AVP-Inspect: Coordinated Cyber-Physical Testing for Privacy Analysis of COTS Apple Vision Pro Applications

Yichang Xiong
George Mason University
Fairfax, Virginia, USA
yxiong2@gmu.edu

Vamsi Shankar Simhadri
George Mason University
Fairfax, Virginia, USA
vsimhadr@gmu.edu

Yue Xiao
William & Mary
Williamsburg, Virginia, USA
yxiao05@wm.edu

Xiaokuan Zhang
George Mason University
Fairfax, Virginia, USA
xiaokuan@gmu.edu

Abstract

XR devices introduce substantial privacy concerns due to their comprehensive data collection capabilities that surpass traditional computing platforms. While existing works have demonstrated privacy concerns on Android-based XR devices such as Meta Quest series by performing network traffic analysis, little attention has been paid to the Apple Vision Pro (AVP) devices, mainly due to the closed nature and the technical challenges associated with AVP devices. In this work, we make a bold attempt to detect privacy violations of AVP applications from network traffic through automatic testing on AVP devices. Our key insight is that effective AVP application testing requires coordinated control of both cyber (software) and physical (hardware) components, which we term *Coordinated Cyber-Physical Testing*.

Building on this insight, we design and implement AVP-Inspect, an automatic dynamic analysis framework for AVP applications, overcoming significant challenges enforced by the closed-source nature of AVP ecosystem. AVP-Inspect consists of three components: an automatic device controller by building customized hardware devices, a 3D UI explorer by designing a new exploration engine, and a privacy violation detector by constructing a unified privacy taxonomy for AVP. We first evaluated AVP-Inspect on a manually constructed ground truth dataset, then performed a large-scale analysis on 324 AVP applications downloaded from the App Store, with each app tested for 20 minutes. We found that 188 (58.0%) of apps exhibit at least one violation, and more than 60% of the network traffic flows are not properly disclosed.

CCS Concepts

• **Security and privacy → Software security engineering**; **Mobile and wireless security**.

Keywords

Extended Reality, Traffic, Privacy Compliance, Apple Vision Pro



This work is licensed under a Creative Commons Attribution 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3846586

ACM Reference Format:
Yichang Xiong, Vamsi Shankar Simhadri, Yue Xiao, and Xiaokuan Zhang. 2026. AVP-Inspect: Coordinated Cyber-Physical Testing for Privacy Analysis of COTS Apple Vision Pro Applications. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3830454.3846586

1 Introduction

Extended Reality (XR) technology is emerging as a fundamental computing paradigm that merges physical and digital worlds, with market projections reaching $472.39 billion by 2029 [1]. XR encompasses mixed reality (MR), which combines augmented reality (AR) and virtual reality (VR) to enable real-time interactions between physical and virtual objects. This integration creates immersive environments that enhance user interaction and experience. XR devices have demonstrated practical value across multiple domains, including healthcare, education, entertainment, and professional training [2–5]. The recent introduction of Apple Vision Pro (AVP) [6] represents a significant development in the XR landscape, potentially accelerating mainstream adoption in both consumer and enterprise markets. Recent deployments of XR devices for surgical assistance [7] further demonstrate XR's emergence as a revolutionary computing platform that fundamentally transforms human-technology interaction.

Although XR devices bring a lot of benefits, they introduce substantial privacy concerns due to their comprehensive data collection capabilities that surpass traditional computing platforms. These devices continuously gather extensive biometric and behavioral data, including eye movements, facial expressions, hand gestures, and full body movements. Recent research has demonstrated how this data can be exploited to extract sensitive information such as passwords, personal characteristics, and health conditions [8–14]. Moreover, XR applications must comply with privacy regulations such as the General Data Protection Regulation (GDPR) [15] and California Consumer Privacy Act (CCPA) [16], by transparently and correctly documenting their data collection, processing, and sharing procedures through comprehensive privacy policies.

While existing research has made important progress in understanding privacy risks in popular XR platforms such as Meta's Quest series [17, 18], their methods cannot be adapted to AVP due to significant differences in both the hardware and software stacks. The

closed-source nature of commercial XR platforms presents significant technical barriers to privacy analysis. This technical obstacle is especially pronounced with AVP due to Apple's restrictive policies regarding third-party system analysis. Unlike iOS, which has benefited from two decades of community-driven security research and analysis tools (*e.g.*, jailbreaking techniques for root access [19] and dynamic analysis tools [20, 21]), AVP was released in early 2024 and lacks established security testing frameworks. Consequently, there exists no comprehensive privacy analysis of the AVP platform, leaving critical questions about its privacy protections unanswered.

Goals and challenges. This research aims to make a bold attempt to bridge the critical research gap: *detecting privacy non-compliance in commercial-off-the-shelf (COTS) AVP applications through network traffic analysis.* However, constructing an effective dynamic analysis framework faces three key technical challenges.

- **C1:** AVP's closed system architecture prevents automated testing. Unlike established platforms such as Android that provide debugging interfaces and root access capabilities, AVP offers no programmatic testing interfaces.
- **C2:** AVP's spatial user interface (UI) design creates fundamental challenges for UI exploration. The platform's 3D interaction model requires handling complex spatial relationships, motion tracking, and view management considering different head orientation of physical position, which creates an expansive state space that necessitates efficient exploration methods.
- **C3:** Detecting privacy violations in AVP network traffic requires a taxonomy that links low-level identifiers observed in packets to the high-level data categories used in privacy disclosures, but existing frameworks [17, 18] are not sufficient to capture AVP-specific data types or maps traffic to Apple's multi-layered documentation system of policies, labels, and manifests.

AVP-Inspect. To tackle the challenges and bridge the gap, we propose AVP-Inspect, a novel testing framework for AVP applications. Our key insight is that effective AVP application testing requires coordinated control of both cyber (software) and physical (hardware) components. We term this approach **Coordinated Cyber-Physical Testing**, which enables comprehensive application analysis by bridging the gap between software interfaces (cyber-space) and hardware interactions (physical-space). To tackle C1, we build a customized hardware automation system that uses Bluetooth Human Interface Device (HID) to generate user inputs in AVP, and implements a digital-to-physical pipeline for feedback-driven control. To tackle C2, we develop a UI exploration engine for exploring the 3D UI states, which incorporates specialized UI recognition and state pruning approaches. To tackle C3, we construct a systematic privacy analysis framework that captures privacy violations from network traffic based on a unified AVP privacy taxonomy.

We implemented a prototype of AVP-Inspect, and we evaluated it using a manually labeled ground-truth network traffic dataset consisting of 50 AVP applications. AVP-Inspect can cover 151 out of 157 privacy violations found via manual exploration within 20 mins of auto-exploration. AVP-Inspect exploration can yield 3.15× of traffic compared to the idle baseline. In addition, we performed a large-scale analysis on 324 COTS AVP applications, and we used AVP-Inspect to test each of them for 20 minutes and collect network traffic to study their privacy compliance.

