serendipity or inside knowledge.

- **Inactive Code Bundles:** Developers might incorporate code segments that are activated only under certain conditions, like using discount codes in an online store. Activating these segments requires specific knowledge or actions that may not be part of the standard testing process.

- **Unused Code:** Over the development lifecycle, some initially planned functionalities may not be integrated into the final user interface but remain in the codebase. These remnants, unless explicitly removed, contribute to the unreachable code during testing by both automated tools and manual efforts.

These examples illustrate why achieving maximum code coverage can be challenging. However, automated analysis tools can potentially explore areas of the code or functionalities that might be difficult or even impossible for a manual user to access. For instance, automated tools can systematically test various input combinations and explore different app states that a human tester might not consider, thereby uncovering hidden features or triggering unused code paths.

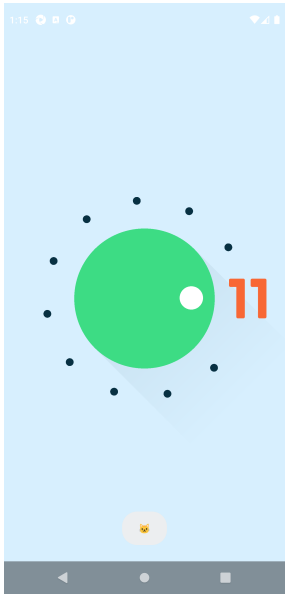


Figure 7.15: Example of an Easter egg in the Android Operating System version "Red Velvet Cake," accessible by tapping the Android version in the settings eight times.

7.4.4 Final remarks

This research provides a series of insightful results derived from the application of two algorithms, SAC and Monkey, which automate the analysis of Android applications. Initially, the study tackled a comparative analysis between SAC and Monkey to ascertain which algorithm demonstrates superior performance in terms of code coverage over an hour-long analysis session.

The investigation then delved into the hypothesis that a synergistic combination of these two algorithms could enhance code coverage performance. This premise is rooted in the observation that a pseudo-random analysis algorithm like Monkey might succeed in probing code segments and executing app functionalities that a Deep Reinforcement Learning algorithm like SAC might overlook.

To ensure the robustness of our findings, the outcomes from these automated analyses were benchmarked against the coverage achieved through a manual testing session. The aim here was to explore as many app functionalities as possible, assessing whether the performance of automated analysis could parallel or even surpass that of manual scrutiny.

In conclusion, this study opens an intriguing perspective on future development, focusing on optimizing the integration between automated and manual analysis. The union of these two approaches has already evidenced its merit in optimizing code coverage. To further bolster this synergy, exploring practical strategies for integration is recommended.

The ultimate goal is to standardize this combination, making it an efficient practice capable of enhancing code coverage from the early stages of the automatic testing process. Moreover, it would be intriguing to investigate how the combined use of dynamic analysis and machine learning algorithms could contribute to a deeper understanding of code vulnerabilities, leading to the creation of safer and more reliable applications.

In light of this study, improving input generation in SCoPe by utilizing ARES, especially in conjunction with Monkey, has proven to be extremely effective. This improvement is particularly notable in the enhanced code coverage observed, showcasing the potential of these integrated approaches in conducting a more exhaustive application analysis.

7.5 Evaluating the Efficacy of SCoPe Automated Context Identification Against Manual Analysis in Android Apps

This chapter exhaustively investigates the efficacy of SCoPe's automated context identification within Android applications. It specifically examines how SCoPe's contextualization methodology, already explained in Section 7.2.3, performs against meticulous manual analysis when using ARES + Monkey as input generator.

Contextualization in Android app analysis is extremely important for understanding how permissions are used within specific user interactions. The complexity and dynamism of modern apps necessitate sophisticated tools that can accurately identify and differentiate these contexts. This study scrutinizes the precision of SCoPe's automated context identification, enhanced by the input generation capabilities of ARES and Monkey, and juxtaposes it with the gold standard of manual analysis.

7.5.1 Experimental Design

The overarching research question guiding this study is:

RQ_2 How does the accuracy of SCoPe's automated context identification in Android app permission usage compare with an in-depth manual analysis, particularly when enhanced by ARES + Monkey as input generation tools?

This question delves into the effectiveness of SCoPe's contextualization approach for identifying permission usage in Android applications. We aim to compare SCoPe's automated context identification against a rigorous manual analysis, specifically evaluating the impact of incorporating ARES and Monkey input generation techniques. Our experimental setup includes two phases:

Phase 1: Open-Source Application Analysis

App Selection: We revisited a subset of 34 open-source applications utilized in our previous analysis in Section 7.4. These applications were chosen due to their availability of source code, enabling code instrumentation for coverage measurement.

Code Instrumentation: We employed Cosmo and Jacoco to instrument the source code of the selected applications. This instrumentation allowed us to obtain code coverage data, a crucial metric for evaluating the effectiveness of SCoPe's context identification.

SCoPe Execution with ARES+Monkey: We executed SCoPe on the instrumented applications, utilizing ARES and Monkey as input generation techniques. This combined approach aimed to maximize code coverage and provide a comprehensive evaluation of SCoPe's contextualization capabilities.

Manual Analysis: Two experienced investigators conducted a manual analysis of the generated SCoPe reports. This analysis involved examining log events, permission usage logs, screenshots, and video recordings of app execution. The goal was to validate the accuracy of SCoPe's identified contexts.

Phase 2: Commercial Application Analysis

App Selection: We expanded our analysis to include 64 commercial applications downloaded from the Google Play Store. These applications were chosen based on their popularity and representativeness of real-world usage scenarios.

SCoPe Execution without Code Instrumentation: Due to the absence of source code for commercial applications, we executed SCoPe without code instrumentation. However, we maintained the use of ARES and Monkey for input generation.

Manual Analysis: Similar to Phase 1, two investigators conducted a manual analysis of the generated SCoPe reports. This analysis involved ex-

amining log events, permission usage logs, screenshots, and video recordings of app execution.

7.5.2 Results

Leveraging ARES and SCoPe, particularly their integration, enabled us to conduct an in-depth empirical study, thoroughly analyzing the dataset of apps under consideration. By comparing the various contexts identified by SCoPe during the analysis sessions with those validated through manual verification, we concluded that the technique used by SCoPe for identifying usage contexts has remarkably high *accuracy*, reaching **84%**. This accuracy rate was achieved by comparing the contexts automatically identified by SCoPe against those manually validated through a rigorous manual analysis process. Out of **435** unique contexts identified automatically across all applications by SCoPE, **365** were correctly identified based on our manual investigation. This high level of accuracy provided a reliable foundation for our subsequent analysis of *sensitive resource access* within these apps.

Sensitive Resource Access

Building on the reliable context identification provided by SCoPe, we conducted a detailed analysis of how apps access sensitive resources. We uncovered that the majority of the apps analyzed during the study access sensitive resources at least once.

In more detail, out of 34 open-source apps, 28 requested access to sensitive resources, of which 17 did so in at least two contexts, while 16 accessed particularly critical permissions (such as camera, microphone, and GPS), of which 12 did so in more than one context. Conversely, regarding commercial apps, out of 64 apps, 56 requested access to sensitive resources, 42 of which did so in at least two different contexts, while 53 accessed the camera, microphone, and GPS, 39 of which did so in multiple different contexts. This study highlights that using a critical resource in two or more different contexts is a widespread practice among mobile app developers. In some cases, this could pose serious privacy risks for users who interact with these applications on a daily basis. Additionally, the

results suggest that in open-source applications, the use of sensitive resources in more than one context occurred in a lower percentage compared to apps for which the source code was not available.

Impact on User Experience

In some cases, one might think that a permission model based on SCoPe could negatively impact Android users' user experience. However, this is not the case because the tool has proven to be highly efficient in identifying the various usage contexts in most of the apps studied. This would mean that explicit user authorization would only be requested if an attempt is made to access that resource in a context different from those previously identified.

The most striking example is provided by the study of the EZVIZ app. During the usage session, the device's camera was accessed 58 times, but only 5 different contexts were identified by SCoPe. Therefore, permission to access the camera would have been requested only 5 times. This would impact the user experience to a much lesser extent than an approach that requires permission for each use of the resource.

The same efficiency in detecting different contexts without impacting the user experience can also be observed during the analysis of SHAZAM, where the automated execution of the app led to accessing the microphone 42 times, but only in 6 different contexts:

Even in this case, an SCoPe-based permission model would have interrupted the user only 6 times, instead of 42.

7.5.3 Conclusion

The presented research delved into the effectiveness of SCoPe, an automated tool for identifying usage contexts in Android app permission requests. By integrating SCoPe with ARES and leveraging Monkey for input generation, we conducted a comprehensive empirical study on a dataset of 98 Android applications, encompassing both open-source and commercial apps.

Our findings demonstrate that Scope exhibits remarkable accuracy in identifying usage contexts, achieving an impressive 84% accuracy rate. This high

accuracy is attributed to Scope's ability to capture contextual information from various sources, including UI interactions, permission usage logs, and screenshots.

Furthermore, our analysis revealed that the majority of the apps analyzed accessed sensitive resources at least once. This highlights the prevalence of permission requests and the need for robust permission models to safeguard user privacy.

Interestingly, we observed that open-source apps exhibited a *lower* tendency to access sensitive resources in multiple contexts than commercial apps. This could be attributed to the increased transparency and community scrutiny associated with open-source development.

Regarding user experience, our study suggests that SCoPe's efficient context identification minimizes interruptions and maintains a positive user experience. By only requesting explicit user authorization when accessing resources in contexts different from those previously identified, SCoPe strikes a balance between security and usability.

Overall, the results of our study underscore the effectiveness of SCoPe as a promising tool for enhancing Android permission models and safeguarding user privacy. Its high accuracy, broad applicability, and minimal impact on user experience make it a valuable addition to Android applications' arsenal of security measures.

CHAPTER 8

Beyond Detection: Using SCoPe to Defuse Logic Bombs in Android Environments

Contents

8.1 Introduction 164
8.2 Experimental Design 168
8.3 Evaluation 170
8.4 Conclusion 173

8.1 Introduction

One of the most significant threats to the security of Android applications is *logic bombs* [128]. A logic bomb is a type of malicious code that is triggered by specific events or conditions. When activated, a logic bomb can execute various harmful actions, such as stealing sensitive data or crashing the system. Anti-malware engines regularly flag thousands of apps. Actually, the AndroZoo [129]

repository has collected over 22 million malicious app versions, among which over 19 million were available on the Google Play Store. Therefore, addressing the spread of malware in app markets is a prime concern for researchers and practitioners.

Unfortunately, automatically detecting logic bombs is still an open issue due to several pitfalls, such as the high rate of false positives since benign and malicious apps can use the same code for benign and/or malicious behavior, thus requiring an analyst to verify the behavior [88]. To address this issue, there is a need for a more context-aware approach to permission control in Android applications. As already extensively explained in Chapter 7, context awareness refers to an application's ability to adapt its behavior based on the current use context.

In recent years, there has been growing interest in using dynamic analysis techniques for detecting logic bombs in Android applications. This is mainly due to the fact that static analysis can be deceived by different obfuscation schemes [130]. Dynamic analysis refers to the analysis of an application's behavior during runtime. Dynamic analysis techniques have been used in various security contexts, including detecting malware and identifying vulnerabilities in web applications [131][132][133].

However, approaches resilient to various logic bomb analysis techniques, including fuzzing, symbolic execution, multi-path exploration, and program slicing, have been proposed [87].

In this work, we adopt a complementary approach in which, admitting the possibility that an app contains a logic bomb, we monitor the resources accessed by the app and regulate their access based on the usage context. Thereby, we are not affected by the false positive problem. It is worth noting that our approach does not aim to provide a solution to the problem of logic bombs. Instead, we address a scenario in which an untrusted app and malware that deceived a bomb analysis tool are executed. In particular, we do not aim to identify the presence of the logic bomb but to prevent it from causing access to protected resources unbeknownst to the user. With the term protected resources, we refer to all the device features associated with potentially sensitive data leakage, which Android protects by requiring access to such dangerous permissions. If the app has not been

granted such permission, it cannot access the resource. Examples of protected resources are: camera, microphone, contacts, SMS, location, and many more.

Although Android has improved the permission management system over the years, it still suffers from some limitations that allow a logic bomb to access, under certain conditions, these resources without the user's awareness. One of the mechanisms that a malicious app can easily implement to bypass Android's protection is to hide the malicious behavior within apps that, due to the functionality they offer, legitimately have access to one or more of these protected resources. Once access permission is obtained, the app triggers the logic bomb without the operating system, deeming it necessary to request further confirmation.

Clearly, logic bombs are designed to trigger this behavior only if the user has previously permanently granted access to the resource they want to exploit so that they do not betray their presence by trying to access unauthorized resources. Android allows the user to specify whether access to a resource should be limited to a single execution or whether it is persistent. However, it is very likely that if the legit functionality is frequent – sooner or later – the user will decide to grant the permission permanently. In such a circumstance the logic bomb would be triggered.

To mitigate this problem, we propose to use SCoPe (Chapter 7) as an extension of the Android permissions model in which the authorization to access a protected resource is associated with the execution context. By execution context we mean a combination of the functionality the app is involved in and the user interface element the user interacted with. There are several ways to identify the execution context. For instance, it is possible to analyze the calling context, i.e., the list of active functions currently on the call stack [134], use the calling context encoding [135], the execution index [136], as well as techniques to recording and reporting dynamic calling contexts [137]. By adopting the calling context as a new dimension of the permission model, the user can continue using the functionality to s/he has already granted permissions without the burden of repeatedly allowing access to the same resource. At the same time, any access to the same protected resource occurring in contexts different from already authorized ones will require explicit authorization. As a result, the logic bomb will not trigger if the context has not been previously analyzed. On the other hand, if the logic bomb was

designed to trigger independently of the context, e.g., because it is unaware of the modified permission model, access to protected resources would have to be explicitly granted by the user. This behavior would be an alarm bell for the user, who could identify it as malicious, thus defusing or mitigating the bomb effect.

