An Empirical Study of Data Access Practice in Android AR Apps to Understand User Privacy Risks

Sabbir Hussain Meraj
University of Texas at San Antonio
San Antonio, USA
sabbirhussain.meraj@utsa.edu

Long Trac
University of Texas at San Antonio
San Antonio, USA
long.trac@utsa.edu

Xiaoyin Wang
University of Texas at San Antonio
San Antonio, USA
xiaoyin.wang@utsa.edu

Xusheng Xiao
Arizona State University
Tempe, USA
xusheng.xiao@asu.edu

Wei Wang
University of Texas at San Antonio
San Antonio, USA
wei.wang@utsa.edu

Abstract

Mobile Augmented Reality (AR) applications have rapidly expanded across domains such as gaming, education, and healthcare. However, AR's reliance on continuous camera input and access to sensitive data, such as faces and the surrounding environment, raises major privacy concerns.

Adhering to best privacy practices requires applications to clearly disclose their use of sensitive data and follow the Principle of Least Privilege (PoLP). This paper presents an empirical study of 179 Android AR applications to assess their data access behaviors and evaluate whether these practices are upheld in the real world.

To this end, we developed a novel static analysis technique to examine data accesses of these apps and identify potential privacy risks. Our analysis centers on core AR components, namely anchors and trackables, which link virtual content to real-world entities. Our findings revealed that AR apps often access sensitive data without adequate disclosure and frequently access more data than their functionality requires through unrestricted SDK APIs.

Notably, our analysis identified 58 instances of undisclosed access to facial data, 2D images, and skeletal information, and 82 instances of PoLP violations through overly broad data access. A subset of these findings was further confirmed through dynamic instrumentation.

Furthermore, our analysis suggests that protecting user privacy in AR apps can benefit from fine-grained access control, AR SDK API redesign, automated disclosure verification, and control/data-flow analysis. Particularly, there are 317 common call paths among the 754 AR-related paths we analyzed, suggesting that many AR apps share similar data usage patterns that can be leveraged for efficient privacy auditing and anomaly detection with manageable overhead.

CCS Concepts

• **Security and privacy → Software and application security**; *Privacy protections*; • **Software and its engineering**;

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834402

Keywords

Augmented Reality, Privacy, Android, Static Analysis, ARCore, Unity, AR Foundation

ACM Reference Format:
Sabbir Hussain Meraj, Long Trac, Xiaoyin Wang, Xusheng Xiao, and Wei Wang. 2026. An Empirical Study of Data Access Practice in Android AR Apps to Understand User Privacy Risks. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3834402

1 Introduction

Augmented Reality (AR) offers an immersive experience by integrating virtual elements, such as visual, auditory, haptic, and somatosensory stimuli, into the real-world environment, enhancing the user's perception and cognition of their physical surroundings. The introduction of AR Software Development Kits (SDKs), such as Google's ARCore [1], Apple's ARKit [18], and Unity's ARFoundation [32], greatly simplify the development of AR applications. Consequently, mobile AR apps have witnessed substantial growth in recent years across diverse sectors, such as education, gaming, manufacturing, retail, and healthcare [10, 11, 20, 26, 28, 39, 41].

AR apps typically require continuous access to live data streams from camera feeds and sensors in order to create an interactive and immersive experience. The collection of large volumes of sensor data, particularly from camera feeds, presents unique privacy concerns for AR apps, which may compromise data confidentiality and user privacy [9]. For instance, Figure 1 illustrates the output of an AR app, which enables users to place virtual objects on surfaces and capture stylized images with AR effects. However, as highlighted by the red oval marks in the figure, the app also accesses sensitive information, such as the coordinates and spatial extent of human faces (which are blurred for privacy protection). Such unauthorized access may jeopardize data confidentiality and infringe upon user privacy.

The privacy issue of AR apps have been long recognized by the research community. There were studies that examined the risks of unintentionally capturing the surrounding environment and bystanders through AR apps [14, 30]. Previous research has also explored AR's privacy and security issues from the end user's perspective and proposed a "code of ethics" for developers [9]. Frameworks have been designed to prevent the leakage of sensitive information

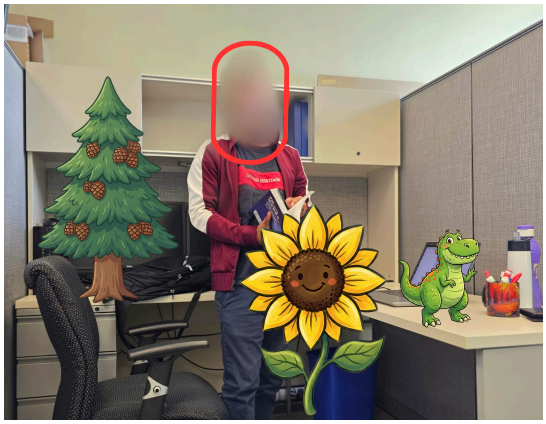


Figure 1: An example of AR App leaking private information.

from the user's surroundings [29, 33, 34]. Additionally, several studies have suggested fine-grained access control and permission systems to improve the security and privacy of AR systems [8, 19, 40]. Most of these studies and solutions essentially advocate two fundamental privacy principles. The first is *disclosure of data usage*, which requires that the app clearly discloses what data they collect and how they are used. The second is the *Principle of Least Privilege (PoLP)*, a well-known principle in information security and privacy, which states that any entity should access only the minimum data required to fulfill its task. The violation of PoLP, sometimes also referred to as Excessive Data Exposure (EDE), has been identified by the open web application security project (OWASP) as a top security risk in its API Security Top 10 reports [3, 4]. In the context of AR apps, disclosure of data usage requires that apps disclose their access to sensitive data such as camera frames, facial features, and environmental information, while PoLP requires that apps access only the specific data types necessary for their intended functionality.

However, after multiple years of advocation and development, it is still not clear to what extent these principles have been enforced in the AR SDKs and the code of real-world applications, leaving users in doubt about their privacy risks when using AR apps.

To bridge this gap, we conducted an empirical static analysis study of 179 Android AR apps, examining how they access sensitive data, particularly those obtained from camera frames, to identify potential *violations of privacy*. More specifically, our analysis focuses on two types of violations: (1) *undisclosed data access* — apps accessing sensitive data without explicitly mentioning it in the app description, privacy policy, or Google Play Data Safety section, and (2) *PoLP violations* — apps accessing more data than their functionality requires through overly broad access patterns and generic SDK APIs.

We focus on Android for two reasons: (1) its widespread adoption and easy hardware access make Android AR users one of the largest real-world AR user groups, and (2) Google's AR SDK is partially open source, and mature decompilation tools for Android apps enable thorough static analysis. Moreover, we emphasize static analysis because it offers insights that can guide and optimize more costly dynamic analysis in the future. Accordingly, this study addresses the following research questions (RQs):

(1) **RQ1:** To what extent do AR apps access sensitive data without disclosure?
(2) **RQ2:** To what extent do AR apps violate the Principle of Least Privilege (PoLP) by accessing more data than their functionality requires?
(3) **RQ3:** What are the potential strategies to protect user privacy in existing AR apps?

A key challenge for our analysis is the absence of static analysis techniques tailored to AR applications. While numerous approaches exist for analyzing Android mobile apps [13, 22, 24, 31, 35–37], they do not account for AR-specific elements or their interactions. As a result, these methods are inadequate for accurately identifying data accesses in Android AR apps.

To overcome this limitation, we developed a static analysis technique for Android AR apps built with two widely used SDKs: Google ARCore [1] and Unity ARFoundation [32]. Our approach focuses on AR-specific components, namely *anchors* and *trackables* [1]. In AR apps, *anchors* mark virtual object locations and are attached to real-world entities (such as surfaces, images, or faces) known as *trackables*. Since anchors and trackables form the foundation of AR functionality and mediate access to physical-world data, analyzing them enables more fine-grained inspection and precise detection of privacy violations. To this end, we extend existing tools such as Androguard and Ghidra with AR-specific entry points that general-purpose Android analyzers do not model, providing a unified analysis pipeline across both ARCore and ARFoundation. Finally, to validate the accuracy of our static analysis, we instrumented AR apps flagged for both undisclosed and broad data access to confirm whether these accesses could occur during execution.