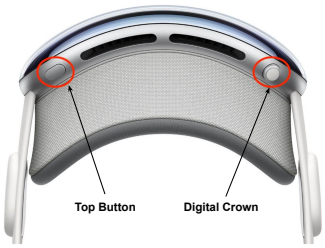


Fig. 1: Apple Vision Pro

Findings. Our large-scale analysis (§6) reveals widespread non-compliance with privacy disclosure requirements in the AVP app ecosystem. 188 of apps with observable network activity had one or more disclosure violations, and more than 60% of the network traffic flows are not properly disclosed. The most common deficiencies stem from omissions and inaccuracies in privacy policies (§6.1) and labels (§6.2). Similarly, privacy manifests, when present, are often incomplete or misrepresent actual data flows (§6.3). A majority of non-disclosures are due to the use of SDKs and the inherent descriptions (§6.4), which are often inadequate. We further observe significant discrepancies between declared and actual purposes of data collection (§6.5), exacerbated by ambiguities in Apple's data taxonomy and insufficient tooling for SDK attribution, ultimately leading to developer confusion and inconsistent compliance (§6.6).

Contributions. We make the following contributions:

- We introduce AVP-Inspect, the *first* end-to-end automatic testing tool for detecting privacy non-compliance of COTS AVP applications. It consists of a customized AVP device control system, a UI explorer, and a privacy violation detector.
- We evaluate AVP-Inspect through a 50-app ground-truth dataset, and show that AVP-Inspect is effective.
- We use AVP-Inspect to perform a large-scale analysis on 324 AVP applications, with each app tested for 20 minutes, which reveal that widespread non-compliance issue.
- We report our findings to the related parties, and we offer recommendations for addressing the issues.

Responsible disclosure. We have reported all findings (privacy violations) to Apple and related app developers. Their responses are discussed in §6.6.

2 Background

2.1 Apple Vision Pro

The Apple Vision Pro (AVP) [6] represents a pivotal step in the evolution of spatial computing. Introduced in 2024 with a price tag of 3,499 USD, Vision Pro integrates dual high-performance chips to support real-time sensor fusion, low-latency environment mapping, and high-fidelity mixed-reality rendering. It runs a specialized Operating System called visionOS which was designed for XR devices. Different from the popular Meta Quest series, AVP's control architecture relies on multimodal input combining precise eye tracking, hand gestures, and voice commands, which allows natural, controller-free interaction. Two hardware interfaces complement this input model (shown in Fig. 1): the Digital Crown (right), which

adjusts immersion levels and system view, and a Top button (left) for power, capture, and system resets.

2.2 Privacy Documents in Apple Vision Pro

Privacy policy. A privacy policy is a legal document provided by app developers that outlines how an app collects, uses, and shares user data, as regulated by laws such as GDPR [15] and CCPA [16]. Each privacy statement in the privacy policy can be modeled as a tuple `(entity, action, data)`, describing what data is collected (or explicitly not collected) by whom [22–24]. The *entity* is typically categorized as a first party (the app developer) or a third party (e.g., advertising or analytics SDKs). Without proper disclosure of these data behaviors in the privacy policy, the app poses privacy risks to end users and falls out of compliance with privacy laws.

Privacy label. To improve transparency and user awareness, Apple requires developers to complete privacy labels for all Vision Pro apps submitted to the App Store [25]. These labels provide standardized, easily readable summaries of the types of data an app may collect and how that data may be used or shared [26]. The privacy disclosure of the privacy label follows a four-layer taxonomy: data usage, purpose, data type and data item. A key distinction of Vision Pro privacy labels is the inclusion of unique data categories that reflect the device's advanced spatial and biometric sensing capabilities. Unlike mobile apps, Vision Pro applications may collect detailed environmental and interaction data, such as room mapping, scene mesh information, and user biometric signals like hand and eye movement data [26]. These expanded categories go beyond traditional location or usage data, underscoring the need for greater transparency in how XR technologies collect and process sensitive information, a focus that our study directly addresses.

Privacy manifest. The privacy manifest is a machine-readable declaration (typically named `PrivacyInfo.xcprivacy`) that developers are required to include with the app binary when submitting to Apple's App Store, as mandated since May 2024 [27]. Unlike privacy policies and privacy labels (which are written for end users), the privacy manifest primarily discloses data collection behaviors to the App Store and automated compliance tools. Its main purpose is to facilitate privacy compliance checks during the app review process. An inaccurate privacy manifest can mislead Apple's review process, resulting in incorrect verification of privacy disclosures and inconsistencies with the information reported in privacy labels. Therefore, our study also evaluates non-compliance in privacy manifests.

3 Overview

Goal. We aim to systematically evaluate privacy compliance in AVP applications through a dynamic analysis pipeline. Our objective is to develop an automated testing framework that triggers and captures network communications from commercial off-the-shelf (COTS) AVP applications *without requiring root access or application source code*. The framework analyzes privacy violations by identifying discrepancies between observed network traffic and the privacy declarations made by applications. We define a privacy compliance violation as any instance where privacy-sensitive data is transmitted to external parties without proper disclosure in the application's privacy documentation. Our analysis considers three key privacy documentation sources associated with AVP applications: privacy policies, privacy labels, and privacy manifests (§2.2).

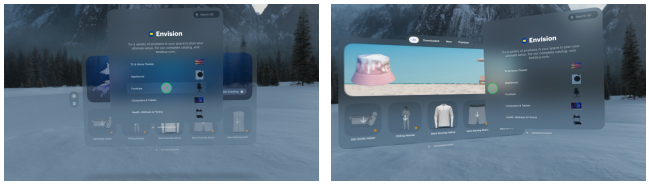


Fig. 2: Same UI with different depth and orientation.

3.1 Challenges

We face three main challenges when attempting to automatically test AVP applications to capture network traffic.

Challenge 1: Lack of programmatic interfaces for input simulation and device control. The first challenge stems from AVP's restrictive system architecture that prevents programmatic input simulation, which is a fundamental requirement for automated testing. Unlike iOS devices where jailbreaking [19] enables root access, AVP follows a strictly closed system model similar to other commercial XR platforms. This architectural choice precludes the use of established iOS testing frameworks like Macaca [28] or XCUITest [29] that require root access. Furthermore, while Android devices support debugging interfaces like ADB [30] even without root access, AVP provides no equivalent programmatic control interface. The platform also lacks standard input/output ports, preventing direct device control through conventional peripherals such as keyboard and mouse. These limitations create significant barriers for implementing automated testing procedures.

Challenge 2: Difficulty in exploring 3D UIs. Different from Android-based applications, AVP applications do not have XML-like UI documents. In addition, AVP applications present unique challenges for automated UI exploration due to their three-dimensional nature. Unlike traditional 2D interfaces where UI states are deterministic, 3D interfaces in AVP exhibit state variability based on user viewpoint parameters such as head orientation and physical position. This variability complicates state identification and deduplication since identical functional states may appear visually different (Fig. 2). This exponential growth in the state space due to duplicated states makes exhaustive exploration computationally intractable without intelligent pruning strategies.