In this work, to identify the calling contexts and instrument the context-based permission model, we extend SCoPe (Chapter 7). ScoPe is designed to monitor any access to protected resources at runtime by tracking the invocation of sensitive APIs and system services and logging useful information to determine the context in which a protected resource is used. Note that these are only a subset of the possible execution contexts, namely those involving the usage of protected resources. A logic bomb not using such resources could be triggered without being intercepted. As said before, this study merely aims to avoid this kind of malicious behavior. Clearly, the approach could be extended to include more sophisticated context identification approaches and different types of resources currently not protected by dangerous permissions.

We used SCoPe both to enforce the permission model with context-aware access control policies and to better understand logic bomb common behaviors, further improving the tool's ability to detect anomalies in accesses to protected resources. By analyzing the logs generated by SCoPe, we can indeed identify the exact conditions under which a logic bomb is triggered, providing a better understanding of its behavior and how it is activated.

To assess the effectiveness of our proposal to identify when a logic bomb is triggered and control the access to protected resources, we used TriggerZoo [138], a publicly available dataset containing apps with injected logic bombs. We manually executed a subset of the apps and, using the indications provided in the dataset, we managed to invoke the app features that contain the triggers for the logic bombs. The achieved results confirm that by using a context-aware permission model leveraging dynamic analysis techniques, we can prevent sensitive data from being leaked without the knowledge of the user, even in the case of a logic bomb.

The rest of the Chapter is organized as follows. Section 8.2 briefly describes the context-aware permission model implemented with SCoPe and the Experiment design. In Section 9.3 we provide a preliminary evaluation of the effectiveness

of our proposal using TriggerZoo. Section 8.4 concludes the work and gives indications for future work.

8.2 Experimental Design

Our work is based on the premise that a logic bomb is triggered by a specific sequence of events, which can be identified by analyzing an application's runtime behavior. To implement our approach, we extended the Android permission model by considering information to limit access to protected resources to specific contexts. This offers several advantages over existing approaches. It allows us to identify context-specific behavior, reducing the risk of false positives. Obviously, the drawback of this approach is that we rely on the user's ability to discern between legitimate and malicious behaviors.

We used the existing dynamic tool SCoPe, which was designed to monitor the execution of mobile applications that access specific device resources. SCoPe prompts the user for explicit permission and saves the decision. SCoPe is installed as an Xposed module on the Android device or emulator and collects information to identify resource usage contexts. Upon starting the application, SCoPe tracks any access to protected resources (e.g., camera, microphone, storage, or location) at the system level. We monitor any action performed on the user interface and any system call to the Android access control and validation mechanism. Chapter 7 provides more details on the tool. To prevent some logic bombs from triggering due to permission denial, we automatically granted all the permissions declared in the Manifest file to set the tool in the worst scenario. Logic bombs are usually designed to trigger only if the user has previously granted access to the resource they want to exploit, so as not to betray their presence. By granting permissions outside the trigger code, we simulated a situation where the logic bomb is free to trigger at any time. Currently, the system identifies a context based on actions on the UI (e.g., touching a button or changing activities) and permissions requested at runtime. It is worth noting that the approach can be enhanced to include more restrictive or exhaustive execution context identification approaches.

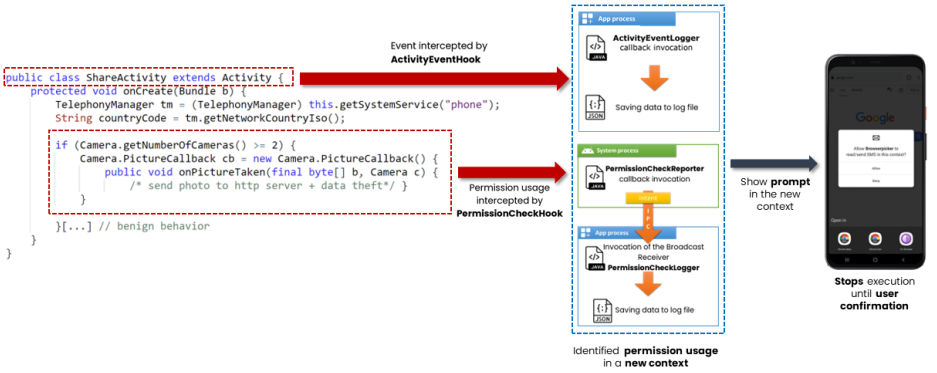


Figure 8.1: The approach *dynamically identifies* individual permission usage contexts, showing the user the *decision prompt*.

The proposed approach is depicted in Figure (Figure 8.1). SCoPe uses the Xposed framework to inject the services we developed dynamically. These services perform method hooking concerning invocating access permissions and use callbacks to inject code that dynamically displays the permission management prompt. Finally, we monitor the app's behavior in the application process by intercepting the calls to access permissions associated with protected resources. The tool uses this information to contextualize the permission requests and decide whether to display the appropriate prompt to the user. Note that the user's contextualized choice can be saved and used in case of further requests, thus avoiding the need to defuse the same bomb several times. Similarly, for each resource, the impact on benign apps is limited by the number of different contexts exceeding the first. To evaluate the effectiveness of SCoPe context-aware permission model in preventing a logic bomb from actually accessing sensitive data without the knowledge of the user, we executed it on a dataset of Android apps called *TriggerZoo* [138], provided by Androzoo [129]. This dataset contains 406 apps with **injected logic bombs** and includes descriptions of the logic bomb activation (e.g., the activity in which it is activated, permissions used, and return values). We executed the tool on an Android 11 (API 30) emulator with SCoPe active. We randomly selected a subset including 70 apps, executed them using the SCoPe monitoring tool, and

conducted an *in-depth analysis of their behavior* to understand the logic bomb triggering process and its relationship with permission misuse.

8.3 Evaluation

In Table 8.1 we report the results of our study. In particular, 45 of the selected apps could not be executed due to various reasons, while for 25 of them, we were able to reproduce all the steps needed to test the trigger condition.

Category	**Number of Apps**
Total apps analyzed	70
Non-executable apps	45
Dataset related issues	36
Incorrect permission declaration	10
Incompatible with Android version	6
Other errors	20
SCoPe related issues	9
Executable apps	25
SCoPe mitigation successful	16
Trigger condition not met	4
App closed by logic bomb	3
SCoPe mitigation failed	2

Table 8.1: Summary of the analysis results for the 70 apps

Amongst the reasons that prevented the app from being executed without crashing, we mention *issues in the repackaged application*, *incorrect permission declarations within the Manifest*, and *incompatibility with the emulator's Android versions* (i.e., Android 11). Specifically, 36 apps were not executable due to dataset-related issues such as corrupted APKs. We verified this issue by attempting to execute the apps on different emulators with various Android versions. We ruled out the possibility that SCoPe caused the application crash by executing the files on different emulators without the tool and experiencing the same issues. Among these 36 apps,

10 had errors related to the incorrect injection of permission declaration within the Manifest by TriggerZoo. For example, in the XML reported in Listing 8.1 instead of declaring the permission as `android.permission.ACCESS_FINE_LOCATION` it has been declared as `Manifest.permission.ACCESS_FINE_LOCATION`. In our emulator, this caused the app to crash as soon as the permission was requested, as Android did not recognize it.

Listing 8.1: Example of incorrect permission declaration in the manifest

```
<manifest
 xmlns:android="http://schemas.android.com/apk/res/android"
 package="com.prueba.joel.sort">
 <uses-permission
  android:name="Manifest.permission.ACCESS_FINE_LOCATION"/>
<uses-permission
  android:name="Manifest.permission.ACCESS_COARSE_LOCATION"/>
<uses-permission
  android:name="android.permission.INTERNET"/>
</manifest>
```

Furthermore, 6 of the 36 apps were incompatible with the Android version used in our analysis due to the outdated target SDK used to build them. The remaining 20 apps had issues with method calls and native code or were improperly recompiled, causing them to crash upon startup. The TriggerZoo authors acknowledged that such issues might be possible. Finally, 9 of the apps failed to execute when SCoPe was active. These were hybrid apps that, for some reason, attempted to use permissions not declared in the Manifest. Since SCoPe sets all manifest permissions to "Granted" during the analysis, using undeclared permission caused our tool to throw an exception and terminate the app execution. It is important to note that such undeclared permissions are no longer allowed in recent versions of Android, so this issue would not exist in a real setting.

Considering the 25 apps that were correctly executed on the emulator, we *manually executed* them while SCoPe monitored the running session. In 16 of them, the tool displayed a **prompt asking us to allow or deny using a sensitive resource** precisely when the trigger event that would activate the logic bomb occurred. As a result, the app execution was paused until we decided on

permission use in that specific context. It is worth noting that the context in which the tool requested resource usage was *off-topic* compared to the requested resource. This is consistent with what we expected from malicious behavior. For example, a request happened in a context where the app displayed a list of images to set as the device's wallpaper. When we pressed the button to set the wallpaper, the app requested permission to send/read SMS. User should naturally deny this usage, preventing the trigger event from activating. To further evaluate whether that was indeed a trigger event, we retraced our steps through the same context while confirming the permission use this time. Through the log analysis, we identified that the event was, in fact, the trigger for a logic bomb. It is important to emphasize that such behavior occurred consistently across all the analyzed apps.

In addition to the 16 apps mentioned earlier, we analyzed four other applications in which the trigger condition for the logic bomb could not be activated due to the design of the apps. Specifically, one of these apps required authentication through a login form whose information was not provided in the dataset description. Two apps were designed to activate the logic bomb only if the device met certain requirements, such as specific IMEI or model information which, unfortunately, are limited on the emulator. Lastly, one of the apps is meant to activate the logic bomb at a specific time of a particular day. By analyzing the behavior of these apps as described by the authors of Triggerzoo, we are confident that, had we met the trigger condition, SCoPe would have correctly presented the user with a prompt to block or allow the use of the protected resources requested by the logic bomb.

For three applications, once the trigger condition for the logic bomb was satisfied, the bomb's malicious behavior involved closing the application, but our tool did not display any prompt. Although in these cases the code behind the trigger was executed without the user being able to prevent it, it should be noted that our approach is specifically designed to control access to protected device resources, so we expected that this behavior would not have been intercepted by the tool as it is not associated with a permission in Android. Similarly, for two applications, the tool was unable to mitigate the logic bomb's behavior since the malicious behavior did not involve using dangerous permissions.

It is worth noting that when running the apps manually, requests for permissions were almost exclusively made during the activation of a logic bomb. This is probably also due to the limited complexity of the apps in the dataset which do not include many features. However, the mechanism for detecting the execution context and monitoring only dangerous permissions has effectively limited repeated requests to the end user. Moreover, from the analysis of the SCoPe logs it is also possible to extract common patterns in the use of permissions by applications containing logic bombs. This could further improve the tool's ability to intercept the most dangerous requests in a more precise manner, potentially providing suggestions to direct the user in the decision to grant or deny a permission request.

To conclude, in our analysis of 25 apps, the SCoPe tool was effective in 16 apps. The prompt was shown to users, allowing them to deny access to protected resources requested by the logic bomb. The trigger condition could not be activated in four apps, so we could not assess the approach against them. In the rest of the apps, the tool could not mitigate the logic bomb behavior due to the lack of permission usage during the malicious action. Still, no privacy violations could occur in similar situations. This is a remarkable result concerning the overall effectiveness of the context-aware fine-grained permission model in defusing logic bombs or mitigating their impact on user privacy. The system's ability to associate UI actions and permission requests to specific contexts allowed for a more comprehensive understanding of the application's behavior and the potential risks.

8.4 Conclusion

In this work, we addressed logic bombs, a type of malicious code that is triggered by specific events or conditions. We proposed a dynamic approach to avoid privacy leakage in case the malicious code is activated. In particular, we aim to identify execution contexts related to protected resources and prevent unauthorized access to privacy-related data through logic bombs.

We assessed the effectiveness of our proposal with TriggerZoo [138], a publicly available dataset containing apps with injected logic bombs. The results are preliminary but promising. The approach allowed us to prevent the execution of

malicious code behind a logic bomb trigger in all situations in which the code actually tried to access a protected resource. We also collected logs generated by SCoPe to identify the exact conditions under which a logic bomb is triggered and permit a better understanding of its behavior. This gave us important insights into the behavior of logical bombs while accessing privacy-related resources that we plan to use to further improve the ability to automatically distinguish legitimate and malicious behaviors, reducing the effort required by the user for decisions regarding dangerous permissions.

The tool used to assess the proposed approach is a proof-of-concept. It requires root access to the mobile device and introduces a processing overhead making it unsuitable for a real context. However, it is desirable that in the future, a permission management aware of the execution context and the usage pattern of dangerous permissions will be integrated into the operating system. In this way, the approach could actually contribute to the overall security of the Android ecosystem and protect user privacy and sensitive data even in case of security threats like logic bombs, notoriously difficult to detect.

CHAPTER 9

Assessing the Effectiveness of an LLM-based Permission Model for Android

Contents

9.1 Introduction 175
9.2 Experimental Design 177
9.3 Analysis results 197
9.3.1 Analysis of Permission Usage in Android Applications 197
9.4 Assessing Effectiveness on Malicious Apps 201
9.4.1 Experiment 207
9.5 Final remarks 208

9.1 Introduction

While SCoPe excels in contextual permission requests, a single step currently requires user intervention: granting permission itself. Actually, the user must

manually consent to the use of permissions, although these requests are made contextually and accompanied by detailed information. This approach has two weaknesses: first, it can lead to 'user fatigue,' where users may tend to accept permission requests simply because they fit within the provided solution; second, users may lack the necessary skills to distinguish between safe and dangerous contexts, potentially compromising their ability to make informed decisions. This knowledge gap can lead to inadvertent exposure of personal data and potential privacy breaches. For this reason, a possible solution is to automate this process.