Our analysis reveals that AR apps frequently access real-world objects (trackables) such as planes, human faces, images (e.g., photos and QR codes), and raw camera frames through API calls or event callbacks. Many of the accesses are not clearly disclosed in the app description, privacy policy, or Google Play Data Safety section, posing significant privacy risks. Our analysis identified 58 cases (9 in ARCore apps and 49 in ARFoundation apps) where facial, image, or body data were explicitly accessed without any disclosure.

Furthermore, AR apps can access more data than their functionality requires, violating PoLP. Such accesses occur in two main ways: (1) apps iterate over all available or updated trackables through generic APIs, exposing all detected objects regardless of type; and (2) apps access the full camera frame image through broad APIs, granting access to everything visible in the camera view. Regarding PoLP violations, our analysis identified 82 cases (54 in ARCore and 28 in ARFoundation).

These findings highlight the urgent need for stronger privacy enforcement in the AR ecosystem, both at the application and SDK levels. AR SDKs need to enforce finer-grained access controls on AR data to better uphold privacy. However, given the numerous types of sensitive AR data, overly fine-grained access control can be burdensome for ordinary users.

A better solution, as shown by our results, is to use static and/or dynamic code analysis. For example, automated disclosure verification can ensure that app disclosures accurately reflect actual data access practices. Additionally, control-flow and data-flow analyses focused on trackables/anchors can enforce privacy through code

audit and refactoring. Our findings further reveal common call paths involving AR-specific classes (236 common paths out of 620 paths in ARCore, and 81 out of 134 in ARFoundation), indicating that many AR apps share similar data usage patterns. This overlap may reduce the complexity and overhead of privacy monitoring and enables more efficient heuristics or machine learning models to detect anomalous or privacy-invasive data usage.

The contributions of this paper include:

(1) A comprehensive static analysis focused on AR-specific elements that was conducted on 179 Android AR apps, encompassing both Google ARCore and Unity ARFoundation apps, and examining both Java and native code.
(2) The types and frequency of sensitive physical-world AR data accessed by existing AR apps, across both ARCore and ARFoundation apps.
(3) The undisclosed data usage and PoLP privacy violations identified in current AR apps, illustrated with representative code snippets.
(4) The analysis of potential approaches for enhancing privacy in AR apps, leveraging fine-grained access control, control/data-flow monitoring, and machine learning.

The rest of this paper is organized as follows. Section 2 first presents background on AR SDKs. Section 3 then describes the analysis methodology used in this study. Subsequently, Section 4 presents our findings on undisclosed data accesses, followed by Section 5, which presents findings on PoLP violations. Section 6 examines how both types of violation are distributed across app categories. Section 7 then discusses potential privacy protection strategies based on our findings. Finally, Section 8 discusses limitations and future work, Section 9 reviews related work, and Section 10 concludes the paper.

2 Background

2.1 AR SDKs

Augmented Reality (AR) SDKs have simplified AR application development by providing essential tools and abstractions needed. Among the most widely adopted SDKs for Android AR development are Google ARCore [1] and Unity ARFoundation [32]. ARCore, used primarily with Java, is Google's platform for creating Android AR experiences. In contrast, Unity ARFoundation, a cross-platform framework built on Unity's XR plugin architecture, allows developers to create applications in C#. ARFoundation apps are compiled into native Android binaries using the IL2CPP backend and rely internally on ARCore when deployed on Android devices.

2.2 AR-Specific Data Types

Both ARCore and ARFoundation provide AR-specific elements that convey information about physical objects and the surrounding environment to AR apps. Two such fundamental elements are: (1) *trackables*, which are real-world features and objects, including planes, surfaces, images, human faces, and various environmental features (e.g., light probes, depth maps, ambient conditions), that AR SDKs detect and monitor continuously, and (2) *anchors*, which associate virtual objects with real-world objects (i.e., trackables).

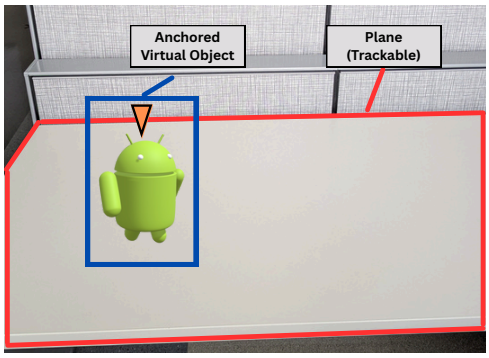


Figure 2: AR app example showcasing trackables and anchors.

While trackables provide an understanding of the physical environment, anchors ensure that virtual objects remain stably and consistently positioned within the augmented scene. For example, as illustrated in Figure 2, the desktop surface is recognized by ARCore as a "plane"-typed trackable, and an anchor is created for this plane, as indicated by the yellow arrow. A virtual Android mascot is placed at the anchor, which appears to be standing on the desktop.

Both SDKs provide access to sensitive real-world data through their trackable classes, as summarized in Table 1. In this study, we focus on the classes that pose the highest privacy risks. Specifically, `AugmentedFace` in ARCore and `ARFace` in ARFoundation track facial features and 3D face meshes, while `AugmentedImage` and `ARImage` detect 2D images such as photos and QR codes in the user's environment. ARFoundation additionally supports full-body tracking via `ARHumanBody`. Furthermore, the `Frame` class in ARCore and the `ARCameraManager` class in ARFoundation provide access to full camera frames, exposing all visual information in the user's surroundings. Finally, ARCore provides generic APIs such as `Session.getAllTrackables()`, `Frame.getUpdatedTrackables()`, and `HitResult.getTrackable()`, that can retrieve all detected trackables of any type, whereas ARFoundation does not provide such generic APIs. These classes and APIs form the basis of our privacy analysis in Sections 4 and 5.

2.3 Data Access Mechanism

Understanding how AR apps access sensitive data is crucial for evaluating their privacy implications. ARCore and ARFoundation have distinct data access mechanisms, as illustrated in Figure 3.

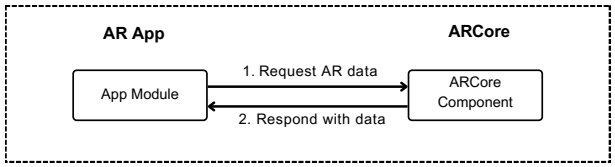
ARCore uses a pull-based, request-driven model, where apps explicitly request AR data through API calls, allowing data to be retrieved on demand but also increasing the risk of data leakage or misuse when access controls are inadequate.

On the other hand, ARFoundation apps predominantly use an event-driven model, where data updates are delivered through push-based event callbacks. Apps register event handlers to subscribe to lifecycle events such as `planesChanged`, `trackedImagesChanged`, and `facesChanged`, and the system automatically dispatches the relevant data when changes occur. Table 2 lists the supported ARFoundation events. This approach can result in excessive sensitive data being sent to apps regardless of their actual needs, complicating data tracking and security. While ARFoundation primarily

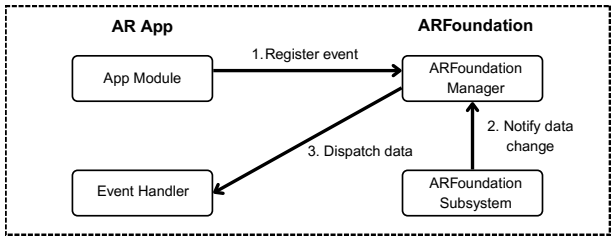
Table 1: AR-specific classes/data in ARCore and ARFoundation, and their usages.

ARCore	ARFoundation	Description
Plane	ARPlane (ARPlane, ARPlaneManager)	Detects flat surfaces
AugmentedFace*	**ARFace (ARFace, ARFaceManager)***	Tracks facial features and 3D face mesh
AugmentedImage*	**ARImage (ARTrackedImage, ARTrackedImageManager)***	Detects and tracks 2D images
Camera	N/A	Holds camera pose and intrinsics
Frame*	N/A	Snapshot of the current AR frame
Earth	N/A	Accesses user's Earth-based location and pose
GeospatialPose	N/A	Earth-referenced position and orientation
InstantPlacementPoint	N/A	Places objects without plane detection
DepthPoint	N/A	Point with depth data
StreetscapeGeometry	N/A	3D mesh of streets and buildings
Point	N/A	Single 3D point in space
N/A	**ARCamera (ARCameraManager)***	Handles camera frame and images
N/A	ARObject (ARTrackedObject, ARTrackedObjectManager)	Tracks 3D physical objects
N/A	**ARHumanBody (ARHumanBody, ARHumanBodyManager)***	Tracks human skeletons
N/A	AREnvironment (AREnvironmentProbe, AREnvironmentProbeManager)	Captures lighting and reflections information
N/A	ARRaycast (ARRaycast, ARRaycastManager)	Raycasts for hit testing in AR
N/A	ARParticipant (ARParticipant, ARParticipantManager)	Manages users in shared AR
N/A	ARPointCloud (ARPointCloud, ARPointCloudManager)	Collection of tracked feature points

* Privacy-sensitive classes analyzed in this study.