Challenge 3: Insufficient privacy taxonomy for AVP traffic. Detecting privacy violations in AVP traffic requires a comprehensive taxonomy that maps lower-level terms (e.g., *"hand dimensions"*) in privacy policy to upper-level privacy data (e.g., *"vr movement"*) in privacy documentation declarations. While prior work has developed privacy taxonomies for other XR platforms (*e.g.*, VPVet [18] and OVRSeen [17]), these frameworks do not capture AVP-specific data categories or the relationships between network traffic and AVP's multi-layered privacy documentation system comprising policies, labels, and manifests due to ambiguity in those Apple specific privacy documentations.

3.2 Overview of AVP-Inspect

To systematically analyze privacy compliance in AVP applications via examining network traffic, we propose AVP-Inspect, a novel testing framework that addresses the aforementioned challenges. Our key insight is that effective AVP application testing requires coordinated control of both cyber (software) and physical (hardware) components. We term this approach **Coordinated Cyber-Physical Testing**, which enables comprehensive automatic testing by bridging the gap between software interfaces (cyber-space) and hardware interactions (physical-space). Based on this insight, we design AVP-Inspect with three integrated components: `Controller` for automated device control, `Explorer` for AVP UI exploration, and `Detector` for privacy violation detection. Together, these components form an end-to-end system that automates privacy compliance testing of AVP applications.

`Controller`: Hardware-based input simulation (addressing Challenge 1). We develop a hardware-based automation system that circumvents AVP's restrictions through a combination of techniques. At its core, the system uses a programmable Bluetooth Human Interface Device (HID) to generate virtual cursor inputs in AVP. We augment this with physical button control mechanisms. This comprehensive approach enables programmatic control of the AVP device without requiring system modifications or root access.

`Explorer`: UI exploration for AVP applications (addressing Challenge 2). We develop a specialized UI exploration engine that efficiently handles AVP's 3D UIs. Our approach first deploys screen mirroring for visual feedback processing, then projects 3D scenes onto 2D planes for simplified analysis. In addition, we design a precise cursor movement algorithm to control `Controller` to reach our target coordinates based on screenshots. We employ machine learning-based UI recognition to identify equivalent interface states across different exploration runs, accounting for visual variations caused by 3D perspective changes. We construct a state transition graph based on interactive UI elements, enabling efficient pruning of redundant states.

`Detector`: Privacy violation detection from network traffic (addressing Challenge 3). We develop a systematic privacy-violation detection module that identifies six types of privacy non-compliance by cross-checking data terms in network traffic against three key privacy documents. To achieve this, we extend prior taxonomies with AVP-specific privacy categories and data types derived through automated keyword extraction from our evaluation dataset. Our framework establishes unified taxonomies that align network traffic patterns with privacy documentation sources including manifests, labels, and policies. Using these aligned taxonomies, we implement a violation detection model that systematically identifies discrepancies between observed runtime network behavior and declared privacy practices.

4 Design of AVP-Inspect

Fig. 3 illustrates the workflow of AVP-Inspect. During the setup process, AVP-Inspect utilizes screen mirroring to replicate the screen of the AVP device onto a Macbook. The Macbook periodically captures screenshots, which are sent to `Explorer` for analysis. Given a list of applications, `Explorer` will first instruct `Controller` to open one target application, then start the exploration through UI

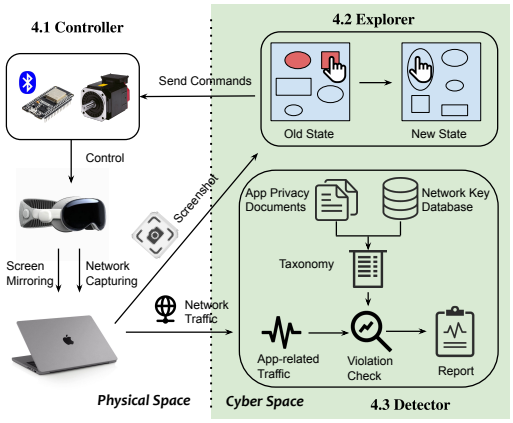


Fig. 3: Overview of AVP-Inspect.

analysis. During the exploration, `Explorer` will communicate with `Controller` frequently to understand the UI contents to decide the next steps. During the exploration, the network traffic will be captured and sent to `Detector`, along with the application's privacy documents, for checking privacy violations. In the end, a report will be created for the target application, detailing the traffic that includes privacy violations and specifying the type of data involved in these violations.

4.1 `Controller`: Simulating User Interactions

Although Apple provides the AVP developer strap [31], its functionality is limited to screen capturing, firmware downgrading, and fast file transfer, and it lacks the ability for programmatic input. In this subsection, we detail our design to overcome the lack of programming interfaces for input simulation. In particular, we aim to tackle two important problems: i) simulating user inputs (e.g., move cursors in the XR scene, touch virtual objects, etc); ii) pressing physical buttons on AVP device for XR-specific control that cannot be simulated.

i) Simulating User Inputs. Emulating an input device (*e.g.*, mouse) purely in software is not feasible on AVP due to several reasons. First, AVP does not expose synthetic input APIs for third-party to call. Moreover, AVP's Human Interface Device (HID) stacks (including both USB and Bluetooth devices) are handled at the system level: only external accessories that i) present as standards-compliant HID devices (e.g., BLE HOGP or Bluetooth Classic HID) and ii) complete secure pairing/bonding are accepted. As a result, software-based "virtual" HID devices created by third-party applications are not recognized by AVP, and cannot be paired or used as inputs.

As software-based simulation cannot work, we resort to hardware-based simulation. We first attempted to use laptops (e.g., Macbook) to simulate as HIDs. However, general-purpose computers typically operate as HID hosts (Central/BR/EDR host) and do not present a valid HID peripheral without dedicated firmware. Next, we have tried to use Bluetooth mouses such as Logitech mouses. While such devices can indeed pair with AVP, they do not provide programming interfaces for automatic control, as they need to be manually operated. To tackle this issue, we design a customized HID using an ESP32 device that presents a compliant Bluetooth HID keyboard

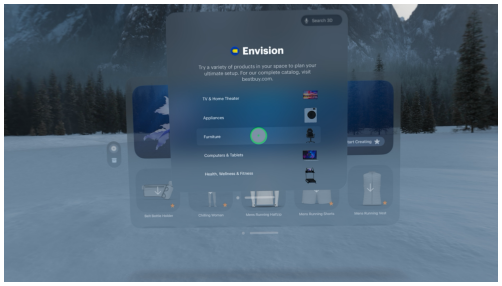


Fig. 4: Overlapping UIs from multiple applications.

and mouse with bonding persistence, following the Apple accessory guidelines [32]. The ESP32 device will listen on the commands from serial port, then emulated mouse and keyboard control event (mouse movement, click, keystroke, etc) via Bluetooth.