This chapter explores the potential of Artificial Intelligence (AI), particularly Large Language Models (LLMs), to address this critical challenge. Our proposed approach leverages AI models capable of analyzing an application's user interface (UI) to assess permission requests and provide context-aware recommendations to users dynamically. Unlike traditional machine learning models that offer only binary classifications (allow/deny), LLMs can generate explanations alongside their recommendations. These explanations, presented as user-friendly prompts, empower users to understand the rationale behind the AI's assessment and the potential risks associated with granting specific permissions. The proposed system offers several advantages over existing permission management solutions. First, it fosters user awareness and education. By providing clear explanations for permission requests, the system fosters a deeper understanding of the implications of granting access to sensitive data and resources. This can empower users to make more informed decisions beyond simply relying on intuition or default settings. Second, the system leverages the dynamic nature of LLMs to personalize recommendations. Factors such as user preferences, app context, and current security trends can be incorporated into the assessment, ensuring that users receive tailored guidance specific to their situation. For instance, an LLM might recommend denying access to location data for a seemingly innocuous game app but suggest granting permission for a navigation app, considering the differing contexts and security implications. Third, the system can continuously learn and improve through user feedback and integration of security research findings. By incorporating user interactions and insights from the evolving threat landscape, the system can adapt to remain effective against emerging privacy and security risks. Integrating explainable AI into permission management represents a significant

step towards a more user-centric approach. This collaborative effort between AI researchers, technology developers, and end-users can lead to developing a secure and privacy-conscious mobile ecosystem.

To assess GPT-4's ability to analyze permission appropriateness, we adopt an experimental design that presents the model with screenshots of app UIs and relevant log information from SCoPe. We utilize OpenAI's APIs to interact with GPT-4-turbo and DALL-E, employing carefully crafted prompts to guide the model's analysis.

Through a series of iterative experiments, we refine our experimental procedure and optimize the prompts to maximize GPT-4's effectiveness in identifying appropriate and inappropriate permission requests. Our findings provide valuable insights into the potential of LLMs like GPT-4 to enhance context-aware permission management and empower users to make informed decisions about granting access to sensitive resources.

The remainder of this chapter is structured as follows. First, we describe our experimental design, including the dataset, prompts, and evaluation metrics. Then, we conducted a further study on malicious app, leaving the experimental design unchanged. Subsequently, we present the results of our experiments and their implications. Finally, we conclude the chapter by summarizing our key contributions and outlining directions for future research.

9.2 Experimental Design

In our experiment, we utilized SCoPe (Chapter 7), particularly its outputs and the dataset of app executions presented in the previous chapter, to analyze with GPT-4. This analysis leveraged the model's capability to analyze images and generate insights from them. We operated under the assumption that the only information available to a user to determine whether it is appropriate to use a sensitive resource, such as a camera or microphone, is the screen on which this resource will be used. Hence, we queried the model to show whether an LLM could determine the appropriateness of using sensitive resources based on an image representing an app's usage screenshot and the information in SCoPe logs. To do this, we employed OpenAI's APIs to query gpt-4-turbo and DALL-E using

various prompts. During the analysis, a series of experiments were conducted to refine the experimental procedure and the prompts used.

SCoPe outputs: In Scope's architecture, each application is organized through several folders that reflect specific aspects of its operation. Of these, the following folders are of particular relevance to our analysis:

1. Events: Contains one or more files in JSON format that document the set of events handled by the application. These events can include state changes, interactions with UI components, and other significant actions as in Listing 9.1. For our study, we will focus on:

   (a) **elementId**: Uniquely identifies the application screen active at the time of the event, providing a direct link to the screen displayed.

   (b) **eventType**: Classifies the event according to its nature. Possible values are "TAP," indicating a selection by the user that triggered the next screen, and "STARTED," signaling the opening of a screen without prior interaction.

   (c) **type**: Describes the item that, when selected, triggered the event, offering details about the action that led to the screen activation.

2. Permissions: Encapsulates one or more log files, also in JSON format as shown in Listing 9.2, that list the permissions requested or used by the application. These files create a direct link between the documented events and the specific permissions used for each, allowing detailed analysis of the use of permissions in relation to the logged events. An event can trigger one or more permissions.

3. Screenshots: Each event or permit request is accompanied by a screenshot in PNG format. Screenshots of events are associated via the elementId, while a timestamp identifies those of permit requests. This visual documentation system provides an immediate and contextualized representation of application interactions and state changes.

.

```
{

  "applicationPackage" : "com.shazam.android",
  "logStartTime" : "2023-10-10T17:51:29.236",
  "elements" : [
      {...},
      {
        "@class" : "it.unimol.rpcdroid.log.elements.
    ViewEventLogElement",
        "timestamp" : "2023-10-10T18:03:07.175",
        "elementId" : 2037511879,
        "eventId" : 1727609034,
        "content" : null,
        "description" : "Cancel identifying song",
        "eventType" : "TAP",
        "parentActivity" : "j.e",
        "tooltipText" : "Cancel identifying song",
        "type" : "androidx.appcompat.widget.z",
        "viewId" : -1
      },
      {...}
    ]
}
```

Listing 9.1: Event file json

```
{
  "applicationPackage" : "com.shazam.android",
  "logStartTime" : "2023-10-10T17:51:29.236",
  "elements" : [
        {...},
        {
        "@class" : "it.unimol.rpcdroid.log.elements.
    PermissionCheckLogElement",
        "permission" : "android.permission.RECORD_AUDIO",
        "timestamp" : "2023-10-10T17:51:29.377",
        "elementId" : 1392783768
      }, {...}
  ]
}
```

Listing 9.2: Permission file json

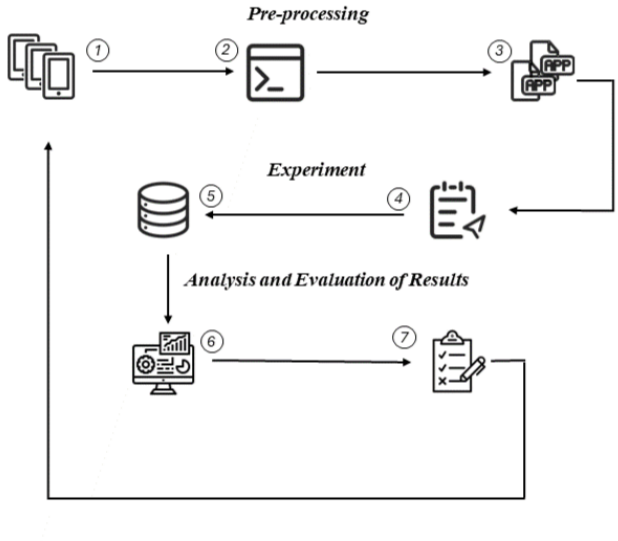


Figure 9.1: Iterative experimental process

The execution process (Figure 9.1) unfolds as follows. Screenshots ① are analyzed to determine the actions to be taken on them. Then, a prompt is created ② to be sent to GPT-4. After we select a subset of applications ③ to evaluate the previously mentioned metrics and thus assess the effectiveness of using the prompt and the sent screenshots ④. Then we send the image with the prompt at gpt-4 and the responses are consistently saved in a relational database to keep all the information compact ⑤. Following this, an analysis of the obtained results ⑥ is conducted to decide whether to make adjustments in aspects such as the prompt, image management, etc. or to continue the analysis with another subset of applications ⑦, progressing until reaching a total of 65 applications.

A **Pre-processing** phase was performed in order to optimize *screenshots* and *prompts*:

1. **Screenshots optimization**: The goal is to adapt the SCoPe output screenshots to the needs of the experiment, ensuring they are of adequate quality and accurately represent the user interfaces of the applications being examined. In the first two cycles of the experiment, this phase is carried

out to optimize the quality and relevance of the screenshots; in later cycles, this operation is not repeated, thanks to the establishment of an optimal structure already achieved.

2. **Prompt engineering**: Preparing the prompt is the most delicate phase of the experiment, requiring the most attention and time commitment. In this step, it's crucial to precisely formulate the questions to be asked to ChatGPT, clearly defining the desired information and the format in which it should be presented. The careful selection of prompts is decisive for the quality and usefulness of the responses obtained, directly influencing the effectiveness of the subsequent analysis.

Each of these points is essential for preparing and conducting the experiment, helping to ensure that the collected data are accurate, relevant, and easily analyzable.

During the **Experimental phase**, the analysis focus on groups of applications. Specific prompts, accompanied by the corresponding image (either in base64 code or a URL), are prepared and sent to GPT-4. Once a response from GPT-4 is received, the results are progressively recorded in a log file, which is later transferred into a relational database. This allows for structured data organization, facilitating analysis and potential retrieval for further evaluations or future experiments. Choosing a relational database ensures flexibility and scalability in data management, allowing for complex queries and maintaining information integrity.

The *log files* include the following essential information:

- **TimeStamp**: The date and time the request was made to GPT-4, serving as the transaction's timestamp.
- **ImageId**: A unique identifier for each analyzed screenshot, allowing easy correlation of the image with its analysis.
- **EventFile**: The name of the folder containing the screenshot in question, corresponds to the event name indicated in the JSON file. This link facilitates the traceability of the event associated with the screenshot.

- **EventType**: Specifies the type of event that generated the screenshot, with possible values being "Started" (indicating the screen was automatically initiated) or "Tap" (showing the screen was activated by a user action).
- **App**: The name of the application from which the screenshot originates, providing immediate context for the analysis.
- **Response**: The response provided by GPT-4, recorded in JSON format. This allows for the complete preservation of the analysis performed on the image, including any details and annotations provided by the artificial intelligence.

A critical aspect to consider is the possibility that GPT-4 may fail to process some images, either due to a lack of relevant information on the screen or problems accessing the image itself. In these cases, we conducted a manual investigation to verify the situation and, if necessary, repeated requests for unanalyzed or misinterpreted screens.

The size of application screenshots used in our analysis was reduced to improve efficiency. Originally, these screenshots had dimensions of 1140x3040 pixels, requiring significant memory and processing resources for 90 applications. We opted for a **proportional resizing** approach [139] to maintain the original aspect ratio and avoid image distortion.

The original image has an aspect ratio of approximately:

$$\text{Original Aspect Ratio} = \frac{1140}{3040} \approx 0.375$$

After resizing, the image has an aspect ratio of approximately:

$$\text{Resized Aspect Ratio} = \frac{280}{591} \approx 0.475$$

While the two ratios are not identical, the resizing attempted to stay as close as possible to the original aspect ratio, with some adjustments to fit the newly specified dimensions. Priority was given to maintaining a specific size (280x591) and adjusting the aspect ratio as close as possible to the original without exceeding the target dimensions [140].

Subsequently, it was decided to categorize the screenshots by events. Therefore, a new folder **ScreenShotResize** was created, which will have subfolders named after the event JSON files (*i.e.*, 2023-09-21T18_31_55.486).

This was accomplished using a Python script named IMAGE_UTILS. The script processes the JSON files for all events, and for each file, it analyzes the "elements". Each element contains an "elementId" value, most often associated with screenshots, as in Listing 9.3.

The script compares each "elementId" with the name of the screenshot (by removing the file extension) to check if an image with that element exists. If so, the image is resized. Listing 9.4.

To resize an image, the script first opens the image to ensure it is in the correct format and then saves it in the "ScreenShotResize" folder, which is associated with the folder of its respective event.

After several trial sessions and thorough analyses of screenshots, we have developed specific prompts for GPT-4 to optimize the analysis of application screens. These prompts were refined based on the responses received and are structured around four key elements essential for the analysis:

1. **Description**: To provide a clear context of the screenshot, we deemed it essential to include a description of what the screen represents. We decided to use a character range (300-350) for this description to ensure both conciseness and completeness.

2. **Permissions**: We focus on potentially dangerous permissions that could be utilized on the screen. Initially, we asked GPT-4 to identify these permissions. Later, we also requested a percentage estimate of their likelihood of use, accompanied by a brief explanation limited to a maximum of 6 words. This compromise was chosen after noting that 4 words were too concise, while 10 words tended to be overly extended.

3. **Danger**: We assess the level of danger to the user's privacy and security presented by the screen, quantifying this evaluation in percentage and providing a brief explanation. This parameter is crucial for determining the potential need to halt the application's operation or to issue a warning to the

```
def copy_from_original_source_and_resize(base_path, application, subfolder):
    # Json folder path of events
    json_folder_path = os.path.join(base_path, subfolder, application, 'events')

    # Folder path of image
    images_folder_path = base_path + "/" + subfolder + "/" + application + "/" + 'screenshots/'
    # Get all json file of events
    json_files = glob.glob(os.path.join(json_folder_path, '*.json'))
    copied_images_count = 0
    screenshot_resize_exist_path = os.path.join(base_path, subfolder, application, 'ScreenshotResize')
    if os.path.exists(screenshot_resize_exist_path):
        print("Scan already performed")
    else:
        #For each json file and each elements, find screenshot with same elementId
        for json_file_path in json_files:
            print(f"JsonFile: {json_file_path}")
            with open(json_file_path, 'r', encoding='utf-8') as json_file:
                data = json.load(json_file)

            if data:
                if data.get('elements', []):
                    # Get list of events included in field "elements"
                    elements = data.get('elements', [])

                    destination_folder_name = os.path.splitext(os.path.basename(json_file_path))[0]

                    for element in elements:
                        element_id = element.get('elementId')
```

Listing 9.3: Copy, resize and organize screenshot. Phase I

```
1
2                       image_file_name = f"{element_id}.png"
3                       #Usually screenshots folde have one subfolder,
    but sometimes that have more than one
4                       subfolders = os.listdir(images_folder_path)
5                       for folder in subfolders:
6                           image_path =
    os.path.join(images_folder_path, folder, image_file_name)
7
8                           if folder:
9                               subfolder_path =
    os.path.join(images_folder_path, folder)
10                              contents = os.listdir(subfolder_path)
11
12                              # iterate through all the photos in the
    subdirectory and check if the photos are present,
13                              # in the json{image_file_name}
14                              for element in contents:
15                                  # Se sono uguali faccio il resize
16                                  if element == image_file_name:
17                                      copied_images_count += 1
18                                      resize(base_path,image_path,
    element, destination_folder_name, subfolder, application)
19                                      print(f" {element} correctly
    copied")
20
21                          else:
22                              print(f"The {images_folder_path} path
    does not exist or is not a directory.")
23
24  if copied_images_count == 0:
25      print("No image found for elementId in the JSON.")
26  else:
27      print(f"Total copied images: {copied_images_count}")
```

Listing 9.4: Copy, resize and organize screenshot. Phase II

user, especially if the screen indicates the use of unauthorized permissions with a high degree of associated risk.