(a) Pull-based data access mechanism of ARCore Apps



(b) Push-based data access mechanism of ARFoundation Apps

Figure 3: Data access mechanism in ARCore and ARFoundation.

employs an event-driven mechanism, it also provides non-callback APIs (e.g., `TryGetBoundingBox()`), although such functions are limited in numbers and are used less frequently in practice.

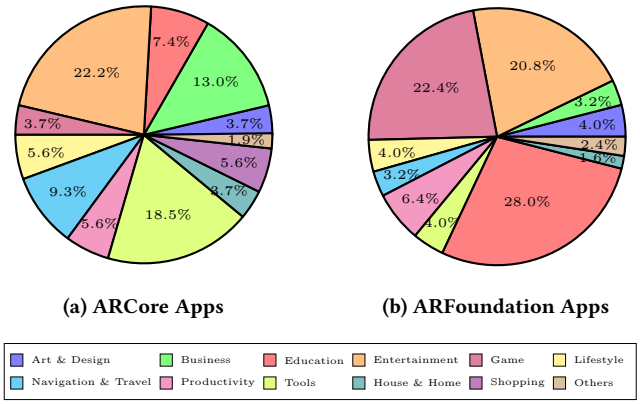
These differing mechanisms directly impact privacy protection: ARCore requires monitoring of API usage, whereas ARFoundation requires inspection of registered event handlers.

3 Analysis Methodology

3.1 AR Apps for Analysis

To conduct our analysis, we curated a dataset of Android AR apps from the Google Play Store by searching for the keywords "AR" and "augmented reality," as there is no specific filter for AR apps. Since APK files cannot be downloaded directly from the Google Play Store, we obtained the corresponding APKs from APKPure [7]. Our analysis focused on apps built with ARCore and ARFoundation. To verify the SDK, we automatically detected the presence of `com.google.ar.core.*` classes in the DEX files extracted by Androguard [2] for ARCore apps. For ARFoundation apps, we confirmed the presence of `Unity.XR.ARFoundation.dll` in the `dump.cs` file generated by IL2CppDumper [25]. Apps that did not satisfy either criterion were excluded from the dataset.

Following this process, we identified a total of 179 Android AR apps, comprising 54 ARCore and 125 ARFoundation apps, which form the basis of our empirical study. These apps span a diverse range of domains. As shown in Figure 4, over half of the apps fall within the categories of entertainment, gaming, and education. Furthermore, as illustrated in Figure 5, the dataset encompasses apps with varying levels of popularity and APK sizes, ranging from over 100 million to fewer than 1000 downloads, and from APK sizes exceeding 1 GB to as small as 50 MB.



(a) ARCore Apps **(b) ARFoundation Apps**

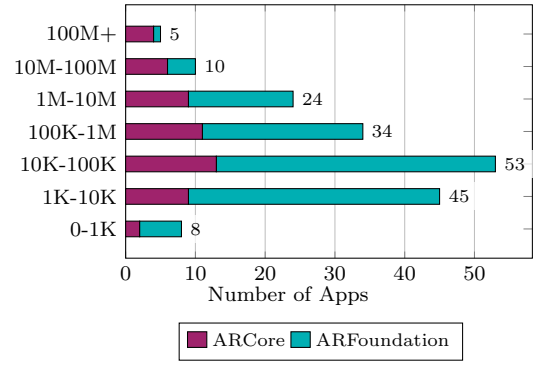
Figure 4: AR app distribution by category domain.

3.2 Data Access Disclosure Examination

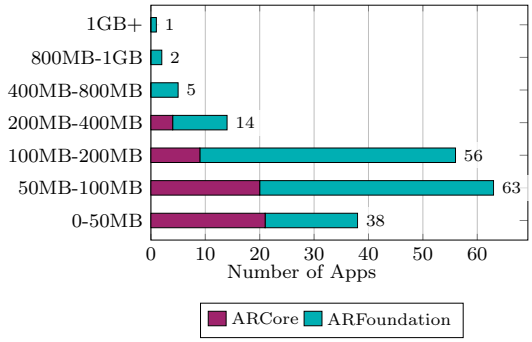
To determine whether sensitive data access was disclosed, we examined three sources for each app: the app description, the privacy policy, and the Google Play Data Safety section. One author manually reviewed all three sources for each app, including cases with

Table 2: ARFoundation events and their descriptions.

Class	Event	Description
ARAnchorManager	anchorsChanged	Triggered when anchors are added, updated, or removed
ARCameraManager	frameReceived	Invoked on each new camera frame update
AREnvironmentProbeManager	environmentProbesChanged	Called when lighting or environmental data updates
ARFace	updated	Triggered when a face's geometry or pose changes
ARFaceManager	facesChanged	Fired when faces are detected, updated, or removed
ARHumanBodyManager	humanBodiesChanged	Initiated when human body tracking data changes
ARMeshManager	meshesChanged	Triggered when mesh geometry updates
AROcclusionManager	frameReceived	Invoked when occlusion or depth data is updated
ARParticipantManager	participantsChanged	Fired when participants join, leave, or change in a session
ARPlane	boundaryChanged	Called when the detected plane's boundary is modified
ARPlaneManager	planesChanged	Triggered when planes are added, updated, or removed
ARPointCloud	updated	Initiated when detected feature points change
ARPointCloudManager	pointCloudsChanged	Called when point cloud data is updated
ARRaycast	updated	Triggered when raycast results change
ARSession	stateChanged	Fired when the AR session state changes
ARTrackedImageManager	trackedImagesChanged	Initiated when tracked image data changes
ARTrackedObjectManager	trackedObjectsChanged	Called when tracked 3D object data updates



(a) Number of Apps per Download Category



(b) Number of Apps per Size Category

Figure 5: AR app distribution by popularity (number of downloads) and APK sizes

vague or ambiguous language. For example, a statement such as "we may collect certain device data" was not considered a disclosure of face tracking, whereas "this app uses your camera to detect facial features" was considered a disclosure. A data access was classified as disclosed if it was explicitly mentioned in at least one of the three sources, and undisclosed otherwise.

3.3 Analysis Pipeline

As mentioned previously, our analysis focuses on AR apps developed with Google ARCore and Unity ARFoundation.

ARCore apps are usually written in Java and can be analyzed using Java-based static analysis tools for Android. In contrast, Unity-based apps are written in C# and are typically compiled into native binaries for Android, requiring a distinct set of binary analysis tools, along with an additional preprocessing step for analysis.

The rest of this section introduces our analysis pipeline, which follows the steps of preprocessing, decompilation, call graph generation, AR-specific call path extraction, code and call path inspection, and instrumentation, as illustrated with Figure 6.

3.3.1 Preprocessing and Metadata Extraction for Unity Apps. Unity apps with IL2CPP backend compile C# scripts into native binaries and strip away readable metadata. To recover this metadata, we employed Il2CppDumper [25], an open-source reverse engineering tool for Unity. It analyzes two key files extracted from the APK: the Unity native library `libil2cpp.so` and the function mapping file `global-metadata.dat`, and generates a JSON file detailing class names, method addresses, field offsets, and type information.

3.3.2 Decompilation. Decompilation serves two key purposes in our analysis. First, the decompiled source code enables inspection of how sensitive data is accessed within AR apps. Second, for Unity-based apps, the decompiled code is directly utilized to construct the call graph, which forms the basis for subsequent program analysis.

For ARCore apps, we used Androguard [2] to decompile their APKs. Specifically, Androguard processes various components, such as configuration files and Dalvik bytecode to produce decompiled source code in both `.java` format and a SMALI-like representation (`.ag` files). It is important to note that the decompiled source code may contain obfuscated class, method, and variable names.