ii) Pressing Physical Buttons. With the simulated user inputs, we can already have the keyboard/mouse input to the AVP. However, AVP still limit the ability of external keyboard/mouse input; there are certain scenarios where physical pressing the hardware buttons is necessary. For example, in AVP, when the user's head moves, the view will not be automatically centered. The user needs to press and hold the Digital Crown button (Fig. 1) to center the view. This is very important, as the UI difference will affect the exploration in `Explorer`. Moreover, AVP does not provide simple ways for killing an application. This is also critical for us, as UIs of different applications may overlap and confuse `Explorer`. One example is shown in Fig. 4. To do so, it requires the user to press Digital Crown and Top buttons together to call out the *Force Quit* window (similar to Windows Task Manager) [33], then choose the app to kill. Therefore, pressing the physical buttons is required for automatically exploring AVP applications. To achieve the goal, we need to be able to apply proper forces to press the physical buttons as if the user is pressing them. To do so, we use servo motors [34] to provide the forces. To make sure that the servo motors can be programatically controlled, we integrate it with the user input simulation system in §4.1. Therefore, we can directly control the motors for pressing the physical buttons through the same ESP32 interface.

4.2 `Explorer`: Exploring AVP UI States

To perform exploration of AVP UIs, `Explorer` is designed with three major sub-components: i) *screen capturer* for taking screenshots of AVP UI; ii) *virtual cursor handler* for handling the movements of virtual cursor using `Controller`; iii) *state explorer* for exploring the UI states using the virtual cursor controller.

i) Screen Capturer. The first step is to capture the screenshot from AVP. While the AVP device provides the screenshot functionality, it is inconvenient to capture the screenshot programatically and send the image out for processing. To automate the screen capturing process, we use the AVP's built-in AirPlay functionality to mirror the AVP screen to a MacBook Pro, then perform recording and screenshot on the MacBook Pro. After capturing the screensots, we will perform analysis on the screenshots, which will inform us next steps (e.g., move cursor, click button, etc).

ii) Virtual Cursor Handler. We design a calibrated cursor positioning and moving system that maps 2D screen coordinates (from the mirrored screenshots) to 3D space interactions (in the AVP device screen) to precisely control the cursor in the 3D space in AVP apps.

Locating the cursor from 2D screenshots. Unlike traditional smartphone UI exploration systems that can query cursor positions directly, AVP does not provide ground truth cursor coordinates. We must therefore locate the virtual cursor from screenshots. The default cursor appears as a small white dot that is difficult to distinguish from application UI elements and backgrounds. To address this challenge, we modify the cursor appearance in system settings by adding an outer circle and changing its color to green. We then apply two computer vision techniques to detect the cursor: (1) color filtering to isolate green pixels, and (2) Hough Circle Transform [35] to identify the coordinates of the green circle with known radius.

Controlling the cursor movement in the 3D space. Since AVP does not provide absolute position control for cursor movement, we design an algorithm to move the virtual cursor to target positions. The key challenge is that 2D screenshots do not accurately represent depth in 3D space. Objects at different depths appear at similar 2D coordinates but require different movement distances to reach. Therefore, we cannot directly map 2D screen distances to cursor movements in 3D space. We address this challenge using a binary search approach. Given the current cursor position (x, y) and target position (x', y') in the screenshot, we move along each axis independently: first along the X-axis to reach (x', y), then along the Y-axis to reach (x', y'). To move from x to x', we perform the following steps: 1) Move the cursor toward x' by λ units (a large initial step size) in `Controller`, and repeat until the cursor moves past x'. 2) Let x_1 denote the new cursor position. If $|x_1 - x'| \leq \delta$ (where δ is a small threshold), the cursor has reached the target. Otherwise, proceed to step 3. 3) Set $\lambda = \lambda/2$ and $x = x_1$, then return to step 1.

iii) State Explorer. To systematically explore the UI states in AVP, we design a state explorer that constructs the UI state graph of the target application on-the-fly, as there is no existing interface to obtain such information.

Definitions. To comprehensively explore the AVP application, we use a *state graph* to represent the application's state transition. The state graph is a directed graph, where the nodes are the UI states of the application, and the edges are the transitions between states.

- *Node (State).* The node (*i.e.*, state) of the state graph is defined as the current UI in the 2D screenshot, which consists of buttons, images, texts, *etc.* To obtain a comprehensive list of objects in the 2D screenshot to represent the state, we use an object detection model pre-trained for UI elements to precess the 2D screenshot. The model will output the detected UI elements with detailed information, such as their coordinates and the text contents, which is recorded for denoting the state. One example is shown in Fig. 5.
- *Edge (State transition).* An edge in the state graph represents a transition between two states, triggered by a UI element. In our implementation, edges correspond to clickable UI elements (buttons) that cause state transitions. An edge can be a forward edge, representing entering a new state, or a backward edge, representing a return to a previous state.

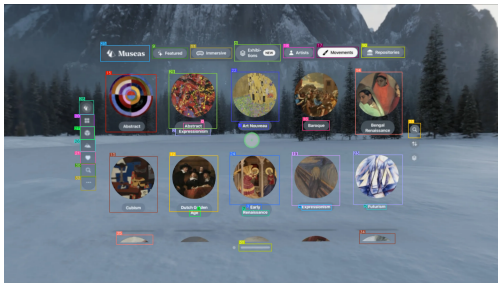


Fig. 5: UI element detection.

Exploration algorithm. algorithm 1 (Appendix D) presents our state exploration algorithm. The algorithm maintains a state graph G that records all discovered states (nodes) and their transitions (edges). Starting from the initial state of the application (`Main()`), the algorithm performs a depth-first search (DFS) to systematically explore all reachable states. For each state, the algorithm identifies all clickable UI elements (buttons) and attempts to click each unexplored button. After each click, it captures a screenshot and extracts UI elements to determine the current state (`CheckCurrentState()`). If the current state is new, it is added to the state graph along with the transition edge. If the state has been visited before, the algorithm resumes exploration from that state by clicking its remaining unexplored buttons (`ResumeExploreState()`). When all buttons in a state have been explored, the algorithm attempts to return to the previous state by clicking the back button (`TryClickBackButton()`). If no unexplored buttons remain in the current exploration path, the algorithm selects another state with unexplored buttons from the state graph, jumps to that state by restarting the application and replaying the path from the initial state (`JumpToCurrentState()`), and continues exploration (`ResumeExploreState()`). This process continues until all states have been fully explored.

Navigating back to previous states. Our DFS-based exploration requires a mechanism to return to previously visited states. In AVP, most applications provide a *back* button at the top left corner of the window, similar to Android. Clicking this button returns the application to the previous state. However, unlike Android where the back button is system-level, AVP applications do not always include this button. Specifically, pop-up windows often lack a back button. In such cases, after exploring all states within the pop-up, we must close the window to return to the previous state. We achieve this by using a cursor movement pattern to click the *Close* button [36] located in the bottom window bar.