4. **Interaction**: We analyze the level of user interaction or engagement expected from the screen, differentiating between activities that require *low or high spatio-temporal attention*. This distinction helps us understand the importance of paying attention to certain application screens and evaluating whether they involve less demanding tasks or activities that require greater concentration.

Through this structure, we aim for a detailed understanding of application screens, emphasizing aspects such as contextual description, identification of dangerous permissions, evaluation of user risk, and the level of interaction required.

The prompt sent in this phase is:

First Prompt

Given this screenshot, knowing that: - I am using Android 11; - I have touched a button (I haven't touched any button). List me, in this way, these informations, don't use bold: 1. Description: Description of the screen, using a maximum of 300-350 characters.
2. Permission: Possible permissions used of the android permission model, you list me the permissions with technical names, a description in maximum 6 words and your security percentage with explanation of this percentage (max 6 words). Example: - `android.permission.RECORD_AUDIO`: permission for audio, 60% because it is important.
3. Danger: Danger level of the screen concerning privacy and security of user, with percentage and explanation of this percentage (max 6 words). Example: 10%, low, no risk.
4. Interaction: Check, as a percentage with a maximum of 6 words for explanation, whether the screen is high interaction or low, i.e., you have to be careful not to gawk at the screen all the time (like in action games) or you can look away several times. Example: - high (or low) interaction: 10%, just an intro screen.

As a preliminary assessment of this first prompt, 1000 screenshots from four different applications were selected, and an automated process was implemented using a Python script. In the context of automated screenshot analysis for applications, the process unfolds through various key steps, preserving the original structure of the description:

1. ***Event Selection***: For each element corresponding to an event as shown in Listing 9.5, identified by the name of the folder containing the screenshot, the `process_image` function is invoked. This critical step determines whether the screenshot is related to a specific user action that might have triggered certain permissions, or if the screen appeared automatically without any interaction.

2. ***process_image function***: This function as shown in Listing 9.6 plays a pivotal role in the analysis, tasked with examining the screenshot in question. It receives parameters such as the image (which can be provided via a link or in base64 format), the triggering event, the specific text to incorporate into the prompt intended for GPT-4 for targeted analysis, and finally, the name of the JSON file that acts as a bridge to access the screenshot folder. The goal is to analyze the image in light of the triggering event to identify any relevant permissions.

3. ***Sending the Request to GPT-4***: Using the `send_request_to_gpt` function, the prompt, enriched with the aforementioned information, is sent to GPT-4. This crucial step allows the model to process the provided information and generate a response based on the image data and the context of the triggering event. GPT-4's response, structured in JSON format, is subsequently stored in a log file, ensuring a secure record and facilitating future analyses.

```
for json_file in json_files:
    with open(json_file, 'r', encoding='utf-8') as file:
        data = json.load(file)
    json_file_name = os.path.splitext(os.path.basename(json_file))[0]
    screenshot_folder = screenshot_folder_base_path + '/' +
    json_file_name

    if os.path.exists(screenshot_folder):
        elements = data.get('elements', [])
        # Elenca tutti i file nella cartella delle screenshot
        screenshot_files = os.listdir(screenshot_folder)
        print(f"File in {screenshot_folder}:")
        for idx, element in enumerate(elements, start=1):
            element_id = element.get('elementId')
        # for idx, file in enumerate(screenshot_files, start=1):
            image.append(element)
            event_type='Not Available'
            if element.get('eventType'):
                event_type = element.get('eventType')
            process_image(idx, len(elements),
                          json_file_name, str(element_id)+'.png',
    event_type)

    else:
        print(f"The folder {screenshot_folder} not exist")
```

Listing 9.5: Send request to GPT-4, Part I

```
def process_image(count, len_image, json_file_name, image,
    event_type):
    print(f"Processed image:  {count}/{len_image}: {image}")

    event_to_write = ''
    if event_type == "TAP":
        event_to_write = "I have touched a button"
    elif event_type == "STARTED":
        event_to_write = "I haven't touched any button"
    else:
        event_to_write = ''

    print(event_to_write)
    resp = send_request_to_gpt(image, event_to_write, json_file_name)

    description, permission, danger, interaction = "", "", "", ""

    content = resp['choices'][0]['message']['content']

    parts = content.split('\n\n')
    for part in parts:
        if part.startswith("1. Description:"):
            description = part.replace("1. Description: ",
    "").replace('"', '').replace("'", "")
        elif part.startswith("2. Permission:"):
            permission = part.replace("2. Permission:\n",
    "").replace("\n", "; ").replace('"', '').replace("'", "")
        elif part.startswith("3. Danger:"):
            danger = part.replace("3. Danger:\n", "").replace('"',
    '').replace("'", "")
        elif part.startswith("4. Interaction:"):
            interaction = part.replace("4. Interaction:\n",
    "").replace('"', '').replace("'", "")

    with open('extracted_data.csv', 'a', newline='',
    encoding='utf-8') as csvfile:
        fieldnames = ['Description', 'Permission', 'Danger',
    'Interaction', 'ImageId', 'Event', 'App', 'EventType']
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

        writer.writerow(
            {'Description': description, 'Permission': permission,
    'Danger': danger, 'Interaction': interaction,
             'ImageId': image, 'Event': json_file_name, 'App':
    appName, 'EventType':event_type})

    countdown(80)
    print("\n")
```

Listing 9.6: Send request to GPT-4, Part II

Given the nature of the images, which can be processed by the gpt-4-vision-preview model in either base64 or URL format, we chose to upload the images to a dedicated server. This decision was driven by the need to reduce the token consumption for each request, as direct inclusion of images in base64 proved too resource-intensive.

Upon receiving the JSON response from GPT-4, we converted it into a record for the permissions table. This conversion highlighted that immediate transformation can introduce discrepancies due to slight variations in the response pattern from GPT-4, which is not always aligned with our anticipated format. Therefore, we decided to delay this conversion until the end of the overall analysis to ensure that the information entered into the permissions table is accurate and meets the required standards.

The data collected in the log file were transformed into records for a database table through another Python script. For each of the four applications, two JSON files were generated:

- **permissions_chatgpt.json**: Contains the image IDs and the permissions identified by GPT-4 as in Listing 9.7.
- **common_permissions.json:** Similarly, it includes the image IDs and lists the common permissions, those identified by both GPT-4 and SCOPE. This file, as in Listing 9.8, was created by comparing the permissions recorded in our database and those present in the SCOPE database for each application, thereby facilitating the identification of matches.

After that, we focused on identifying duplicated images within each application to reduce computational effort.

Three main methodologies were explored to detect identical images:

1. Image Hashing: This method converts images into hash values using perceptual hash functions (such as aHash, pHash, or dHash). Perceptual hashes aim to generate similar hash values for visually similar images, unlike cryptographic hashes which produce vastly different outputs for minor changes in input data. Generally, a Hamming distance metric is used to compare the hashes, where a smaller distance indicates greater similarity. A

```
[
    {
        "id": 37,
        "ImageId": "-2115990886.png",
        "permission": [
            {
                "name": "android.permission.READ_SMS",
                "percentage": 75
            },
            {
                "name": "android.permission.RECEIVE_SMS",
                "percentage": 75
            }
        ]
    }
    [...]
]
```

Listing 9.7: Permission for each image found by gpt-4

```
[
    {...},
    {
        "id": 5,
        "imageId": "2033022610.png",
        "Common_Permission": [
            "android.permission.READ_EXTERNAL_STORAGE",
            "android.permission.ACCESS_NETWORK_STATE"
        ],
        "CommonPermissionFound": "2 of 3",
        "Permission_rpcdroid": [
            "android.permission.WRITE_EXTERNAL_STORAGE",
            "android.permission.READ_EXTERNAL_STORAGE",
            "android.permission.ACCESS_NETWORK_STATE"
        ],
        "Permission_found": [
            "android.permission.INTERNET",
            "android.permission.ACCESS_NETWORK_STATE",
            "android.permission.READ_EXTERNAL_STORAGE"
        ],
        "additional_permission": [
            "android.permission.INTERNET"
        ]
    }, {...}
]
```

Listing 9.8: Common permission between SCoPE and GPT-4

common threshold is to consider images with a Hamming distance of less than 10 bits as similar, allowing for minor variations without losing the ability to detect substantial similarities.

2. Histogram Comparison: This technique involves comparing the image histograms while excluding white pixels. Although this method can be effective, it has limitations in scenarios where screens contain a significant amount of white pixels, leading to potential misclassification.

3. Structural Similarity Index (SSIM): SSIM [141] assesses image quality based on three components: luminance, contrast, and structure. Despite its greater accuracy compared to hash-based methods, using SSIM proved to be too time-consuming, especially when applied to a large number of images [142, 143, 144].

In light of these considerations, the comparison via perceptual hashing was chosen as the primary method for identifying duplicate images. To address the issue of excessive sensitivity to minor variations, we adopted a three-bit tolerance margin in the Hamming distance: if the difference between the hashes of two images is less than three bits, they are considered the same. This approach is distinct from cryptographic hashing, which is not suitable for detecting visual similarities due to its design principles that produce unrelated outputs even for minor input differences.

The automated process of detecting duplicate images, operates in the following manner:

- A reference image is selected and compared against all other images within the application.
- If a duplicate image is identified, only the path of the original image is retained.
- Once all images have been analyzed, those already considered are excluded, and the process continues with the rest of the set until analysis is complete. The outcome of this process is stored in a JSON file named `image_duplicate.json` for each application.

This file records for each group of duplicate images:

- **Original**: the path of the original reference image.

- **Duplicates**: the paths of the images identified as duplicates.

This system not only simplifies image management but also ensures that each screenshot is accurately evaluated, avoiding redundancies and optimizing the overall analysis. By reducing the number from 90,094 to 5,865 images, we achieved a significant reduction, approximately **93%**, in the total number of screenshots to be analyzed. This drastic decrease is crucial for optimizing the analysis process, substantially reducing the number of requests needed to GPT-4. In this context, if an image, for example, "123.png", has 10 duplicates, permissions, description, danger level, and interaction levels are assigned to each of the duplicates. This approach is adopted because there is a possibility that the main image might not be available in the SCOPE database. Therefore, to ensure comprehensive analytical coverage, it's necessary to extend the analysis to all duplicates of the concerned image. This methodology ensures that every relevant aspect related to permissions and user security is thoroughly evaluated, regardless of the immediate availability of the main image in the database. The prompt was modified to have an appropriate response to the fourth point (*interaction type*), specifying the difference between high and low spatio-temporal [145]:

Second Prompt

Given this screenshot, knowing that: I am using Android 11. I have touched android.widget.button (I haven't touched anything). List me, in this way, this information, don't bold anything:
1. Description: Description of the screen, using a maximum of 300-350 characters.
2. Permission: Possible permissions used of the Android permission model, you list the permissions with technical names, a description in a maximum of 6 words, and your security percentage with an explanation of this percentage (max 6 words). Example: `-android.permission.RECORD_AUDIO`: permission for audio, 60% because I notice a specific button.
3. Danger: Danger level of the screen concerning privacy and security of the user, with percentage and explanation of this percentage (max 6 words). Example: - 10%, low, no risk.
4. Interaction: Check, as a percentage with a maximum of 6 words for an explanation, whether the screen is high interaction or low (engagement levels of the user). This consisted of two main categories of tasks: low spatiotemporal attention tasks (quizzes) and high spatiotemporal attention tasks (games). Example (you can use only one of these): - high spatiotemporal: 90%, high, you pay attention to the screen
- low spatio-temporal: 80%, you have to do anything.

We then conducted another preliminary analysis, improving the efficacy of our solution in terms of time. This increased processing and analysis speed was made possible by optimizing processes and eliminating duplicates, reducing the number of images to analyze.

A detailed verification method was implemented to evaluate the accuracy and specificity of the analysis conducted with the support of GPT-4. This method involves associating the information provided by GPT-4 for a selected image with all its identified duplicates. Duplicates were taken based on the criterion that if identical screens make use of the same permission, then they are duplicates; otherwise, they are two different images. In practice, this means that the permissions, description, danger level, and interaction level detected for

an image are automatically extended to its duplicates. This procedure ensures that the analysis is consistent and comprehensive, as it considers all versions of an image as a single informative entity. Given the results from the analysis, a decision was made to revise the prompt, with a focused emphasis on ‘dangerous‘ permissions. The initial assumption was that only ‘dangerous‘ permissions would be relevant; however, for LLM, it is crucial to engineer the prompt meticulously. Considering our primary interest in Accuracy and Specificity, we have opted for a rephrasing of the prompt, focusing specifically on permissions considered dangerous rather than on permissions in general, such as those related to internet connection (ACCESS_NETWORK_STATE, INTERNET).