For ARFoundation apps, we used Ghidra [6] to analyze their native binaries, leveraging the JSON output from Il2CppDumper to recover human-readable identifier names. Ghidra generates decompiled C# source code, facilitating effective inspection and analysis of the app's behavior.

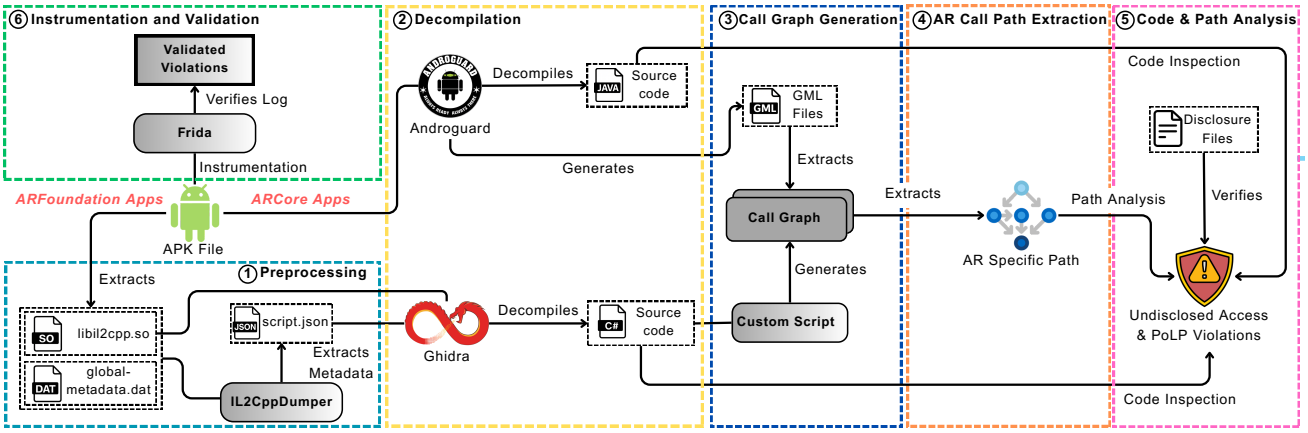


Figure 6: The methodology of our static analysis.

3.3.3 Call Graph (CG) Generation. Call graphs are employed to analyze the flow of sensitive data within AR apps. For ARCore apps, we utilized Androguard, which constructs call graphs as Graph Modeling Language (GML) files, where functions are represented as vertices and function invocations as edges. For Unity-based apps, we developed a custom script to generate call graphs from the Ghidra-decompiled C# source code.

3.3.4 AR-Specific Call Path Extraction. Since this study focuses on AR-specific elements – namely, `trackables` and `anchors` – we extracted call paths that include functions associated with these elements, enabling targeted investigation of their usage within AR apps. As exhaustively evaluating every path within a call graph is computationally expensive, we adopted a reverse search strategy. Specifically, we initiated graph traversal from known trackable or anchor functions, performing a backward traversal to identify partial paths leading to these functions, followed by a forward traversal from the same functions to capture the remaining segments. Combining both traversals, we reconstructed complete call paths for all trackable/anchor function invocations.

3.3.5 Code and Call Path Inspection. The decompiled source code and extracted call paths were then used to address the research questions outlined in Section 1. Specifically, the call paths were used to identify sensitive data accesses by detecting calls to AR-specific trackable classes and APIs, while the decompiled source code enabled detailed examination of how such data is accessed in practice. To identify undisclosed accesses, we cross-checked the detected sensitive data accesses against the three disclosure sources described in Section 3.2. To identify PoLP violations, we examined call paths for generic APIs such as `getAllTrackables()` and broad access APIs such as `acquireCameraImage()`, which grant access to more data than the app's functionality requires.

3.3.6 Instrumentation and Validation. To verify whether flagged sensitive data accesses indeed occur during app execution, we used Frida [5], a dynamic instrumentation toolkit. Specifically, a Frida server was deployed on a Google Pixel 8a running Android 16, and custom JavaScript scripts were loaded to intercept specific method calls in the identified AR apps. The generated logs confirmed whether the flagged accesses occurred at runtime.

4 Undisclosed Data Access Violation

As previously noted, AR apps continuously collect and process visual and spatial information from the user's environment through AR-specific classes. While some apps explicitly mention such access in their app descriptions, privacy policies, or the Google Play Data Safety section, many fail to do so. In this section, we first examine the types and frequency of sensitive data access in our dataset and then investigate the extent to which such access occurs without disclosure, addressing RQ1.

> **RQ1:** To what extent do AR apps access sensitive data without disclosure?

4.1 Data Access Patterns

4.1.1 ARCore Apps. The call graph analysis of ARCore-based Android apps reveals that certain data types are accessed more frequently than others, as shown in Figure 7. The most widely used class is `Frame`, appearing in 39 apps, followed by `Camera` in 38 apps and `Plane` in 35 apps, all commonly used for fundamental AR functionality. Among the more privacy-sensitive classes, `AugmentedImage` is used in 8 apps and `AugmentedFace` in 5 apps, reflecting access to image-based and facial data respectively. Other classes such as `Point` (3 apps), `Earth` and `InstantPlacementPoint` (1 app each) are rarely used, while `DepthPoint`, `GeospatialPose`, and `StreetscapeGeometry` are not used by any app in our dataset. Note that Figure 7 reports direct references to a class in app code. A class may still appear in an app's call paths through generic SDK APIs; for example, `Frame.getUpdatedTrackables()` may internally constructs `DepthPoint` objects.

4.1.2 ARFoundation Apps. Compared to ARCore, ARFoundation apps show different data access patterns, although the core types of AR data accessed remain similar. As shown in Figure 8, `ARPlane` is the most widely used class, appearing in 92 apps (out of 125 apps), followed by `ARRaycast` in 86 apps and `ARCamera` in 66 apps.

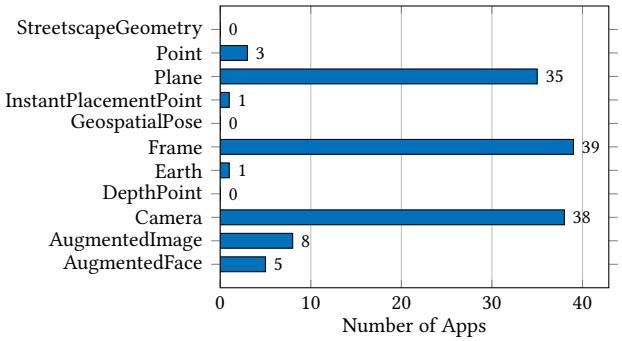


Figure 7: The numbers of ARCore apps accessing different types of AR-specific data. There are 54 ARCore Apps in total. Descriptions of these classes are given in Table 1.

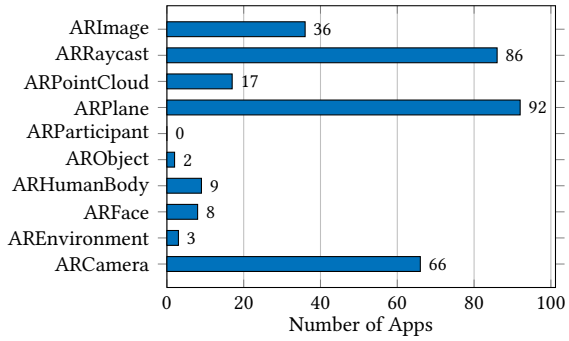


Figure 8: The numbers of ARFoundation apps accessing different types of AR-specific data. 125 ARFoundation Apps in total. Descriptions of these classes are given in Table 1.

These classes support core AR operations such as surface detection and user interaction. Privacy-sensitive classes, such as, `ARImage` is used in 36 apps, `ARPointCloud` in 17 apps, `ARHumanBody` in 9 apps, and `ARFace` in 8 apps, reflecting access to image, spatial, skeletal, and facial data respectively. `AREnvironment` and `ARObject` are seldom used, appearing in only 3 and 2 apps respectively, while `ARParticipant` is not used by any app in our dataset.

4.1.3 Summary. Overall, ARFoundation apps access a broader range of sensitive data types than ARCore apps, including full-body tracking via `ARHumanBody`, which has no equivalent in ARCore. In both SDKs, camera-related classes are among the most widely used, reflecting the heavy reliance of AR apps on visual data.