Returning to the initial state. The exploration algorithm requires a method to return to the initial application state when jumping between unexplored states (`JumpToCurrentState()` of algorithm 1 in Appendix D). In AVP, simply closing all windows does not reset the application. When reopened, the application resumes at its last state rather than the initial state. To reset the application, we use the *Force Quit* feature. We first trigger *Force Quit* through `Controller`, then execute a cursor movement pattern to terminate the application, ensuring it restarts from the initial state.

Same-state pruning. Unlike 2D scenarios (e.g., Android) where identical states produce identical UI layouts, same AVP states can have small differences in UI layouts with user viewpoint parameters such as head orientation and physical position. To handle these minor UI variations, we compute state similarity using the (position, size) of the detected UI elements. If the similarity exceeds a threshold, we consider the two states identical. During exploration, we maintain all discovered states in the state graph. When transitioning from state A to state B, we check whether state B matches any adjacent explored state of A. If a match is found, we return to state A. Otherwise, we add state B as a new adjacent state of A and continue exploration. This pruning strategy significantly reduces the number of states explored.

4.3 `Detector`: Detecting Privacy Violations

`Detector` analyzes network traffic captured during app exploration to detect privacy violations by comparing transmitted data against privacy disclosures. It consists of three steps: (1) building a unified privacy taxonomy that maps data types from privacy policies, labels, manifests, and network traffic to a common representation; (2) extracting privacy-related keys from network traffic and mapping them to taxonomy nodes; and (3) identifying violations by comparing detected data types against privacy disclosures.

Privacy Taxonomy. To detect privacy violations, we build a unified taxonomy that maps data types from multiple sources—privacy policies, privacy labels, privacy manifests, and network traffic—to a common representation. We construct our taxonomy by extending VPVet's taxonomy [18] with AVP-specific data types through a four-step process. First, we extract data types from privacy documents. We use the `PrivBERT` model to extract privacy statement triplets ⟨`entity, action, data type`⟩ from privacy policies, directly use Apple's predefined data type categories [26] from privacy labels, and parse the `NSPrivacyCollectedDataTypes` field from privacy manifests. Second, we map all extracted data phrases to VPVet's existing taxonomy using synonym matching and collect phrases that fail to match as *unmapped phrases*, which represent AVP-specific data types not covered by traditional mobile privacy taxonomies. Third, we discover new data types from unmapped phrases using the embedding method in VPVet (see algorithm 2 in Appendix D): we manually select a representative phrase as a new data type candidate, compute similarity between the candidate and remaining phrases, group similar phrases (similarity ≥ 0.8) under the same data type, and repeat until no new types can be identified. Fourth, we manually insert newly discovered data types into appropriate positions in the taxonomy hierarchy (e.g., biometric-related types under the `PII` branch), with the grouped phrases from the previous step becoming synonyms for each new data type to enable phrase-to-term mapping.

Concretely, we pair each lower-level data type with its upper-level node by annotating is-a (hypernym) relations: a lower-level type B is linked to an upper-level node A when *B is an A*. For example, the traffic-side type `user id` is paired with the Apple privacy-label category *User ID*. Two annotators performed this labelling independently and a third adjudicated the disagreements. Starting from VPVet's 106 data type (after filtering 1 Android-specific data type), we extend our taxonomy to include new data type. We add 12 data types from privacy policies and 35 data types from privacy labels. Finally, we have a total of 153 data type in our taxonomy.

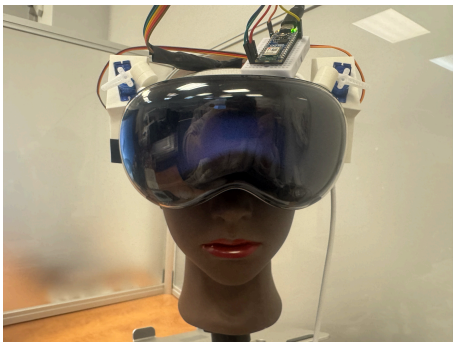


Fig. 6: Implementation of `Controller`.

Network Traffic Key Extraction. To identify privacy-related data in network traffic, we extract all key-value pairs from HTTP requests by parsing three sources: (1) HTTP headers, (2) URL query parameters, and (3) request bodies (including nested JSON/XML structures). For nested data structures, we recursively flatten them to obtain all key-value pairs. We then map extracted keys to taxonomy nodes using three matching strategies. First, *synonym matching* maintains a synonym list for each taxonomy node and maps keys that exactly match any synonym (e.g., `"user_id"`, `"uid"`, and `"userId"` all map to `user identifier`). Second, *key pattern matching* uses regular expressions to match common naming patterns (e.g., keys ending with `"_id"` or `"_token"` map to identifier-related nodes). Third, *value pattern matching* uses regular expressions to match value formats for certain data types (e.g., values matching `xxx.xxx.xxx.xxx` map to `IP address`).

Privacy Violation Detection. We compare data types found in network traffic against three privacy disclosure sources to identify violations. A unique violation is defined as a distinct `(key, category)` pair that appears in network traffic but is not covered by the app's privacy documentation: 1) a *policy violation* occurs when the data type is not disclosed in the privacy policy; 2) a *label violation* occurs when it is not declared in the App Store privacy label; and 3) a *manifest violation* occurs when it is not listed in the privacy manifest. For example, if an app transmits `x-device-id` (mapped to "Identifiers") but does not declare "Identifiers" in its privacy documents, this constitutes a violation.

More details about the three matching mechanisms, first-party versus third-party classficiation, and why neither the policy nor the label can serve as ground truth are presented in Appendix L.

5 Evaluation

Implementation. We implement a prototype of AVP-Inspect, which consists of 286 lines of ESP32 code and about 3,000 lines of Python code. Fig. 6 shows our implementation of `Controller`. We stabilize the AVP device using a makeup training head mounted on a clamp stand during experiments. We implement the customized HID using an Arduino ESP32-S3 microcontroller with built-in Bluetooth Low Energy support. To construct UI states from screenshots, we deploy a pretrained OmniParserV2 model [37]. We capture network traffic from the AVP device using a Man-in-the-Middle (MITM) approach. More details are presented in Appendix K.

Experimental Setup. In our experiment, the AVP device is running visionOS 2.1. The AVP screen is mirrored to a MacBook M2 Pro with 32GB RAM running macOS Sequoia. The same MacBook runs `mitmproxy` to capture network traffic from the AVP. We run the `OmniParserV2` model on a Lambda Vector Pro server with Threadripper PRO 7985WX CPUs (64 cores), two NVIDIA RTX A6000 GPUs (48GB each), 256GB RAM, and 10TB NVMe SSD. Screenshots captured on the MacBook are sent to the Lambda server for processing. Both machines are connected to the same local network to minimize communication latency.

Evaluation Process. We created an automated script to download and install target apps from the AVP App Store using AVP-Inspect based on App IDs from our dataset. For each app in the list, we install it on AVP and open it by searching for the application name in the `Spotlight` search. We then use the Digital Crown button to center the view and begin testing for 20 minutes. After testing completes, we force quit the application and proceed to the next one. For each app, we used screen recording to record all explorations when AVP-Inspect is exploring the app. We also record the network traffic triggered during exploration. We use the timestamps of starting/ending each app to segment the network traffic for each app.