Third Prompt

Given this screenshot, knowing that: - I am using android 11; - I have touched android.widget.button (I haven't touched anything). List me, in this way this information, don't bold anything:
1. Description: Description of the screen, using a maximum of 300-350 characters; 2. Permission: Possible permissions, only dangerous (e.g. `READ_CALENDAR`, `WRITE_CALENDAR`, `CAMERA`, `READ_CONTACTS`, `WRITE_CONTACTS`, `GET_ACCOUNTS`, `ACCESS_FINE_LOCATION`, `ACCESS_COARSE_LOCATION`, `RECORD_AUDIO`, `READ_PHONE_STATE`, `READ_PHONE_NUMBERS`, `CALL_PHONE`, `ANSWER_PHONE_CALLS`, `BODY_SENSORS`, `SEND_SMS`, `RECEIVE_SMS`, `READ_SMS`, `READ_EXTERNAL_STORAGE`, `WRITE_EXTERNAL_STORAGE`), using the android permission model, you list the permissions with technical names, a description in a maximum of 6 words, and your security percentage with an explanation of this percentage (max 6 words). I don't want internet permissions, access_network_state, or similar. I don't want normal permissions Example: -android.permission.RECORD_AUDIO: permission for audio, 60% because it is important. 3. Danger: Danger level of the screen concerning privacy and security of the user, with percentage and explanation of this percentage (max 6 words). Example: - 10%, low, no risk. 4. Interaction: Check, as a percentage with a maximum of 6 words for an explanation, whether the screen is high interaction or low (engagement levels of the user). This consisted of two main categories of tasks: low spatio-temporal attention tasks (quizzes) and high spatio-temporal attention task (games). Example (you can use only one of this): - high spatio-temporal: 90%, high, you pay attention to the screen, - low spatio-temporal: 80%, you don't have to do anything

	TP	FP	TN	FN	Accuracy	Specificity	#
android.permission.ACCESS_FINE_LOCATION	1901	3054	10225	802	76%	77%	31
android.permission.CAMERA	370	462	12311	141	95%	96%	25
android.permission.WRITE_EXTERNAL_STORAGE	864	325	6614	117	94%	95%	15
android.permission.RECORD_AUDIO	139	46	4053	20	98%	98%	12
android.permission.READ_PHONE_STATE	31	2	5133	100	98%	99%	9
*android.permission.*READ_EXTERNAL_STORAGE	160	301	1953	227	80%	86%	7
android.permission.READ_CONTACTS	51	21	2505	3	99%	99%	7
android.permission.ACCESS_COARSE_LOCATION	20	7	493	2	98%	99%	3
android.permission.READ_SMS	7	5	336	0	98%	98%	2
android.permission.CALL_PHONE	3	52	48	1	49%	48%	1

Table 9.1: Summary of results

9.3 Analysis results

This section assesses ChatGPT's ability to identify Android permissions utilizing a composite approach that integrates screenshots with supplementary information extracted from the outputs of the SCoPe dynamic analysis tool.

Specifically, the new approach combines the output from app execution through SCoPe.

9.3.1 Analysis of Permission Usage in Android Applications

The Table 9.2 presents a comprehensive overview of permission usage in Android applications.

Overview of Results

Permission Usage Distribution:

The Table 9.2 lists various Android permissions along with corresponding *True Positives* (**TP**), *False Positives* (**FP**), *True Negatives* (**TN**), *False Negatives* (**FN**), *Accuracy*, *Specificity*, and the total *count* of applications (#) utilizing each permission.

Accuracy and Specificity

Accuracy and *specificity* metrics are essential for assessing the reliability and effectiveness of permission usage. Higher accuracy and specificity values indicate better performance in correctly identifying the presence or absence of permissions.

	TP	FP	TN	FN	Accuracy	Specificity	#
android.permission.ACCESS_FINE_LOCATION	1901	3054	10225	802	76%	77%	31
android.permission.CAMERA	370	462	12311	141	95%	96%	25
android.permission.WRITE_EXTERNAL_STORAGE	864	325	6614	117	94%	95%	15
android.permission.RECORD_AUDIO	139	46	4053	20	98%	98%	12
android.permission.READ_PHONE_STATE	31	2	5133	100	98%	99%	9
*android.permission.*READ_EXTERNAL_STORAGE	160	301	1953	227	80%	86%	7
android.permission.READ_CONTACTS	51	21	2505	3	99%	99%	7
android.permission.ACCESS_COARSE_LOCATION	20	7	493	2	98%	99%	3
android.permission.READ_SMS	7	5	336	0	98%	98%	2
android.permission.CALL_PHONE	3	52	48	1	49%	48%	1

Table 9.2: Evaluation of the final results

Table 9.2 presents a comprehensive overview of permission usage in Android applications on 65 analyzed applications. Permissions are crucial for ensuring the security and privacy of user data on mobile devices, however, their usage needs to be scrutinized to prevent potential misuse or abuse by applications. Let's delve into the analysis of the presented results.

It's important to note this data regarding location, based on the last table 9.2:

According to the official Android documentation, *ACCESS_FINE_LOCATION* must be requested along with ACCESS_*COARSE_LOCATION*. So, it's rather strange that FINE_LOCATION is requested more frequently than COARSE_LOCATION in all applications.

This is both an interesting and concerning finding because the number of permissions for ACCESS_COARSE_LOCATION (3) should ideally be at least as high as the number for FINE_LOCATION (31), given that having an approximate location is a prerequisite for obtaining a precise one.

The positive aspect of our approach is that the false negative count (802) for exact localization is rather high. This indicates that our tool wouldn't have requested it for 802 instances where applications utilize this localization. Upon analyzing the specific applications that make use of FINE_LOCATION in detail, however, every time the permission wasn't identified, it was still an application utilizing location services.

The only applications that correctly utilize location services (i.e., first COARSE and then FINE) are: *com.mendhak.gpslogger* and *info.zamojski.soft.towercollector*, which are respectively two applications using maps for geolocation. Instead,

com.shazam.android only uses COARSE, which according to official documentation, is useful for determining where the song recognition is being requested from. For instance, if someone in Italy asks to recognize a song, it can approximately identify their location to offer Shazam charts of trending songs in that nation.

Based on a manual analysis of a sample of screenshots, all other applications that use precise localization seem to do so correctly. However, they should adhere to the Android paradigm of using approximate location first and then precise location.

The study by Huang *et al.* [146], demonstrates that there is a security issue in Android related to the lack of granular permission control for external storage. Currently, any application can easily access all data stored in external storage. However, this could pose serious security problems as many applications store sensitive data in this area.

Indeed, upon analyzing the data on storage, 22 applications make use of it, which is a rather high number. This warrants evaluating the results of these applications to verify their proper usage.

As final results, experiments were conducted to assess the effectiveness of using an LLM for identifying permissions given a specific screen. In the first experiment, as depicted in Table 9.3, it is evident that among the analysis of 15 applications, permissions related to camera and location services are the most frequently requested. While some permissions exhibited good accuracy and specificity, others displayed notably lower values. For instance, permissions related to storage showed an accuracy of 0% and a specificity of 100%, while permissions related to contacts showed an accuracy of 33% and a specificity of 66%.

Considering these results, the prompt was revised with a focused emphasis on "dangerous" permissions. The initial assumption was that only "dangerous" permissions would be relevant; however, for LLM, it is crucial to design the prompt to elicit appropriate responses meticulously.

Analyzing the two tables, 9.5 and 9.4, we can observe significant improvements in terms of accuracy and specificity. However, there is an exception with ACCESS_FINE_LOCATION, which is difficult for an LLM to identify. This difficulty arises because, without clear visual cues like a map or indications of location usage, GPT-4 struggled to pinpoint the permission accurately.

	TP	FP	TN	FN	Accuracy	Specificity	#
android.permission.ACCESS_FINE_LOCATION	32	94	2813	156	91%	97%	6
android.permission.CAMERA	59	161	4181	126	94%	96%	8
android.permission.WRITE_EXTERNAL_STORAGE	0	0	179	6	0%	100%	1
android.permission.READ_PHONE_STATE	0	0	4161	93	97%	100%	4
android.permission.READ_CONTACTS	1	2	1	1	33%	66%	1
android.permission.RECORD_AUDIO	3	0	791	3	99%	100%	1

Table 9.3: Permission Metric Analysis: II Improving Analysis result

	TP	FP	TN	FN	Accuracy	Specificity	#
android.permission.CAMERA	123	54	6003	14	98%	99%	9
android.permission.ACCESS_FINE_LOCATION	176	1326	178	44	20%	11%	7
*android.permission.*READ_EXTERNAL_STORAGE	7	3	613	1	99%	99%	2
android.permission.WRITE_EXTERNAL_STORAGE	7	0	650	0	100%	100%	3
android.permission.READ_CONTACTS	3	1	504	1	99%	99%	1
android.permission.RECORD_AUDIO	21	3	1390	1	99%	99%	5
android.permission.READ_PHONE_STATE	4	0	590	0	100%	100%	1

Table 9.4: Evaluation Permission for commercial application: III Main Experiment

Significantly high data, on the other hand, is observed for CAMERA in both open-source and commercial applications. Particularly for commercial ones, the usage of the camera correlates well with high accuracy and specificity, indicating the effectiveness of utilizing an LLM to analyze camera-utilizing applications. If used to counter illegitimate camera usage, it could predict improper camera usage accurately 99% of the time.

However, for the WRITE_EXTERNAL_STORAGE permission (across both application categories), while the values are higher than before, there are notable instances of false positives. This prompts us to consider that employing an AI-based tool might lack accuracy. It is accurate in identifying the permission 69% of the time, a somewhat low figure for a permission that involves writing on the device.

A remarkably high value is observed for RECORD_AUDIO, which, concerning its usage frequency (#5), boasts an accuracy and specificity of 99% in both cases. Once again, this underscores the potential of using AI to assess actual microphone usage effectively.

Given this information, we have concluded that we can proceed with the chosen prompt. Thus, we have finished analyzing the remaining applications, as

	TP	FP	TN	FN	Accuracy	Specificity	#
android.permission.ACCESS_FINE_LOCATION	812	635	2081	220	77%	76%	8
android.permission.CAMERA	45	43	679	0	94%	94%	2
*android.permission.*READ_EXTERNAL_STORAGE	125	281	910	177	69%	76%	2
android.permission.WRITE_EXTERNAL_STORAGE	766	1550	5295	106	78%	77%	6
android.permission.READ_CONTACTS	32	15	1750	0	99%	99%	2
android.permission.ACCESS_COARSE_LOCATION	15	6	132	2	94%	95%	2
android.permission.READ_PHONE_STATE	11	0	113	5	96%	100%	1

Table 9.5: Evaluation Permission for Open-Source application: III Main Experiment

	TP	FP	TN	FN	Accuracy	Specificity	#
android.permission.ACCESS_FINE_LOCATION	1611	2792	1710	382	51%	38%	10
android.permission.CAMERA	143	100	1848	1	95%	95%	6
*android.permission.*READ_EXTERNAL_STORAGE	28	17	430	49	87%	96%	3
android.permission.WRITE_EXTERNAL_STORAGE	91	175	490	5	76%	74%	5
android.permission.READ_CONTACTS	15	3	250	1	98%	98%	3
android.permission.ACCESS_COARSE_LOCATION	5	1	361	0	99%	99%	1
android.permission.READ_PHONE_STATE	16	2	296	2	98%	99%	3
android.permission.RECORD_AUDIO	115	43	1872	19	96%	97%	7
android.permission.READ_SMS	7	5	336	0	98%	98%	2
android.permission.CALL_PHONE	3	52	48	1	49%	48%	1

Table 9.6: Summary of analyzed application: Final Experiment

illustrated in the table 9.6. These data reinforce the previous assertions regarding both storage, which has remained more or less consistent, and the same reasoning applies to location services.

9.4 Assessing Effectiveness on Malicious Apps

To assess the validity of our study, we selected 8 applications suspected of abusing permissions. We started by analyzing these apps with SCoPE to extract permissions and usage contexts. Next, we conducted a static analysis by decompiling the APKs and examining all suspicious activities for illicit use of permissions.

We then refined the experimental protocol to fit the analysis of such applications in real-world contexts. The goal was no longer just to understand GPT-4's ability to identify permissions, but to test whether, given a screen and the permissions

used, the model could discern whether or not the use of such permissions was appropriate or not.

Pre-processing phase

At this stage, we redefined the prompt and modified the approach to select relevant screenshots. For each activity that triggered permissions, we documented the following information:

- The activity triggering the permission.
- The activity preceding the current one.
- The next activity that would be triggered if it interacted with a button.
- The permissions detected by SCoPE and GPT-4's response of their appropriate use.

An example of the result is shown in 9.4 and 9.4 :

Example of detecting malicious use of permission

- **ImageId**: 993555763.png
- **Activity**: AgreementActivity
- **ParentActivity**: InstallActivitys
- **ElementTap**: android.widget.Button
- **Content**: Agree
- **ChildActivity**: ConsoleActivitys
- **Permissions**: - android.permission.SEND_SMS
- **Prompt**: Given this screenshot, knowing that: - android.widget.Button was selected which has the following text: Agree - dangerous permission was requested: android.permission.SEND_SMS Do you think it is appropriate to request permission at this time, yes (Motivate me your answer in a maximum of 300 characters) or no (Motivate me your answer in a maximum of 300 characters)?
- **Gpt Response**: No, it's not appropriate. The app requests SMS permission, which could charge you via premium services without clear consent for each transaction. It's a common tactic in scam apps.

Example of normal use of permissions

- **ImageId**: -262974777.png
- **Activity**: AgreementActivity
- **ParentActivity**: InstallActivitys
- **ElementTap**: android.widget.Button
- **Content**: Disagree
- **ChildActivity**: InstallActivitys
- **Permissions**: None
- **Prompt**: Given this screenshot, knowing that: - android.widget.Button was selected which has the following text: Disagree - no dangerous permission has been requested Do you think any permission dangerous should be required at this time, yes (List me the possible permissions or the possible permission like this: - android.permission.NAME) or no (Motivate me your answer in maximum 300 characters)?
- **Gpt Response**: No, if no dangerous permissions have been requested, it is not necessary to require them just because the user disagrees with the terms of service. Disagreeing should not change the app's access to device features or data; it typically means the user opts not to use the service..