Table 3: Number of apps with undisclosed data access.

SDK	Data Type	# of Apps with Undisclosed Access	
		Static Analysis	Instrumentation Verified
ARCore	Face	2	2
	Image	7	2
ARFoundation	Face	7	2
	Image	34	16
	Skeletal	8	1
Total		**58**	**23**

4.2 Undisclosed Data Access

4.2.1 ARCore Apps. Among the ARCore apps that access sensitive data, a notable proportion fail to disclose such access to users. Table 3 presents the number of ARCore apps with undisclosed access to sensitive data. Overall, we identified 9 instances of undisclosed access. Specifically, 2 apps accessing `AugmentedFace` do not disclose this access, both of them were confirmed through instrumented execution, and 7 apps accessing `AugmentedImage` do not disclose this access, with 2 confirmed at runtime. These results indicate that existing ARCore apps indeed access sensitive facial and image data while fail to inform users about such access, raising significant privacy concerns.

4.2.2 ARFoundation Apps. As shown in Table 3, ARFoundation apps exhibit a significantly higher number of undisclosed accesses than ARCore, with 49 instances identified across 125 apps. This is particularly evident for `ARImage`, where 34 apps do not disclose image data access, with 16 confirmed at runtime. Similarly, 7 apps using `ARFace` and 8 apps using `ARHumanBody` fail to disclose such access, with 2 and 1 confirmed through instrumented execution respectively. Notably, `ARHumanBody` has no equivalent in ARCore, representing an additional category of undisclosed sensitive data unique to ARFoundation apps.

4.2.3 Summary. Overall, we identified 58 instances of undisclosed sensitive data access across both SDKs, with ARFoundation apps exhibiting a significantly higher prevalence than ARCore apps, likely due to the broader range of sensitive data types supported by ARFoundation. Of these, 23 were confirmed through dynamic instrumentation; the remainder could not be verified due to specific runtime conditions not present in our test environment, as further discussed in Section 8. Most app descriptions provide vague or minimal information about AR data usage, making it difficult for users to make informed privacy decisions. These findings highlight the need for clearer disclosure requirements and stronger enforcement mechanisms in AR app development.

5 PoLP Violation

While Section 4 examines undisclosed data access violations, this section investigates a distinct but equally important privacy concern: PoLP violations, addressing RQ2. Even when data access is disclosed, apps accessing more data than their functionality requires pose significant privacy risks. We identify two main forms of PoLP violations in AR apps: (1) access to full camera frames through broad APIs, and (2) access to all available trackables through generic SDK APIs. The latter is exclusive to ARCore, as ARFoundation does not provide equivalent generic APIs.

RQ2: To what extent do AR apps violate the Principle of Least Privilege (PoLP) by accessing more data than their functionality requires?

5.1 Full Camera Frame Access

Accessing the full camera frame is a major and common PoLP violation, as it exposes all visual information in the user's surroundings regardless of what the app actually needs. For example, an AR

```
1  private boolean fillARFrameInfo(Frame frame) throws Throwable {
2      Image imageAcquireCameraImage;
3      float f6;
4      if (frame == null) {
5          return false;
6      }
7      if (this.mARFrameInfo == null) {
8          this.mARFrameInfo = new ARFrameInfo();
9      }
10     Image image = null;
11     try {
12         imageAcquireCameraImage = frame.acquireCameraImage() // Direct
                   access to camera image
13     }
14     ...
15 }
```

Code Snippet 1: Accessing raw camera image with ARCore.

app designed for furniture placement only needs plane detection data, yet both ARCore and ARFoundation provide APIs that grant unrestricted access to the full camera frame.

Table 4: Number of apps with PoLP violations.

SDK	Data Type	# of Apps with PoLP Violation	
		Static Analysis	Instrumentation Verified
ARCore	Full Camera Frame	18	7
	All Trackables	36	27
ARFoundation	Full Camera Frame	28	6
Total		**82**	**40**

5.1.1 ARCore Apps. In ARCore, full camera image access is granted through the `Frame.acquireCameraImage()` API, as illustrated in Code Snippet 1 (line 12). This API returns the full raw camera image, exposing all visual information in the user's surroundings regardless of the app's actual data requirements. As shown in Table 4, our analysis identified 18 ARCore apps that access full camera frames through this API, with 7 confirmed through instrumented execution, constituting a PoLP violation for apps that do not require such broad visual access to fulfill their intended functionality.

5.1.2 ARFoundation Apps. In ARFoundation, the equivalent API is `ARCameraManager.TryAcquireLatestCpuImage()`, which grants access to the full raw camera image. As shown in Table 4, 28 ARFoundation apps access full camera frames through this API, with 6 confirmed through instrumented execution. The higher number of affected apps compared to ARCore reflects the widespread use of `ARCamera` in ARFoundation apps, as shown in Section 4.

5.1.3 Summary. In total, 46 apps across both SDKs access full camera frames through broad APIs, granting unrestricted access to all visual information in the user's surroundings. This represents a significant PoLP violation, as most of these apps do not require access to the full camera image to fulfill their intended functionality.

5.2 Access via Generic API

Unlike explicit access of specific typed objects, there exist generic APIs that can return all available trackables regardless of type, exposing sensitive data beyond what the app requires.

5.2.1 ARCore Apps. In addition to full camera image access, ARCore apps can also violate PoLP by accessing all available trackables via generic APIs.

```
1  public void onDrawFrame(GL10 gl10) {
2      ...
3      try {
4          Session session3 = this.session;
5          ...
6          Collection<AugmentedFace> allTrackables = session3.getAllTrackables(
               AugmentedFace.class);
7          ...
8          for (AugmentedFace augmentedFace : allTrackables) {
9              if (augmentedFace.getTrackingState() != TrackingState.TRACKING)
                   {
10                 break;
11             }
12             ...
13             float[] fArr4 = new float[16];
14             augmentedFace.getCenterPose().toMatrix(fArr4, 0);
15             this.augmentedFaceRenderer.draw(fArr, fArr2, fArr4, fArr3,
                   augmentedFace);
16         }
17     } catch (Throwable th) {
18         ...
19     }
20     ...
21 }
```

Code Snippet 2: ARCore's `getAllTrackables` gives AR apps full access to any typed trackables.

```
1  public boolean j(String str, r5 r5Var) {
2      for (AugmentedImage augmentedImage : this.f28043y.getUpdatedTrackables(
           AugmentedImage.class)) {
3          if (str.equals(augmentedImage.getName())) {
4              q0(augmentedImage.getCenterPose(), r5Var);
5              return true;
6          }
7      }
8      return false;
9  }
```

Code Snippet 3: ARCore's `getUpdatedTrackables` gives AR apps full access to any typed trackables.

```
1  public final boolean j(com.google.ar.core.HitResult p6, Float p7) {
2      long v0_0 = p6.getTrackable(); // Retrieve Trackable object
3      if (v0_0.getTrackingState() == com.google.ar.core.TrackingState.TRACKING
           ) {
4          if (v0_0 instanceof com.google.ar.core.Plane) {
5              if (((com.google.ar.core.Plane) v0_0).isPoseInPolygon(p6.
                   getHitPose())) {
6                  // Handle distance and plane validation
7                  return 1;
8              }
9          }
10     }
11     return 0;
12 }
```

Code Snippet 4: Hit test results in ARCore give AR apps full access to any typed trackables.

A notable example is the `getAllTrackables` API from ARCore's `Session` class. As shown in Code Snippet 2, at line 6 the app retrieves all detected faces by passing `AugmentedFace.class` to `Session.getAllTrackables()`, then iterates over them to check their tracking state (lines 8–11). Crucially, passing `Trackable.class` instead returns all trackables of all types, enabling broad implicit access without explicitly declaring the required data types.

A similar issue arises with `getUpdatedTrackables` from ARCore's `Frame` class. As shown in Code Snippet 3, at line 2 the app retrieves updated image trackables, but passing `Trackable.class` instead would return all updated trackables, enabling access to a broad set of AR data.

Broad access also occurs during hit tests. When a user taps the screen, `hitTest` returns a list of `HitResult` objects, each corresponds to a trackable of any trackable type. As shown in Code Snippet 4, at line 2 the app retrieves the trackable from a `HitResult`, and line 4 confirms it can be of any type, potentially exposing sensitive objects without user awareness (i.e., the user may not be aware that tapping screen can grant data access to AR app).