5.1 Datasets

We constructed our AVP app dataset by crawling the 3rd party website [38] (the official VisionPro App Store does not provide app list) that list all available AVP apps, which contained 671 apps, of which 460 were free. For the free apps, we excluded 86 apps that were not applicable to our analysis: 16 Apple system apps (e.g., Safari) that are out of scope, 4 Microsoft apps (e.g., Word) whose IPA files could not be acquired through standard methods, 20 Apple Arcade games that require an active subscription despite being listed as free, and 46 apps requiring visionOS versions newer than our test device (visionOS 2.1), including 7 apps requiring visionOS 2.2-2.6 and 39 apps requiring visionOS 26.x. After applying these filters, our final dataset comprises 374 AVP apps that we successfully tested and analyzed. We used the third-party directory only to enumerate which titles are available for visionOS, as the official App Store exposes no such listing; every app we analyze was downloaded from the official App Store through `ipatool`, and privacy labels were collected from App Store. The number of apps per category is proportional to that category's size, and within each category the apps are selected at random.

Ground Truth Dataset. To evaluate AVP-Inspect's ability to detect privacy violations, we construct a ground truth dataset through manual exploration of AVP apps in our dataset. We selected 50 apps spanning top-10 popular categories. As the number of available apps varies across categories, we selected apps proportionally based on the size of each top-10 category, and each category is represented by at least two apps in our dataset. During manual exploration, we trigger all possible interactive UI elements (e.g., buttons and objects) in the app to cover as many states and network requests as possible. The exploration stops when no interactive elements remain. In cases where there are multiple pages of the same type of items (e.g., a BestBuy store page), we only explore one page, as interacting with different items typically generates the same type of network traffic. We manually analyze the network traffic to identify

Table 1: Network traffic and privacy violation detection results for 31 apps. GT: Number of unique network requests found during manual exploration; Auto: Number of unique network requests found by AVP-Inspect; GT Vio: Number of unique violations found during manual exploration; Auto Vio: Number of violations found by AVP-Inspect. Some violation numbers exceed network request counts because a single request may contain multiple violations.

Category	App	GT	Auto	GT Vio	Auto Vio
Productivity	Rad Timer	2	1	0	0
	FreelanceKit	4	4	0	0
	Cardhop	5	1	13	13
	Focus - Timer	28	12	0	0
	FocusBeats	4	3	0	0
	Flippy	7	3	12	11
Utilities	Craft	67	10	0	0
	Broadcasts	312	238	0	0
	Qlone	2	1	0	0
	WWidgets	17	6	4	2
Games	Retrogram	26	9	21	21
	Things	78	40	0	0
Entertainment	Callsheet	413	374	0	0
	Turn Off the Lights	21	5	0	0
	Wet Your Beak	45	24	2	2
	Paramount+	150	60	8	8
Lifestyle & Health	alo Sanctuary	130	102	4	3
	Decathlon USA	21	8	4	4
	Mindr	5	1	0	0
	Bible	15	15	14	13
	Best Buy	44	44	12	12
Education	Chemistry	8	8	9	9
	Drawing Desk	7	7	0	0
	Foxar	81	81	4	4
	Inviewer	9	3	24	23
Creative	Kineo	1	1	5	5
	Spatial Station	55	25	0	0
	Theater	340	176	0	0
	Twin	14	5	8	8
	Cubes	12	4	13	13
Social	OverSoul	14	5	0	0
Total	**31**	**1937**	**1276**	**157**	**151**

unique network requests, extract keys, and check for violations. We consider network requests that retrieve or send the data type keys (regardless of the value) as identical. If any network key maps to our privacy taxonomy but not declared in the privacy documentation, we mark it as a potential violation. Manual exploration and analysis required approximately 48 person-hours for the 50 apps. This dataset is used for our evaluation (§5).

Large-scale AVP App Dataset. We use the remaining 324 apps in our dataset for large-scale analysis. For each app, we extracted metadata and privacy information (privacy policy URLs and privacy labels) from the AVP App Store. We also downloaded the IPA files using `ipatool` and extracted the privacy manifests. We use this dataset for large-scale analysis (§6) and to build our network key dataset, which we map to our privacy taxonomy as described in §4.3.

5.2 Privacy Violation Detection Results

We evaluate AVP-Inspect's ability to detect privacy violations by comparing violations discovered through automated exploration against those found in manual exploration. We record all explorations and network traffic when AVP-Inspect explores the 50 ground-truth apps. 19 of the 50 evaluation apps produce no captured network traffic during either manual or automated exploration. Table 1 shows the network traffic and violation detection results for the remaining 31 apps. Among the 31 apps with captured traffic (8 categories), 15 contain at least one privacy violation from our manual analysis. AVP-Inspect achieves an overall violation coverage of 96.2%, detecting 151 out of 157 ground truth violations across 31 apps with captured traffic, despite only exploring for 20 minutes.

The lowest-coverage app in our evaluation is *WWidgets* (50.0%, 2 of 4), where two violations occur in widget-configuration flows that require additional navigation steps not reached within the 20-minute window. These results show that AVP-Inspect is effective for privacy compliance auditing, as it detects the majority of privacy violations within the first 20 minutes of exploration. We expect that with more time, AVP-Inspect can cover more violations. We present the runtime performance of AVP-Inspect in Appendix F. We also examined the raw captures to quantify interception failures due to MitM settings. We observed no certificate rejections, no undecryptable payloads, and no app exhibiting the behavior characteristic of certificate pinning. Appendix N reports the details.

5.3 Effectiveness of Auto-exploration

A natural question is whether the network requests captured during AVP-Inspect's automated exploration genuinely reflect interaction-driven behavior, rather than background traffic the app would emit even when left untouched (e.g., periodic analytics requests). To answer this, we compare per-app traffic under two conditions on our 50-app ground-truth set: (i) *Auto*, with AVP-Inspect performing 20-minute UI exploration (from Table 1), and (ii) *Idle*, where the same app is launched and left foregrounded but untouched for the same 20-minute duration. Both runs have a 20-minute time limit from app launch. Note that the Auto will end early if all elements are exhausted. To account for early termination we restrict each app's *Auto* run timeframe to its *active interval* (first click ~ last click + 3s) and trim the *Idle* run to a window of equal length. All 50 evaluation apps have a (Auto, Idle) pair.

Manual and automated exploration follow the same protocol and detection logic, systematically triggering every interactive element until none remain, interacting with one representative item per group of same-type elements, and applying the same key-extraction and taxonomy-mapping pipeline. They differ only in how the UI is driven: a human interprets the interface directly, whereas automation relies on non-deterministic OmniParser detection, so the two traverse elements in a different order and depth and do not reach identical states or paths. Manual exploration therefore captures the upper bound of an app's network flows, which is why GT request counts exceed Auto in Table 1.