ScreenShot

We chose to consider only screenshots related to events that activated permissions, to concretely evaluate the accuracy of the model in detecting permissions in authentic contexts. For each app, therefore, we selected a minimum of one to a maximum of eight screenshots.

Prompt

The prompt was modulated according to whether a button had been pressed causing the activation of a permit, or whether no action had been taken and, despite this, the use of one or more permits had been detected.

Button selected and permission detected

Given this screenshot, knowing that: - android.widget.Button was selected which has the following text: Install - dangerous permission was requested: `android.permission.SEND_SMS` Do you think it is appropriate to request permission at this time, yes (Motivate me your answer in a maximum of 300 characters) or no (Motivate me your answer in a maximum of 300 characters)?

Started activity and permission detected

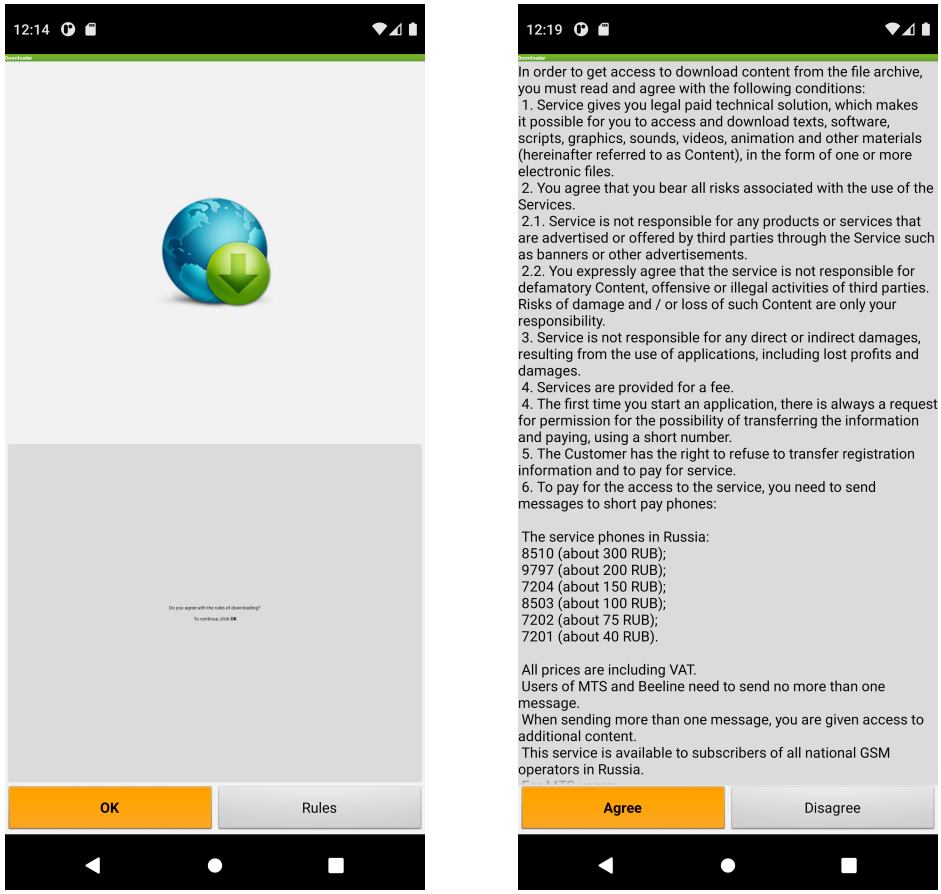
Given this screenshot, knowing that: - I did not select any buttons - dangerous permission was requested: - `android.permission.DEVICE_POWER`, - `android.permission.GET_TASKS`, - `android.permission.READ_EXTERNAL_STORAGE`, `android.permission.WRITE_EXTERNAL_STORAGE` Do you think it is appropriate to request permission at this time, yes (Motivate me with your answer in a maximum 300 characters) or no (Motivate me your answer in a maximum 300 characters)?

Example of response for malicious permission

An example of a malicious application is com.opera.A.install; this application serves to download the actual Opera application. Before performing the download, selecting one or more buttons, allows the sending of numerous promotional messages using the phone number of the user who downloaded the application. After sending the messages, it downloads the official application, without raising the user's suspicion about the message sending.

As seen in figure 9.2a, it is a simple screen to accept or refuse the download of the real application. In this case, if you click on "OK," then the messages are sent, and the application is subsequently downloaded, while if you click on "Rules," you go to the terms and conditions screen. Also in this second case as seen in figure 9.2b, if you click on "Agree," then the messages are sent and the application is downloaded; otherwise, if you click on "Disagree," the application closes.

To verify this issue, the prompt 9.4 was sent when the "Agree" button was clicked, and the download was initiated.



(a) (b)

Figure 9.2: Accept rules for download(a), read rules and accept for download (b)

An example of a ChatGPT response is as follows:

Response example

No, it's not appropriate. The app requests SMS permission, which could charge you via premium services without clear consent for each transaction. It's a common tactic in scam apps.

This type of response could be used to warn the user, perhaps through a pop-up, about the dangers of misusing permissions.

However, if for GPT-4 the use of that permission is considered correct, it would not block the user with a warning pop-up.

9.4.1 Experiment

For each app, we formulated and sent GPT-4 the most suitable prompt based on the permissions and events encountered. This work was done manually by our team, evaluating the relevance of the model responses against the appropriateness of the permissions involved.

Analysis and Evaluation of Results

Through manual code analysis, we confirmed the misuse of permissions and evaluated the reliability of GPT-4's conclusions. Surprisingly, the model recognized inappropriate use of permissions in all 8 applications examined. To further verify the validity of this approach, we also tested screenshots of apps that made correct use of permissions, and even under these circumstances, GPT-4 was able to correctly assess their appropriateness.

This experiment demonstrated the remarkable effectiveness of GPT-4 in detecting and evaluating the use of permissions, accurately flagging when permissions are improperly used and thus offering a potential tool for automating the permissions checking process in Android applications.

In the table 9.7, we see the 8 applications analyzed, the malicious permissions used, and the actual blocking of these permissions by GPT-4.

Malicius Application	**Use of improper permission**	**Blocked by gpt-4**
com.opera.A.install	2	2
com.opera.B.installer	9	9
com.excelliance.dualaid	4	4
com.ksa.passbook	1	1
com.ps.yams	1	1
com.sileria.alsalah	2	2
com.sohu.inputmethod.sogou	6	6
com.ydbl.kudou	5	5

Table 9.7: Malicious application analyzed

9.5 Final remarks

This Chapter has presented a comprehensive study on applying Large Language Models (LLMs) for dynamic permission management in Android applications. Our investigation centered on the innovative use of GPT-4 for analyzing app interfaces and logs to discern the appropriateness of permission requests, thereby aiming to mitigate unauthorized data access and enhance user privacy.

The experimental results, derived from a series of iterative procedures using SCoPe, have demonstrated that LLMs can effectively interpret the context of permission use, distinguish between benign and potentially malicious permissions, and provide recommendations with considerable accuracy. GPT-4, in particular, showcased a robust capability in generating context-aware assessments that align closely with the security requirements of modern mobile ecosystems.

Moreover, our approach has highlighted the feasibility of using AI to automate the evaluation of permission requests, which could significantly reduce user dependency on making security-critical decisions, thus potentially lowering the risk of 'user fatigue' and errors in permission granting. The success of this method in the experimental setup suggests a promising avenue for integrating LLMs into real-world mobile security frameworks, offering a more adaptive and user-centric approach to permission management.

CHAPTER 10

Conclusion

In today's mobile-centric world, Android has emerged as the dominant operating system, powering billions of devices worldwide. However, the vast popularity of Android has also made it a prime target for malicious actors, highlighting the critical need for robust security measures. The pervasiveness of Android devices has transformed the way we interact with technology, providing seamless access to information, communication, and entertainment. However, this convenience comes at a cost, as the vast amount of personal data stored on these devices makes them attractive targets for cybercriminals. Android's permission model, designed to control app access to sensitive resources, has been criticized for its lack of granularity and transparency, often leaving users vulnerable to privacy intrusions and security breaches. To address these shortcomings, this thesis employed a multi-pronged research approach, encompassing dynamic analysis, user studies, and the development of novel permission management systems. The primary goal was to investigate and implement techniques that empower users with greater control over their data, enhance transparency, and mitigate security risks.

As **key findings and contributions**, The research conducted in this thesis yielded significant findings and contributions that have the potential to revolutionize Android app security and privacy:

- **Dynamic Analysis and Misuse Detection (Chapter 3):** To gain a deeper understanding of how commercial Android apps utilize permissions, a dynamic analysis tool named RPCDroid was developed. RPCDroid monitors app behavior at runtime, capturing detailed logs of permission requests and UI events. This analysis revealed that many apps exhibit opaque and excessive permission usage, often accessing sensitive data without explicit user consent. These findings underscore the need for improved transparency and user control over app permissions.

- **Assessing the Effectiveness of Privacy Indicators (Chapters 4 & 5):** Chapters 4 and 5 focused on evaluating Google's Privacy Indicators (PI), which were introduced in Android 12 to enhance user security. Despite their potential, our studies involving 122 participants revealed that the PIs did not significantly impact user awareness in detecting resource use, especially in latent contexts. This suggests a need for more informative designs that can actively engage users. Consequently, a redesigned PI featuring textual information was tested, showing improved user recognition and awareness. An innovative follow-up study employed eye-tracking technology to capture real-time user engagement, further demonstrating that specific visual features like color and motion could enhance PI detectability. However, tasks demanding high cognitive load, such as gaming, often led to inattentional blindness, indicating that even enhanced PIs can be overlooked under certain conditions.

- **SCoPe: A Context-Based, User-Centered Permission Management System (Chapter 6):** To address the shortcomings of traditional permission models, SCoPe, a context-based permission management system, was developed. SCoPe leverages dynamic analysis to identify the context in which permission requests are made, enabling users to grant or deny permissions based on specific usage scenarios. SCoPe's effectiveness was highlighted by its ability to achieve an 84% accuracy rate in contextualizing

permission usage across various apps. This tool represents a significant step forward in enabling users to control their data more effectively and securely.

- **SCoPe's effectiveness against Logic Bombs (Chapter 8):** Logic bombs are malicious code snippets that activate under specific conditions, such as a certain date or user action. They can execute harmful operations like data deletion or unauthorized data access. Logic bombs are challenging to detect because they remain dormant until triggered, often escaping standard security scans. By contextualizing permission requests, SCoPe can identify and mitigate suspicious activities potentially linked to logic bombs. If an app requests access to sensitive resources under unusual circumstances, SCoPe can alert the user and restrict permission, effectively preventing the malicious action. This targeted approach helps safeguard devices against hidden and emergent threats, enhancing overall app security and user trust.

- **Leveraging AI for Informed Decision-Making (Chapter 9):** Recognizing the potential of Artificial Intelligence (AI), particularly Large Language Models (LLMs), to enhance security decision-making, this thesis explored their integration into permission management systems. LLMs can analyze app UIs and provide context-aware recommendations to users, empowering them to make informed choices about granting permissions.

10.1 Lessons Learned

Dynamic Permission Models are Essential: Our research validated that static permission systems are inadequate for the nuanced demands of current mobile app usage. Dynamic models like SCoPe, which adjust permissions based on context and real-time interactions, provide a more robust framework for privacy management.

Context Matters: The effectiveness of security tools greatly depends on the context in which they are applied. Tools like SCoPe that offer contextual analysis provide more relevant and secure management of permissions.

User Engagement is Key: For security measures to be effective, they must engage the user. Our studies showed that even well-designed PIs could fail if they

do not sufficiently capture the user's attention or are too complex to understand at a glance. Adaptive Security Solutions are Essential: As threats evolve, so must our approaches. Integrating AI into security solutions offers a dynamic and adaptive framework that can respond to new challenges as they arise.

User Awareness is Crucial: The limited effectiveness of existing Privacy Indicators highlighted the need for designs that are more visible and informative. Enhanced user awareness leads to better privacy practices and reduces the risk of unintentional sensitive data access.

AI Integration Can Enhance Privacy Controls: Employing artificial intelligence, particularly LLMs like GPT-4, in evaluating and predicting suitable permissions can significantly improve the accuracy and efficiency of permission models. This integration also helps in making the systems more user-friendly and less intrusive.

The Impact of Cognitive Load on Privacy Notifications: Our studies using eye-tracking technology indicated that the cognitive load under which a user operates can impact the effectiveness of privacy indicators. This insight is critical for designing future user interfaces that need to communicate privacy information effectively under various user conditions.

Transparency Correlates with Privacy: The observation that open-source apps typically exhibit lower rates of sensitive resource access suggests that transparency in app development may encourage better privacy practices. This relationship between transparency and privacy should be further explored and leveraged in future app development policies.

Balancing Security and Convenience: SCoPe's approach to minimize interruptions by requesting permissions only in novel contexts is a testament to the delicate balance between maintaining security and enhancing user experience. This strategy also helps in combating "warning fatigue," where users become desensitized to alerts, potentially overlooking critical privacy risks.

These lessons point toward a future where mobile app privacy is dynamically managed, contextually aware, and deeply integrated with user-centric designs, paving the way for safer and more user-friendly digital environments.

Appendices

APPENDIX A

Publications

C1 **M. Guerra**, R. Milanese, R. Oliveto, and F. Fasano, "RPCDroid: Runtime Identification of Permission Usage Contexts in Android Applications," in Proceedings of the 9th International Conference on Information Systems Security and Privacy, ICISSP 2023, Lisbon, Portugal, February 22-24, 2023, pp. 714–721, 2023.