Table 5: Distribution of violations across app categories.

Category	# of Apps	Undisclosed Access	PoLP Violation
Art & Design	7	3	1
Business	11	0	5
Education	39	16	11
Entertainment	38	13	15
Game	30	3	4
House & Home	4	6	3
Lifestyle	8	4	4
Navigation & Travel	9	1	3
Productivity	11	4	7
Shopping	3	1	2
Tools	15	6	9
Others	4	1	1
Total	**179**	**58**	**65**

Such accesses are typically undisclosed by AR apps and violate PoLP as they give apps access to data beyond what their functionality requires. As shown in Table 4, 36 ARCore apps access sensitive data through these generic APIs, with 27 confirmed through instrumented execution.

Note that while hit tests can potentially expose sensitive information, none of the apps in our analysis passed the generic `Trackable.class` to either of these two APIs: `getAllTrackables` and `getUpdatedTrackables`. However, the fact that the current ARCore SDK supports such generic access indicates that PoLP violations are inherently permitted by the SDK design, posing a latent privacy risk even if not currently exploited by the apps in our dataset.

5.2.2 Summary. In total, 36 ARCore apps violate PoLP through generic APIs, with 27 confirmed at runtime. This form of violation is exclusive to ARCore, as ARFoundation does not provide equivalent generic APIs. However, ARFoundation apps remain susceptible to PoLP violations through full camera frame access, as discussed in the previous subsection. It is worth noting that 17 ARCore apps violate PoLP through both full camera frame access and generic APIs.

6 Violation Patterns Across App Categories

As discussed in Sections 4 and 5, we examined RQ1 and RQ2, particularly undisclosed access and PoLP violations by SDK. Both questions can also be examined in terms of application domains, specifically whether certain types of AR apps are more prone to undisclosed data access or excessive data access. This section revisits both findings from that perspective.

Table 5 reports, for each app category, the number of apps in our dataset together with the undisclosed accesses and PoLP violations attributed to them. Note that the PoLP violation counts sum to 65 rather than the 82 reported in Table 4. This is because 17 ARCore apps violate PoLP through both full camera frame access and generic APIs, and are counted once per app here but once per violation type there.

As shown in Table 5, undisclosed accesses are not evenly distributed among application categories. Education and Entertainment show the highest counts, with 16 and 13 undisclosed accesses respectively, though this largely reflects their size as the two largest categories in our dataset. Relative to category size, House & Home and Lifestyle are particularly noteworthy: House & Home shows 6 accesses across only 4 apps, while Lifestyle shows 4 accesses across 8 apps. Both domains involve integrating virtual content with real-world surfaces and human figures, encouraging image and face trackable use that they rarely disclose. The Games category show the opposite pattern, with only 3 accesses across 30 apps, consistent with most game apps in the dataset relying on plane detection alone.

On the other hand, PoLP violations are spread more evenly. Entertainment and Education again show the highest counts, with 15 and 11 violations respectively. However, House & Home, Shopping, and Productivity show the highest proportions: House & Home reveals 3 violations across 4 apps, Shopping shows 2 across 3 apps, and Productivity shows 7 across 11 apps, reflecting their reliance on full camera frame access for scanning and preview features.

In summary, these patterns suggest that privacy risk in AR applications also varies by application domain, in addition to the SDK-level differences reported in Sections 4 and 5. This insight suggests that a category-aware auditing approach could help prioritize the domains where sensitive trackable elements are most likely to be accessed without adequate disclosure, as well as those where excessive access is most likely to occur.

7 Potential Privacy Protection Strategies

As outlined earlier, AR apps frequently access sensitive data without adequate disclosure and often access more data than their functionality requires. These findings raise the question of how such privacy risks can be mitigated in practice. In this section, we analyze potential privacy protection strategies for existing AR apps based on our study results, addressing RQ3.

> **RQ3:** What are the potential strategies to protect user privacy for existing AR apps?

7.1 Fine-Grained Access Control

The coarse-grained access model in the current AR SDKs allows apps to access sensitive data without explicitly specifying the permissions. For example, an outdoor AR navigation app should not have access to facial information, yet the lack of granular permissions allows such access by default. Introducing fine-grained access control will effectively limit unnecessary data exposure, improve transparency, and reduce the attack surface for potential misuse by requiring apps to explicitly request and justify access to specific types of AR data (e.g., camera, faces, images). However, prior work has shown that fine-grained access control does not always translate into improved security and privacy [15], as overly fine-grained permissions may lead to consent fatigue for ordinary users, making it necessary to complement this approach with automated solutions as discussed below.

7.2 AR SDK API Redesign

A key insight from our findings is that PoLP violations in AR apps are not solely due to developer negligence but are also a consequence of how AR SDKs are designed. Generic APIs such as

`Session.getAllTrackables()`, `Frame.getUpdatedTrackables()`, `HitResult.getTrackable()`, and broad-access APIs such as `Frame.acquireCameraImage()` are the primary sources of PoLP violations, allowing apps to access more data than necessary. A fundamental solution is to redesign AR SDK APIs to enforce PoLP by default, by restricting generic APIs to type-specific alternatives only and redesigning camera frame access APIs to provide different levels of granularity. This would shift the responsibility of PoLP enforcement from individual developers to the SDK itself, making privacy-respecting behavior the default rather than the exception.

7.3 Automated Disclosure Verification

Our finding of 58 instances of undisclosed sensitive data access highlights the inadequacy of the current self-reported disclosure system, where developers are trusted to accurately report their data access practices without any automated verification. A more effective approach is to integrate automated disclosure verification into the app store submission pipeline, where static analysis is used to identify sensitive AR data accesses and cross-check them against the app description, privacy policy, and Google Play Data Safety section before publication. Apps that fail this verification would be required to update their disclosures before being published. This shifts the verification burden from users to the platform, ensuring that disclosures accurately reflect actual app behavior and reducing the risk of users being misled about how their data is being used.

7.4 Privacy-Oriented Code Audit and Refactoring

The control-flow and data-flow analysis, well-studied for traditional Android apps [22, 35–37], can also be applied to AR apps to track how sensitive data are accessed and propagated throughout the app. The analysis results can help app stores flag privacy-violating apps, notify users, and help developers refactor their code for better data access practices. However, such analyses face known limitations, including imprecise static models, high dynamic overhead, and false positives and negatives. To assess their feasibility for AR apps, we analyzed the previously extracted call paths. As shown in Figures 9 and 10, all AR-specific data classes have call paths shared across multiple apps — 236 out of 620 in ARCore and 81 out of 134 in ARFoundation. These shared patterns have four key implications.

First, the relatively small number of AR-specific call paths — only a few hundred per SDK — enables static and dynamic analyses to focus on a manageable subset of code relevant to AR data, significantly reducing analysis time and improving scalability.

Second, because most AR-specific data types involve only a limited number of paths, dynamic analysis targeting a specific data type can be implemented with relatively low runtime overhead, making it practical for real-world deployment.

Third, common call paths across multiple apps enable the development of reliable control/data-flow models and the training of machine learning models for anomaly detection, reducing false positives and negatives while improving the accuracy of privacy violation detection.

Fourth, the existence of common invocation chains indicates the feasibility of automatic code refactoring techniques to transform

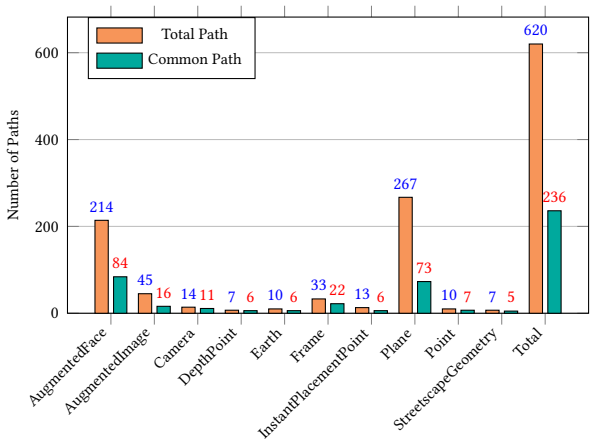


Figure 9: ARCore Class Path Summary

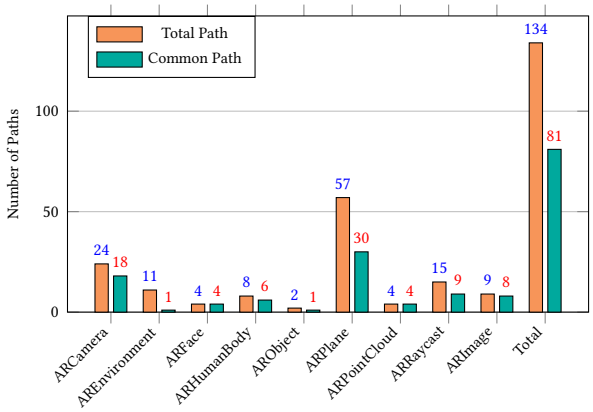


Figure 10: ARFoundation Class Path Summary

privacy-violating code that enforce PoLP, without requiring manual developer intervention.