Results. Of the 50 apps, 26 produce non-zero traffic in both conditions, 5 trigger requests only in *Auto*, and 19 produce no traffic in either condition. Aggregating across all 50 apps, AVP-Inspect's exploration triggers **1,276** unique requests versus **405** from a length-matched idle baseline, i.e., a **3.15**× increase. The average network request per minute is 3.51 (Auto) vs. 0.63 (Idle), indicating a **5.58**× boost. A detailed per-app comparison is shown in Table 3 (in Appendix F). Among the 26 apps with traffic in both conditions, 17

see strictly more traffic under *Auto* and 7 are tied (e.g., `Kineo`, 1 vs. 1). the largest gaps appear in interaction-heavy apps such as `Broadcasts` (238 vs. 1, 238×) and `alo Sanctuary` (102 vs. 1, 102×).

Overall, these results confirm that AVP-Inspect's exploration is effective in triggering new traffic that cannot be observed without interaction. We also present a more detailed study on 1) correlation between clicks and traffic, 2) runtime overhead breakdown of AVP-Inspect in Appendix F. We also studied the results of longer runtime (1 hr) in Appendix M.

A naive metric is the fraction of manually reached UI states that AVP-Inspect rediscovers. However, this is not suitable in our case. AVP provides no view hierarchy, so "state" is defined by our own perceptual dedup heuristic over detected UI elements (§4.2); there is no ground-truth state identity against which either side can be scored, and the same functional screen can be counted once or several times depending on head pose. Consistent with this, repeated runs of the same app produce noticeably different state counts. We therefore evaluate AVP-Inspect on the outcome that is well-defined and that downstream analysis actually consumes (the set of privacy violations recovered) for which manual analysis does provide a ground truth.

6 Large-scale Analysis

In this section, we present the first large-scale analysis of automated AVP app exploration in the wild to capture privacy violations in network traffic. AVP-Inspect explores 324 AVP apps in our large-scale dataset. Of these, 247 apps (76.2%) generate network traffic during our exploration window, resulting in a total of 25,901 network requests triggered across the experiment; the remaining 77 apps (23.8%) emit no traffic.

Violation Landscape. Our privacy compliance analysis shows that 188 of the 247 apps with captured traffic exhibit at least one violation, and 61.1% of observed privacy-related data flows occur without proper disclosure. The distribution of all violation traffic is shown in Fig. 7; productivity, entertainment, and utility apps are the top three non-compliant app categories. Across the three disclosure documents, privacy labels account for the largest share of violations: 1,785 label violations (658 contrary, 1,127 neglect) from the 324-app dataset. By comparison, we observe 1,529 policy violations (29 incorrect, 278 mismatched-entity, 1,222 neglect/omit) and 138 manifest violations (23 contrary, 115 neglect). Furthermore, the most frequently undisclosed data types are device information, identifiers, and usage information, and they are often transmitted to third-party advertising or analytics endpoints such as *Mixpanel*, *RevenueCat*, and *Amplitude* for user tracking, posing significant privacy risks to end users. Note that our result is a conservative **lower bound** of potential privacy non-compliance, due to the inherent limitations of dynamic analysis, where the UI exploration may not be exhaustive within a fixed timeframe.

6.1 Privacy Policy Non-Compliance

We collected privacy policies for 247 AVP apps with network traffic and successfully analyzed 212 apps. The remaining 35 apps were excluded due to anti-crawl mechanisms, unavailable web content, or non-English content. Following prior work [39–41], we employed the flow-to-policy consistency model [23] to identify three types of non-compliance (Fig. 15 in Appendix G):

• ***Omit Disclosure*** arises when data is collected, but there are no statements in the privacy policy that disclose or mention this collection. In our study, we identified 1,222 instances of Omitted Disclosure across 137 apps with policy violations. The most frequently omitted data types are *device identifiers*, *platform information*, *app version*, and *system version*. Notably, the vast majority of omitted disclosures involve data collected by third-party services, while only 9.1% involve first-party collection, suggesting that developers often fail to account for data collection by embedded SDKs.

• ***Incorrect Disclosure*** is present when data is indeed collected, but the privacy policy conversely states that such data collection does not occur (i.e., a negative sentiment sharing or collection statement). In our study, we identified 29 instances of Incorrect Disclosure across 12 apps with policy violations. The most commonly incorrectly disclosed data types are *usage time*, *usage info*, *session data*, and *geographical location*. The vast majority of incorrect disclosures originate from third-party services/SDKs, contradicting their "no data collection" claims.

• ***Mismatched Entity Disclosure*** arises when data is collected and sent to a specific recipient entity (often a third party, e.g., an advertiser), but the privacy policy attributes that data collection to a different entity (often the first party, i.e., the app developer). This mismatch matters because users may be willing to share data with the app developer but not with unknown or untrusted third parties, so misattributing the recipient can mislead users' privacy expectations and erode trust. In our study, we identified 278 instances of Mismatched Entity Disclosure across 81 apps, where the vast majority of violations involved data collected by third-party services without proper attribution. The most common undisclosed third-party collectors include RevenueCat (in-app purchase management), Unity Analytics, Google/Firebase services, and Mixpanel (analytics).

6.2 Privacy Label Non-Compliance

We categorize the discovered violations into two primary types based on the nature of the privacy label discrepancy (Fig. 8):

The data types in different figures are at different granularity. Fig. 7 gives an overview of all violation traffic, aggregated into the six top-level data type categories defined in the VPVet taxonomy. Fig. 8 details privacy-label violations over the top-30 data types. Fig. 9 and Fig. 10 analyze purpose non-compliance over the 15 top-level Apple privacy-label categories. Relatedly, *person name* is a fine-grained data type originating from the VPVet/PoliCheck dataset, whereas *Name* is a coarser-grained type defined in Apple's privacy-label taxonomy; they refer to the same data category at different levels of granularity.

• ***Neglect Disclosure***: Apps that collect data types without any corresponding privacy label declaration. These represent complete omissions where developers fail to disclose data collection practices entirely. In our study, Neglect Disclosure accounts for 63.1% (1,127 instances) of all privacy label violations, with the most commonly neglected data types being *diagnostic data*, *usage info*, and *device id*. See §N.1 for a representative Neglect Disclosure case study.