Candidate CRediT authorship contribution statement: Conceptualization (equal), Methodology (lead), Validation (equal), Investigation (lead), Writing - Original Draft (lead), Visualization (lead)

J1 **M. Guerra**, S. Scalabrino, F. Fasano, and R. Oliveto, "An empirical study on the effectiveness of privacy indicators," IEEE Transactions on Software Engineering, vol. 49, pp. 4610–4623, oct 2023. (**Accepted as Journal-First at ICSE 24 IEEE/ACM International Conference on Software Engineering**)

Candidate CRediT authorship contribution statement: Methodology (equal), Software (lead), Investigation (lead), Data Curation (lead), Writing - Original Draft (lead), Writing - Review Editing (equal), Visualization (lead), Formal analysis (equal)

C2 **M. Guerra**, R. Milanese, M. Deodato, V. Perozzi, and F. Fasano, "Visual Attention and Privacy Indicators in Android: Insights from Eye Tracking," in Proceedings of the 10th International Conference on Information Systems Security and Privacy, ICISSP 2024, Rome, Italy, 2024.

Candidate CRediT authorship contribution statement: Conceptualization (lead), Methodology (lead), Validation (equal), Writing - Original Draft (lead), Writing - Review Editing (equal), Visualization (lead), Supervision (equal)

C3 Fasano, F., **Guerra, M.**, Milanese, R., Oliveto, R. (2023). "A Dynamic Approach to Defuse Logic Bombs in Android Applications," In Data and Applications Security and Privacy XXXVII. DBSec 2023. Lecture Notes in Computer Science, vol 13942. Springer, 2023

Candidate CRediT authorship contribution statement: Conceptualization (equal), Methodology (lead), Validation (equal), Investigation (lead), Writing - Original Draft (equal), Visualization (lead)

A.1 Other Publications

C4 S. Khandker, **M. Guerra**, E. Bitsikas, R. P. Jover, A. Ranganathan, and C. Pöpper. "ASTRA-5G: Automated Over-the-Air Security Testing and Research Architecture for 5G SA Devices" In Proceedings of 17th ACM Conference on Security and Privacy in Wireless and Mobile Networks (ACM WiSec), 2024

A.2 Software Artifacts

S1 **RPCDroid Framework** to perform a runtime identification of permission usage in Android Applications with visual insights.

Framework: `https://github.com/rmilanese99/rpcdroid-analysis` *(Released to the scientific community)*

Example output: `https://github.com/MicheleGuerra/RPCdroid-output`

S2 The Figma prototypes of the apps used in the **Privacy Indicators experimentation** on Android are available and fully customizable. These tools allow for easy personalization and adaptation for further research or practical applications.

More details: `https://dibt.unimol.it/report/privacy-indicators/` *(Released to the scientific community)*

S3 **ASTRA-5G:** Automated Over-the-Air Security Testing and Research Architecture for 5G SA Devices

Framework: `https://github.com/s21sm/ASTRA-5G` *(Released to the scientific community)*

Bibliography

[1] "Forecast of the number of smartphone users in the world from 2010 to 2025." https://www.statista.com/forecasts/1143723/.

[2] "Number of apps available in leading app stores as of 3rd quarter 2022." https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[3] "Mobile operating systems' market share worldwide from january 2012 to june 2021." https://www.statista.com/statistics/272698/.

[4] N. Eling, S. Rasthofer, M. Kolhagen, E. Bodden, and P. Buxmann, "Investigating users' reaction to fine-grained data requests: A market experiment," in *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pp. 3666–3675, IEEE, 2016.

[5] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang, "Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing," in *Proceedings of the 2012 ACM conference on ubiquitous computing*, pp. 501–510, 2012.

[6] G. L. Scoccia, M. Autili, and P. Inverardi, "A self-configuring and adaptive privacy-aware permission system for android apps," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pp. 38–47, IEEE, 2020.

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

[8] G. L. Scoccia, S. Ruberto, I. Malavolta, M. Autili, and P. Inverardi, "An investigation into Android run-time permissions from the end users' perspective," in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 45–55, 2018.

[9] I. M. Almomani and A. Al Khayer, "A comprehensive analysis of the android permissions system," *Ieee access*, vol. 8, pp. 216671–216688, 2020.

[10] V. K. Malviya, C. W. Leow, A. Kasthuri, Y. N. Tun, L. K. Shar, and L. Jiang, "Right to know, right to refuse: Towards ui perception-based automated fine-grained permission controls for android apps," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–6, 2022.

[11] G. L. Scoccia, I. Malavolta, M. Autili, A. Di Salle, and P. Inverardi, "Enhancing trustability of android applications via user-centric flexible permissions," *IEEE Transactions on Software Engineering*, 2019.

[12] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov, "Dynamically regulating mobile application permissions," *IEEE Security & Privacy*, vol. 16, no. 1, pp. 64–71, 2018.

[13] Z. Fang, W. Han, and Y. Li, "Permission based android security: Issues and countermeasures," *computers & security*, vol. 43, pp. 205–218, 2014.

[14] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the eighth symposium on usable privacy and security*, pp. 1–14, 2012.

[15] V. K. Malviya, Y. N. Tun, C. W. Leow, A. T. Xynyn, L. K. Shar, and L. Jiang, "Fine-grained in-context permission classification for android apps using control-flow graph embedding." 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023).

[16] Y. Wang, Y. Wang, S. Wang, Y. Liu, C. Xu, S.-C. Cheung, H. Yu, and Z. Zhu, "Runtime permission issues in android apps: Taxonomy, practices, and ways forward," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 185–210, 2022.

[17] S. Chitkara, N. Gothoskar, S. Harish, J. I. Hong, and Y. Agarwal, "Does this app really need my location? context-aware privacy management for smartphones,"

Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, vol. 1, no. 3, pp. 1–22, 2017.

[18] M. Guerra, S. Scalabrino, F. Fasano, and R. Oliveto, "An empirical study on the effectiveness of privacy indicators," *IEEE Transactions on Software Engineering*, 2023.

[19] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," *ACM Trans. Softw. Eng. Methodol.*, 2021.

[20] D. Akhawe and A. P. Felt, "Alice in warningland: a {Large-Scale} field study of browser security warning effectiveness," in *22nd USENIX security symposium (USENIX Security 13)*, pp. 257–272, 2013.

[21] StatCounter, "Market share of mobile operating systems worldwide from 2009 to 2023, by quarter," 2024.

[22] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, "The android platform security model," *ACM Transactions on Privacy and Security*, vol. 24, no. 3, 2021.

[23] "Architecture overview." `https://source.android.com/docs/core/architecture`.

[24] "Platform architecture." `https://developer.android.com/guide/platform`.

[25] "Android runtime (art) and dalvik." `https://source.android.com/docs/core/runtime`.

[26] "<permission>." `https://developer.android.com/guide/topics/manifest/permission-element`.

[27] "Permissions on android." `https://developer.android.com/guide/topics/permissions/overview`.

[28] "Request runtime permissions." `https://developer.android.com/training/permissions/requesting`.

[29] V. K. Malviya, Y. N. Tun, C. W. Leow, A. T. Xynyn, L. K. Shar, and L. Jiang, "Fine-grained in-context permission classification for android apps using control-flow graph embedding," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1225–1237, 2023.

[30] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, (New York, NY, USA), p. 627–638, Association for Computing Machinery, 2011.

[31] J. Tang, R. Li, H. Han, H. Zhang, and X. Gu, "Detecting permission over-claim of android applications with static and semantic analysis approach," in *2017 IEEE Trustcom/BigDataSE/ICESS*, pp. 706–713, 2017.

[32] P. Calciati, K. Kuznetsov, A. Gorla, and A. Zeller, "Automatically granted permissions in android apps: An empirical study on their prevalence and on the potential threats for privacy," in *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, (New York, NY, USA), p. 114–124, Association for Computing Machinery, 2020.

[33] M. Diamantaris, E. P. Papadopoulos, E. P. Markatos, S. Ioannidis, and J. Polakis, "Reaper: Real-time app analysis for augmenting the android permission system," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, CODASPY '19, (New York, NY, USA), p. 37–48, Association for Computing Machinery, 2019.

[34] H. Wang, Y. Li, Y. Guo, Y. Agarwal, and J. I. Hong, "Understanding the purpose of permission use in mobile apps," *ACM Transactions on Information Systems*, vol. 35, no. 4, 2017.

[35] M. Wei, X. Gong, and W. Wang, "Claim what you need: A text-mining approach on android permission request authorization," in *2015 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, 2015.

[36] R. Wang, Z. Wang, B. Tang, L. Zhao, and L. Wang, "Smartpi: Understanding permission implications of android apps from user reviews," *IEEE Transactions on Mobile Computing*, vol. 19, no. 12, p. 2933 – 2945, 2020.

[37] M. Z. Hossen and M. Mannan, "On understanding permission usage contextuality in android apps," *Lecture Notes in Computer Science*, vol. 10980 LNCS, p. 232 – 242, 2018.

[38] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," 2012.

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

[40] L. Reinfelder, A. Schankin, S. Russ, and Z. Benenson, "An inquiry into perception and usage of smartphone permission models," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11033 LNCS, p. 9 – 22, 2018.

[41] G. L. Scoccia, S. Ruberto, I. Malavolta, M. Autili, and P. Inverardi, "An investigation into android run-time permissions from the end users' perspective," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, (New York, NY, USA), p. 45–55, Association for Computing Machinery, 2018.

[42] W. Cao, C. Xia, S. T. Peddinti, D. Lie, N. Taft, and L. M. Austin, "A large scale study of user behavior, expectations and engagement with android permissions," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 803–820, USENIX Association, Aug. 2021.

[43] R. Baalous and R. Poet, "Factors affecting users' disclosure decisions in android runtime permissions model," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 1113–1118, 2020.

[44] B. Shen, L. Wei, C. Xiang, Y. Wu, M. Shen, Y. Zhou, and X. Jin, "Can systems explain permissions better? understanding users' misperceptions under smartphone runtime permission model," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 751–768, USENIX Association, Aug. 2021.

[45] J. Palmerino, "Improving android permissions models for increased user awareness and security," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, (New York, NY, USA), p. 41–42, Association for Computing Machinery, 2018.

[46] M. Guerra, S. Scalabrino, F. Fasano, and R. Oliveto, "An empirical study on the effectiveness of privacy indicators," *IEEE Transactions on Software Engineering*, vol. 49, pp. 4610–4623, oct 2023.

[47] B. Liu, J. Lin, and N. Sadeh, "Reconciling mobile app privacy and usability on smartphones: could user privacy profiles help?," in *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, (New York, NY, USA), p. 201–212, Association for Computing Machinery, 2014.

[48] A. Oglaza, R. Laborde, P. Zaraté, A. Benzekri, and F. Barrère, "A new approach for managing android permissions: learning users' preferences," *Eurasip Journal on Information Security*, vol. 2017, no. 1, 2017.

[49] B. Rashidi, C. Fung, A. Nguyen, T. Vu, and E. Bertino, "Android user privacy preserving through crowdsourcing," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 3, p. 773 – 787, 2018.

[50] H. Gao, C. Guo, Y. Wu, N. Dong, X. Hou, S. Xu, and J. Xu, "Autoper: Automatic recommender for runtime-permission in android applications," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 107–116, 2019.

[51] C. Fung, B. Rashidi, and V. G. Motti, "Multi-view permission risk notification for smartphone system," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, vol. 10, no. 1, p. 42 – 57, 2019.

[52] M. Nauman, S. Khan, A. T. Othman, and S. Musa, "Realization of a user-centric, privacy preserving permission framework for android," *Security and Communication Networks*, vol. 8, no. 3, p. 368 – 382, 2015.

[53] "Context-centric security," in *7th USENIX Workshop on Hot Topics in Security (HotSec 12)*, (Bellevue, WA), USENIX Association, Aug. 2012.

[54] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. android and mr. hide: fine-grained permissions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, (New York, NY, USA), p. 3–14, Association for Computing Machinery, 2012.

[55] D. Akhawe and A. P. Felt, "Alice in warningland: A Large-Scale field study of browser security warning effectiveness," in *22nd USENIX Security Symposium (USENIX Security 13)*, (Washington, D.C.), pp. 257–272, USENIX Association, Aug. 2013.

[56] J. Fu, Y. Zhou, H. Liu, Y. Kang, and X. Wang, "Perman: Fine-grained permission management for android applications," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 250–259, 2017.

[57] G. L. Scoccia, I. Malavolta, M. Autili, A. Di Salle, and P. Inverardi, "Enhancing trustability of android applications via user-centric flexible permissions," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2032–2051, 2021.

[58] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), p. 1036–1046, Association for Computing Machinery, 2014.

[59] Y. Li, Y. Guo, and X. Chen, "Peruim: understanding mobile application privacy with permission-ui mapping," in *Proceedings of the 2016 ACM International Joint*

Conference on Pervasive and Ubiquitous Computing, UbiComp '16, (New York, NY, USA), p. 682–693, Association for Computing Machinery, 2016.

[60] M. Duan, L. Jiang, L. K. Shar, and D. Gao, "Uipdroid: Unrooted dynamic monitor of android app uis for fine-grained permission control," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 227–231, 2022.

[61] X. Du, X. Pan, Y. Cao, B. He, G. Fan, Y. Chen, and D. Xu, "Flowcog: Context-aware semantic extraction and analysis of information flow leaks in android apps," *IEEE Transactions on Mobile Computing*, vol. 22, no. 11, pp. 6460–6476, 2023.

[62] H. Fereidooni, M. Conti, D. Yao, and A. Sperduti, "Anastasia: Android malware detection using static analysis of applications," in *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, 2016.

[63] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 377–396, 2016.

[64] H. Kang, J.-w. Jang, D. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," *International Journal of Distributed Sensor Networks*, vol. 2015, pp. 1–9, 06 2015.

[65] D. Papp, L. Buttyán, and Z. Ma, "Towards semi-automated detection of trigger-based behavior for software security assurance," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[66] Q. Zhao, C. Zuo, B. Dolan-Gavitt, G. Pellegrino, and Z. Lin, "Automatic uncovering of hidden behaviors from input validation in mobile apps," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1106–1120, 2020.