8 Discussion and Future Work

Limitation of Static Analysis on ARCore-based Apps. For ARCore-based apps, our analysis relied on static program analysis of Dalvik bytecode, which is inherently constrained by the limitations of static analysis. While occurrences of reflection and dynamic class loading are relatively rare in these apps, such behaviors fall outside the scope of our analysis. Additionally, code obfuscation further complicated the interpretation of decompiled Java code, and the absence of runtime context limited our ability to accurately capture active execution paths, potentially resulting in false positives (inactive paths) or missed paths. To partially address these limitations and validate our static findings, we conducted instrumented executions for apps flagged with privacy violations, confirming that the flagged accesses occurred at runtime. A complete dynamic analysis of all AR apps is left for future work.

Limitation of Static Analysis on ARFoundation-based Apps. The above limitations also apply to ARFoundation-based apps. Since Unity compiles C# scripts into native binaries, the APKs of ARFoundation apps contain mostly natively compiled code, resulting

in a significant number of dynamic dispatches. However, these dispatches typically occur at the ARFoundation-ARCore interface rather than within app logic, limiting their impact on our analysis. Nonetheless, to improve coverage and accurcay, we plan to apply dynamic analysis to ARFoundation apps and investigate the ARFoundation-ARCore interface to understand how sensitive data is exchanged between these SDKs in future work.

Limitation of Dynamic Verification. The accesses confirmed through dynamic instrumentation represent a lower bound of actual privacy violations, as many sensitive data accesses are triggered only under specific runtime conditions, such as the presence of a particular image marker, that may not have been present in our test environment. More generally, it is often unclear what specific conditions are required to trigger the execution of certain data access functions. Thus, the absence of runtime confirmation does not imply that the access never occurs. Static analysis complements this by providing an upper bound, identifying all potential access paths regardless of runtime conditions.

Other AR SDKs. Our analysis focused on ARCore and ARFoundation, the two most widely used SDKs in our dataset. However, other SDKs such as Vuforia and EasyAR are also present though have fewer apps using them, and we plan to explore them in future work. Additionally, while this study focuses on Android, iOS AR apps typically rely on ARKit and may exhibit different behaviors. Nevertheless, many iOS AR apps are developed using Unity/ARFoundation, and thus our findings are expected to be broadly applicable across platforms.

9 Related Work

Several studies have examined user privacy concerns in mobile AR apps. Yang and Zhang [38] conducted a large-scale empirical study focused on dangerous permission requests and traditional data leaks such as location, identifiers, and contacts, but their analysis does not address AR-specific data access such as facial tracking or environmental understanding, which operate beyond manifest-declared permissions.

Moreover, Lehman et al. [21] introduced the concept of hidden operations in AR apps, demonstrating that malicious developers can embed covert computer vision tasks to collect personal information. While their study illustrates the privacy risks of undeclared operations, it relies solely on synthetic prototypes and does not examine whether such operations occur in real-world AR applications.

To enforce privacy at a more granular level, Kim et al. [19] proposed Erebus, a fine-grained access control system that intercepts ARCore API calls and enforces developer-specified rules through a domain-specific language. While Erebus shows promise, it requires SDK modifications, has only been validated on a small set of apps, and focuses solely on ARCore. Similarly, Abraham et al. [8] introduced a user-centered permission model for AR headsets that allows users to control data granularity through visual sliders, but does not address privacy risks in current mobile AR platforms or analyze real-world app behaviors. Furthermore, as discussed in Section 7, the diversity of AR data types can make configuring access permissions overly burdensome for average users, while determined malicious apps may still find ways to bypass these controls.

Guo et al. [16] conducted a comprehensive analysis of Oculus applications, revealing undeclared access to biometric functions including hand and eye tracking. Although methodologically rigorous, their study is limited to VR platforms, which differ from AR in their interaction models and data acquisition paths.

Other works have targeted privacy awareness and interface-level adaptations. Harborth and Frik [17] examined how contextual justifications in permission prompts could improve user trust, while Alghamdi and Mohaisen [12] used LLM-based methods to analyze AR/VR app privacy policies, and Rajaram et al. [27] introduced privacy-driven adaptation techniques for AR apps. However, these works do not detect privacy violations in current apps and are largely orthogonal to our work.

To the best of our knowledge, our work is the first to use static code analysis across both Java and native layers to systematically detect AR-specific privacy risks in existing applications, revealing how physical-world data can be collected without user awareness or manifest transparency.

10 Conclusion

This paper presented a comprehensive empirical study of privacy risks in Android AR apps, revealing widespread undisclosed access to sensitive real-world data such as facial features, 2D images, and camera frames, as well as frequent violations of the Principle of Least Privilege through overly broad API usage. Our analysis identified 58 instances of undisclosed sensitive data access and 82 PoLP violations across 179 ARCore and ARFoundation apps. To mitigate these risks, we proposed four complementary privacy protection strategies: fine-grained access control, AR SDK API redesign, automated disclosure verification, and privacy-oriented code audit and refactoring. Our analysis of shared call paths — 236 out of 620 in ARCore and 81 out of 134 in ARFoundation — suggests that static and dynamic analysis paired with anomaly detection models can provide effective privacy protection with manageable overhead. Overall, our findings underscore the urgent need for stronger privacy safeguards in the AR ecosystem, both at the application and SDK levels.

Data Availability Statement

Replication packages, including our analysis scripts, the complete flagged-app list, and all reported results, are made available on Zenodo [23] at https://doi.org/10.5281/zenodo.21753818.

Acknowledgments

This work was partially supported by National Science Foundation (NSF) grants: 2221843, 2318486, 2155096, 2215359, and 2418093. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied by NSF. This paper was edited with the assistance of LLM models to refine the writing. The authors would like to thank the anonymous reviewers for their insightful comments.

References

[1] 2025. *Build new augmented reality experiences that seamlessly blend the digital and physical worlds | ARCore.* https://developers.google.com/ar

[2] 2025. *Welcome to Androguard's documentation! — Androguard 3.4.0 documentation.* https://androguard.readthedocs.io/en/latest/index.html

[3] 2019. *OWASP Foundation 2019 Top 10 API Security Risks.* https://owasp.org/API-Security/editions/2019/en/0x11-t10/

[4] 2023. *OWASP Foundation 2023 Top 10 API Security Risks.* https://owasp.org/API-Security/editions/2023/en/0x11-t10/

[5] 2025. Frida • A world-class dynamic instrumentation toolkit. https://frida.re/. Accessed: 2025-09-08.

[6] 2025. Ghidra. https://github.com/NationalSecurityAgency/ghidra. Accessed: 2025-05-28. Originally released: 2019-03-01.

[7] 2026. APKPure – Download APK files for Android. https://apkpure.com. Accessed: 2026-03-25.

[8] Melvin Abraham, Mark McGill, and Mohamed Khamis. 2024. What You Experience is What We Collect: User Experience Based Fine-Grained Permissions for Everyday Augmented Reality. In *Proceedings of the CHI Conference on Human Factors in Computing Systems* (Honolulu HI USA, 2024-05-11). ACM, 1–24. doi:10.1145/3613904.3642668

[9] Devon Adams, Alseny Bah, Catherine Barwulor, Nureli Musabay, Kadeem Pitkin, and Elissa M. Redmiles. 2018. Ethics Emerging: the Story of Privacy and Security Perceptions in Virtual Reality. In *Proceedings of the Fourteenth USENIX Conference on Usable Privacy and Security*.

[10] Edidiong Elijah Akpan. 2024. Healthcare Applications of Augmented Reality. 201 (2024). Publisher: IGI Global. https://books.google.com/books?hl=en&lr=&id=hOghEQAAQBAJ&oi=fnd&pg=PA201&dq=Healthcare+Applications+of+Augmented+Reality(AR)+and+Virtual+Reality+(VR):+Immersive+Simulation+in+Medical-+Clinical+Education&ots=ENWuVigxyy&sig=v-PyaHWUWDVWE8NBQDjBWC91fsI

[11] Murat Akçayır and Gökçe Akçayır. 2017. Advantages and challenges associated with augmented reality for education: A systematic review of the literature. 20 (2017), 1–11. Publisher: Elsevier. doi:10.1016/j.edurev.2016.11.002

[12] Abdulaziz Alghamdi and David Mohaisen. 2024. Through the Looking Glass: LLM-Based Analysis of AR/VR Android Applications Privacy Policies. In *2024 International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 534–539.

[13] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. 49, 6 (2014), 259–269. doi:10.1145/2666356.2594299

[14] Tamara Denning, Zakariya Dehlawi, and Tadayoshi Kohno. 2014. In situ with bystanders of augmented reality glasses: perspectives on recording and privacy-mediating technologies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto Ontario Canada, 2014-04-26). ACM, 2377–2386. doi:10.1145/2556288.2557352

[15] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. 2012. Android permissions: user attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*. doi:10.1145/2335356.2335360

[16] Hanyang Guo, Hong-Ning Dai, Xiapu Luo, Zibin Zheng, Gengyang Xu, and Fengliang He. 2024. An Empirical Study on Oculus Virtual Reality Applications: Security and Privacy Perspectives. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon Portugal, 2024-04-12). ACM, 1–13. doi:10.1145/3597503.3639082

[17] David Harborth and Alisa Frik. 2021. Evaluating and redefining smartphone permissions with contextualized justifications for mobile augmented reality apps. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)* (2021). 513–534. https://www.usenix.org/conference/soups2021/presentation/harborth

[18] Apple Inc. [n. d.]. *ARKit 6 - Augmented Reality.* https://developer.apple.com/augmented-reality/arkit/

[19] Yoonsang Kim, Sanket Goutam, Amir Rahmati, and Arie Kaufman. 2023. Erebus: Access control for augmented reality systems. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023). 929–946. https://www.usenix.org/conference/usenixsecurity23/presentation/kim-yoonsang

[20] Yiannis Koumpouros. 2024. Revealing the true potential and prospects of augmented reality in education. 11, 1 (2024), 2. doi:10.1186/s40561-023-00288-0

[21] Sarah M. Lehman, Abrar S. Alrumayh, Kunal Kolhe, Haibin Ling, and Chiu C. Tan. 2022. Hidden in Plain Sight: Exploring Privacy Risks of Mobile Augmented Reality Applications. 25, 4 (2022), 1–35. doi:10.1145/3524020

[22] Changlin Liu, Hanlin Wang, Tianming Liu, Diandian Gu, Yun Ma, Haoyu Wang, and Xusheng Xiao. 2022. ProMal: precise window transition graphs for android via synergy of program analysis and machine learning. In *Proceedings of the 44th International Conference on Software Engineering*. 1755–1767. doi:10.1109/ICSE-Companion52605.2021.00061

[23] Sabbir Hussain Meraj, Long Trac, Xiaoyin Wang, Xusheng Xiao, and Wei Wang. 2026. Artifact for: An Empirical Study of Data Access Practice in Android AR Apps to Understand User Privacy Risks. doi:10.5281/zenodo.21753818

[24] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to android inter-component communication analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015), Vol. 1. IEEE, 77–88. https://ieeexplore.ieee.org/abstract/document/7194563/

[25] Perfare. 2025. Il2CppDumper. https://github.com/Perfare/Il2CppDumper. Accessed: 2025-05-28. Originally released: 2016-12-30.

[26] Atieh Poushneh and Arturo Z. Vasquez-Parraga. 2017. Discernible impact of augmented reality on retail customer's experience, satisfaction and willingness to buy. 34 (2017), 229–234. Publisher: Elsevier. https://www.sciencedirect.com/science/article/pii/S0969698916304477

[27] Shwetha Rajaram, Macarena Peralta, Janet G Johnson, and Michae l Nebeling. 2025. Exploring the Design Space of Privacy-Driven Adaptation Techniques for Future Augmented Reality Interfaces. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–19. doi:10.1145/3706598.3713320

[28] Philipp A. Rauschnabel, Alexander Rossmann, and M. Claudia tom Dieck. 2017. An adoption framework for mobile augmented reality games: The case of Pokémon Go. 76 (2017), 276–286. Publisher: Elsevier. doi:10.1016/j.chb.2017.07.030

[29] Nisarg Raval, Animesh Srivastava, Ali Razeen, Kiron Lebeck, Ashwin Machanavajjhala, and Lanodn P. Cox. 2016. What You Mark is What Apps See. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services* (Singapore Singapore, 2016-06-20). ACM, 249–261. doi:10.1145/2906388.2906405

[30] Franziska Roesner, David Molnar, Alexander Moshchuk, Tadayoshi Kohno, and Helen J. Wang. 2014. World-Driven Access Control for Continuous Sensing. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale Arizona USA, 2014-11-03). ACM, 1169–1181. doi:10.1145/2660267.2660319

[31] Atanas Rountev and Dacong Yan. 2014. Static Reference Analysis for GUI Objects in Android Software. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando FL USA, 2014-02-15). ACM, 143–153. doi:10.1145/2544137.2544159

[32] Unity Technologies. 2019. AR Foundation. https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@5.0/manual/index.html.

[33] Robert Templeman, Mohammed Korayem, David J. Crandall, and Apu Kapadia. 2014. PlaceAvoider: Steering First-Person Cameras away from Sensitive Spaces.. In *NDSS* (2014), Vol. 14. Citeseer, 23–26. doi:10.14722/ndss.2014.23014

[34] John Vilk, David Molnar, Benjamin Livshits, Eyal Ofek, Chris Rossbach, Alexander Moshchuk, Helen J. Wang, and Ran Gal. 2015. SurroundWeb: Mitigating privacy concerns in a 3D web browser. In *2015 IEEE Symposium on Security and Privacy* (2015). IEEE, 431–446. doi:10.1109/SP.2015.33

[35] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. 25, 4 (2018), 833–873. doi:10.1007/s10515-018-0237-6

[36] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static control-flow analysis of user-driven callbacks in Android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (2015), Vol. 1. IEEE, 89–99. https://ieeexplore.ieee.org/abstract/document/7194564/

[37] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. 2015. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 303–313. doi:10.1109/ICSE.2015.50

[38] Xiaoyi Yang and Xueling Zhang. 2022. A Study of User Privacy in Android Mobile AR Apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester MI USA, 2022-10-10). ACM, 1–5. doi:10.1145/3551349.3560512

[39] Andy Wai Kan Yeung, Anela Tosevska, Elisabeth Klager, Fabian Eibensteiner, Daniel Laxar, Jivko Stoyanov, Marija Glisic, Sebastian Zeiner, Stefan Tino Kulnik, and Rik Crutzen. 2021. Virtual and augmented reality applications in medicine: analysis of the scientific literature. 23, 2 (2021), e25499. Publisher: JMIR Publications Inc., Toronto, Canada. https://www.jmir.org/2021/2/e25499/1000

[40] Shaowei Zhu, Hyo Jin Kim, Maurizio Monge, G. Edward Suh, Armin Alaghi, Brandon Reagen, and Vincent Lee. 2022. Verifiable Access Control for Augmented Reality Localization and Mapping. arXiv:2203.13308 [cs] doi:10.48550/arXiv.2203.13308

[41] Á. Zsila, G. Orosz, B. Bőthe, I. Tóth-Király, O. Király, M. Griffiths, and Z. Demetrovics. 2018. An Empirical Study on the Motivations Underlying Augmented Reality Games: The Case of Pokémon Go During and After Pokémon Fever. *Personality and Individual Differences* 133 (2018), 56–66. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0191886917304117. doi:10.1016/j.paid.2017.06.024

Received 2026-03-26; accepted 2026-06-18