[67] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware," *Proceedings of the 7th European Workshop on System Security, EuroSec 2014*, 04 2014.

[68] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. X. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*, 2008.

[69] M. Choudhary and B. Kishore, "Haamd: Hybrid analysis for android malware detection," in *2018 International Conference on Computer Communication and Informatics (ICCCI)*, pp. 1–4, 2018.

[70] H. Drewes, A. De Luca, and A. Schmidt, "Eye-gaze interaction for mobile phones," pp. 364–371, 09 2007.

[71] N. Peiravian and X. Zhu, "Machine learning for android malware detection using permission and api calls," in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pp. 300–305, 2013.

[72] J. Sahs and L. Khan, "A machine learning approach to android malware detection," in *2012 European Intelligence and Security Informatics Conference*, pp. 141–147, 2012.

[73] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding android obfuscation techniques: A large-scale investigation in the wild," in *Security and Privacy in Communication Networks* (R. Beyah, B. Chang, Y. Li, and S. Zhu, eds.), (Cham), pp. 172–192, Springer International Publishing, 2018.

[74] E. Erturk, "A case study in open source software security and privacy: Android adware," in *World Congress on Internet Security (WorldCIS-2012)*, pp. 189–191, 2012.

[75] H. Pieterse and M. S. Olivier, "Android botnets on the rise: Trends and characteristics," in *2012 Information Security for South Africa*, pp. 1–5, 2012.

[76] T. Yang, Y. Yang, K. Qian, D. C.-T. Lo, Y. Qian, and L. Tao, "Automated detection and analysis for android ransomware," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pp. 1338–1343, 2015.

[77] M. H. Saad, A. Serageldin, and G. I. Salama, "Android spyware disease and medication," in *2015 Second International Conference on Information Security and Cyber Forensics (InfoSec)*, pp. 118–125, 2015.

[78] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE Symposium on Security and Privacy*, pp. 95–109, 2012.

[79] O. Gadyatskaya, A.-L. Lezza, and Y. Zhauniarovich, "Evaluation of resource-based app repackaging detection in android," pp. 135–151, 11 2016.

[80] L. Li, T. F. Bissyande, and J. Klein, "Rebooting research on detecting repackaged android apps: Literature review and benchmark," *IEEE Transactions on Software Engineering*, vol. 47, pp. 676–693, apr 2021.

[81] L. Li, T. F. Bissyandé, and J. Klein, "Simidroid: Identifying and explaining similarities in android apps," in *2017 IEEE Trustcom/BigDataSE/ICESS*, pp. 136–143, 2017.

[82] W. Zhou, X. Zhang, and X. Jiang, "Appink: Watermarking android apps for repackaging deterrence," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2013.

[83] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps," 01 2017.

[84] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, (New York, NY, USA), p. 93–104, Association for Computing Machinery, 2012.

[85] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, pp. 358–366, 1953.

[86] H. Agrawal, J. L. Alberi, L. Bahler, J. Micallef, A. Virodov, M. Magenheimer, S. Snyder, V. Debroy, and E. Wong, "Detecting hidden logic bombs in critical infrastructure software," 2012.

[87] Q. Zeng, L. Luo, Z. Qian, X. Du, Z. Li, C.-T. Huang, and C. Farkas, "Resilient user-side android application repackaging and tampering detection using cryptographically obfuscated logic bombs," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 6, pp. 2582–2600, 2021.

[88] J. Samhi and A. Bartel, "On the (in)effectiveness of static logic bomb detection for android apps," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, pp. 3822–3836, 2022.

[89] J. Samhi, L. Li, T. F. Bissyand'e, and J. Klein, "Difuzer: Uncovering suspicious hidden sensitive operations in android apps," *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 723–735, 2021.

[90] S. Furman and M. Theofanos, "Preserving privacy – more than reading a message," in *Universal Access in Human-Computer Interaction. Design for All and Accessibility Practice* (C. Stephanidis and M. Antona, eds.), (Cham), pp. 14–25, Springer International Publishing, 2014.

[91] S. Egelman, "My profile is my password, verify me! the privacy/convenience tradeoff of facebook connect," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, (New York, NY, USA), p. 2369–2378, Association for Computing Machinery, 2013.

[92] P. A. Punde, M. E. Jadhav, and R. R. Manza, "A study of eye tracking technology and its applications," in *2017 1st International Conference on Intelligent Systems and Information Management (ICISIM)*, pp. 86–90, IEEE, 2017.

[93] B. T. Carter and S. G. Luke, "Best practices in eye tracking research," *International Journal of Psychophysiology*, vol. 155, pp. 49–62, 2020.

[94] M. Arianezhad, L. J. Camp, T. Kelley, and D. Stebila, "Comparative eye tracking of experts and novices in web single sign-on," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, (New York, NY, USA), p. 105–116, Association for Computing Machinery, 2013.

[95] T. Whalen and K. Inkpen, "Gathering evidence: use of visual security cues in web browsers," in *Proceedings of Graphics Interface 2005*, GI 2005, (School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada), pp. 137–144, Canadian Human-Computer Communications Society, 2005.

[96] D. Miyamoto, T. Iimura, G. Blanc, H. Tazaki, and Y. Kadobayashi, "Eyebit: Eye-tracking approach for enforcing phishing prevention habits," in *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pp. 56–65, 2014.

[97] M. Guerra, S. Scalabrino, F. Fasano, and R. Oliveto, "An empirical study on the effectiveness of privacy indicators," *IEEE Transactions on Software Engineering*, vol. 49, pp. 4610–4623, oct 2023.

[98] B. B. Anderson, J. L. Jenkins, A. Vance, C. B. Kirwan, and D. Eargle, "Your memory is working against you: How eye tracking and memory explain habituation to security warnings," *Decision Support Systems*, vol. 92, pp. 3–13, 2016.

[99] B. Shen, L. Wei, C. Xiang, Y. Wu, M. Shen, Y. Zhou, and X. Jin, "Can systems explain permissions better? understanding users' misperceptions under smartphone runtime permission model.," in *USENIX Security Symposium*, pp. 751–768, 2021.

[100] Y. Elbitar, M. Schilling, T. T. Nguyen, M. Backes, and S. Bugiel, "Explanation beats context: The effect of timing & rationales on users' runtime permission decisions," *USENIX Security'21*, 2021.

[101] G. L. Scoccia, I. Malavolta, M. Autili, A. Di Salle, and P. Inverardi, "Enhancing trustability of android applications via user-centric flexible permissions," *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2032–2051, 2021.

[102] T. Osman, M. Mannan, U. Hengartner, and A. Youssef, "Securing applications against side-channel attacks through resource access veto," *Digital Threats: Research and Practice*, vol. 1, no. 4, pp. 1–29, 2020.

[103] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee, "From Zygote to Morula: Fortifying Weakened ASLR on Android," in *2014 IEEE Symposium on Security and Privacy*, pp. 424–439, 2014.

[104] Google, "Android developers guide: Overview of memory management," 2022. `https://developer.android.com/topic/performance/memory-overview`.

[105] X. Liu, J. Liu, S. Zhu, W. Wang, and X. Zhang, "Privacy risk analysis and mitigation of analytics libraries in the Android ecosystem," *IEEE Transactions on Mobile Computing*, vol. 19, no. 5, pp. 1184–1199, 2019.

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

[107] Google, "Android studio guide: Ui/application exerciser monkey," 2022. `https://developer.android.com/studio/test/monkey`.

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

[109] M. Guerra, R. Milanese, R. Oliveto, and F. Fasano, "RPCDroid: Runtime Identification of Permission Usage Contexts in Android Applications," in *Proceedings of the 9th International Conference on Information Systems Security and Privacy, ICISSP 2023, Lisbon, Portugal, February 22-24, 2023* (P. Mori, G. Lenzini, and S. Furnell, eds.), pp. 714–721, SciTePress, 2023.

[110] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov, "Dynamically regulating mobile application permissions," *IEEE Security Privacy*, vol. 16, no. 1, pp. 64–71, 2018.

[111] M. A. Cohen, P. Cavanagh, M. M. Chun, and K. Nakayama, "The attentional requirements of consciousness," *Trends in Cognitive Sciences*, vol. 16, no. 8, pp. 411–417, 2012.

[112] A. CHALMERS and K. CATER, "41 - exploiting human visual perception in visualization," in *Visualization Handbook* (C. D. Hansen and C. R. Johnson, eds.), pp. 807–816, Burlington: Butterworth-Heinemann, 2005.

[113] A. Mack, "Inattentional blindness: Looking without seeing," *Current Directions in Psychological Science*, vol. 12, no. 5, pp. 180–184, 2003.

[114] R. A. Fisher, "Statistical methods for research workers," in *Breakthroughs in statistics*, pp. 66–70, Springer, 1992.

[115] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[116] H. Chen, P. Cohen, and S. Chen, "How big is a big odds ratio? interpreting the magnitudes of odds ratios in epidemiological studies," *Communications in Statistics—simulation and Computation®*, vol. 39, no. 4, pp. 860–864, 2010.

[117] H. Aguinis and K. J. Bradley, "Best practice recommendations for designing and implementing experimental vignette methodology studies," *Organizational Research Methods*, vol. 17, no. 4, pp. 351–371, 2014.

[118] D. J. Woehr and C. E. Lance, "Paper people versus direct observation: An empirical examination of laboratory methodologies," *Journal of Organizational Behavior*, vol. 12, pp. 387–397, 1991.

[119] R. Hughes and M. Huby, "The application of vignettes in social and nursing research," *Journal of advanced nursing*, vol. 37, pp. 382–6, 03 2002.

[120] A. Papoutsaki, P. Sangkloy, J. Laskey, N. Daskalova, J. Huang, and J. Hays, "Webgazer: Scalable webcam eye tracking using user interactions," in *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 3839–3845, AAAI, 2016.

[121] A. Papoutsaki, A. Gokaslan, J. Tompkin, Y. He, and J. Huang, "The eye of the typer: A benchmark and analysis of gaze behavior during typing," in *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications*, ETRA '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[122] "Xposed framework api reference." `https://api.xposed.info/`.

[123] K. Lim, Y. Jeong, S.-j. Cho, M. Park, and S. Han, "An android application protection scheme against dynamic reverse engineering attacks.," *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, vol. 7, no. 3, pp. 40–52, 2016.

[124] "Service." https://developer.android.com/reference/android/app/Service.

[125] "Services overview." https://developer.android.com/guide/components/services.

[126] M. Guerra, R. Milanese, R. Oliveto, and F. Fasano, "Rpcdroid: Runtime identification of permission usage contexts in android applications.," in *ICISSP*, pp. 714–721, 2023.

[127] A. Romdhana, M. Ceccato, G. C. Georgiu, A. Merlo, and P. Tonella, "Cosmo: Code coverage made easier for android," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pp. 417–423, 2021.

[128] P. Kotzias, J. Caballero, and L. Bilge, "How did that get in my phone? unwanted app distribution on android devices," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 53–69, 2021.

[129] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, (New York, NY, USA), pp. 468–471, ACM, 2016.

[130] S. Khalid and F. B. Hussain, "Evaluating dynamic analysis features for android malware categorization," in *2022 International Wireless Communications and Mobile Computing (IWCMC)*, pp. 401–406, 2022.

[131] J. Bellizzi and M. Vella, "Wexpose: Towards on-line dynamic analysis of web attack payloads using just-in-time binary modification," *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, vol. 04, pp. 5–15, 2015.

[132] R. C. Sekar, "An efficient black-box technique for defeating web application attacks," in *Network and Distributed System Security Symposium*, 2009.

[133] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *Network and Distributed System Security Symposium*, 2007.

[134] W. N. Sumner and X. Zhang, "Identifying execution points for dynamic analyses," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 81–91, 2013.

[135] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang, "Precise calling context encoding," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1160–1177, 2012.

[136] B. Xin, W. N. Sumner, and X. Zhang, "Efficient program execution indexing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, (New York, NY, USA), p. 238–248, Association for Computing Machinery, 2008.

[137] M. D. Bond, G. Z. Baker, and S. Z. Guyer, "Breadcrumbs: efficient context sensitivity for dynamic bug detection analyses," in *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 2010.

[138] J. Samhi, T. F. Bissyandé, and J. Klein, "Triggerzoo: A dataset of android applications automatically infected with logic bombs," in *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, (New York, NY, USA), p. 459–463, Association for Computing Machinery, 2022.

[139] PBS Learning Media, "Proportional scaling." `https://www.pbslearningmedia.org/resource/rttt12.math.propscaling/proportional-scaling/`, 2023.

[140] ABC Fotografia, "Ricampionamento e interpolazione." `https://www.abc-fotografia.it/ricampionamento-e-interpolazione/`, 2023.

[141] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[142] J. Wang, S. Kumar, and S.-F. Chang, "Semi-supervised hashing for large-scale search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 12, pp. 2393–2406, 2012.

[143] V. Bruni and D. Vitulano, "An entropy based approach for ssim speed up," *Signal Processing*, vol. 135, pp. 198–209, 2017.

[144] A. Osokin and D. Sidorov, "Modification of ssim metrics," in *Information Technologies and Mathematical Modelling* (A. Dudin, A. Nazarov, R. Yakupov, and A. Gortsev, eds.), (Cham), pp. 351–355, Springer International Publishing, 2014.

[145] M. Guerra, R. Milanese, M. Deodato, V. Perozzi, and F. Fasano, "Visual attention and privacy indicators in android: Insights from eye tracking," in *Proceedings of the 10th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP*, pp. 320–329, INSTICC, SciTePress, 2024.

[146] F. Huang, W. Wu, M. Yang, and J. Luo, "A fine-grained permission control mechanism for external storage of android," in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 002911–002916, 2016.