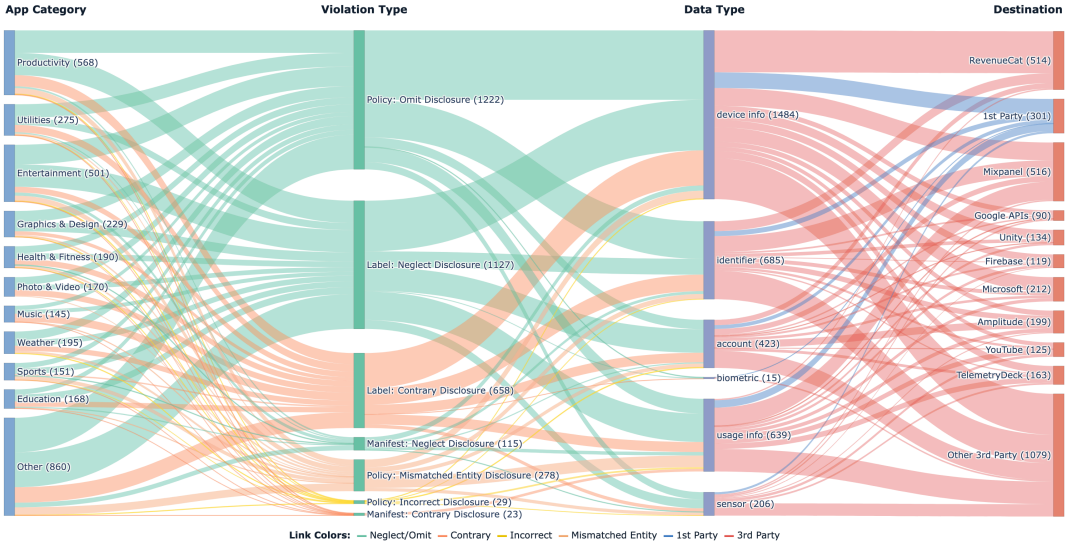


Fig. 7: Data flow of all violation traffic.

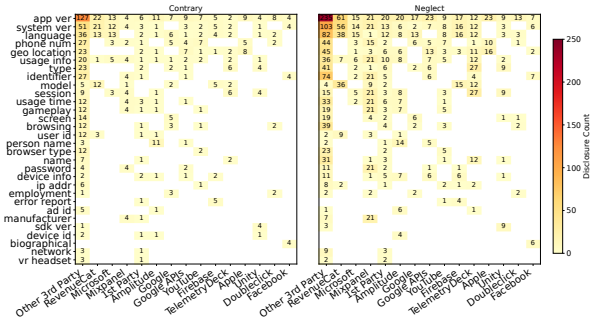


Fig. 8: Results: Privacy Label (Partial)

• ***Contrary Disclosure***: Apps that explicitly state they do not collect certain data types in their privacy labels while actually collecting that data during runtime. These violations represent direct contradictions between stated and actual privacy practices. In our study, Contrary Disclosure accounts for 36.9% (658 instances) of violations, affecting 64 apps with a total of 1,423 label-violating network requests. Notably, first-party apps exhibit significantly fewer Contrary Disclosure violations compared to third-party SDK-driven collection. This pattern suggests that app developers are generally aware of their own data collection practices, but may not fully understand or account for the data collected by integrated third-party services. See §N.1 for a representative Contrary Disclosure case study.

6.3 Privacy Manifest Non-Compliance

Apple introduced Privacy Manifests (`PrivacyInfo.xcprivacy`) in 2024, requiring developers to declare privacy-sensitive API usage, tracking domains, and collected data types directly within the app bundle [27]. Unlike privacy policies (free-form text) and privacy labels (App Store metadata), privacy manifests are embedded in the IPA file and follow a structured plist format, enabling automated compliance verification.

Dataset Coverage. Of the 324 AVP apps in our large-scale dataset, 84 (25.9%) include a Privacy Manifest. The declaration patterns cover four key manifest fields. While all 84 apps declare `NSPrivacyAccessedAPITypes` (required API usage reasons), only 30 (35.7%) declare `NSPrivacyCollectedDataTypes`, and a small minority list any `NSPrivacyTrackingDomains`. This suggests that most developers treat the manifest as an API-compliance checkbox rather than a comprehensive privacy declaration.

We identify manifest-specific violations in 41 of 84 apps (48.8%), with 138 total violation instances (23 Contrary, 115 Neglect). We categorize these violations using the same two-type framework applied to privacy labels (Fig. 16 in Appendix G):

• ***Contrary Disclosure (23 apps).*** Of the 84 apps with manifests, 23 explicitly set `NSPrivacyTracking` to `False` yet still transmit privacy data to known tracking domains (e.g., `mixpanel.com`, `amplitude.com`, `doubleclick.net`) during runtime, constituting a direct contradiction.

• ***Neglect Disclosure (18 apps).*** Neglect Disclosure manifests in two forms. First, among 7 apps that correctly set `NSPrivacyTracking` to `True`, 1 provide incomplete `NSPrivacyTrackingDomains` lists that omit observed tracking endpoints. Second, 17 apps declare `NSPrivacyCollectedDataTypes` but omit data types that are actually collected, with the most frequently missing types including identifiers and usage data collected by third-party SDKs.

6.4 Non-Compliance due to Third-party SDKs

Motivated by our observation in §6.1 and §6.2 that the majority of privacy violations in our dataset are associated with such third-party data flows, we focus on data transmitted via third-party SDKs.

Table 2: Apps and violations per category

Category	Apps	Violations
Productivity	26	568
Utilities	19	275
Games	2	60
Entertainment	30	501
Lifestyle	7	140
Health & Fitness	10	190
Education	10	168
Photo & Video	9	170
Graphics & Design	12	229
Social	6	56
Weather	8	195
Sports	5	151
Music	10	145
Finance	5	109
Reference	5	70
Puzzle	4	69
Other (13 categories)	20	356
All	**188**	**3,452**

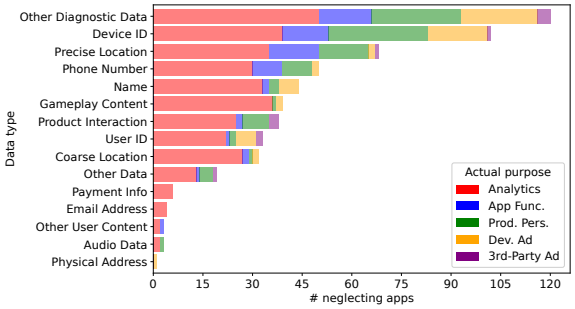


Fig. 9: Undisclosed purposes in privacy labels.

We categorize SDKs into five categories based on manual analysis: analytics, authentication, advertising, subscription, and others.

Figures 17, 18, 19 (Appendix H) shows that privacy policy, label and manifest omissions are not evenly distributed across Apple's data taxonomy. The SDK related violations are most prevalent in high-volume categories such as *Productivity* and *Entertainment*, followed by *Utilities*. In these categories, *Analytics* SDKs account for the most of the violations.

In contrast, certain categories exhibit a different dominant source. For example, in *Games*, violations are more heavily driven by *Advertising* SDKs. While analytics-related SDKs dominate at scale, the relative contribution of SDK types varies across categories, aligning with differences in app usage patterns and monetization strategies.

Table 2 reports how many apps underlie each category, since violation counts alone conflate category size with per-app severity. Productivity (26 apps) and Entertainment (30 apps) dominate in absolute terms because they are the largest categories.

6.5 Purpose Non-Compliance

Each app's privacy label specifies collected data types and their intended purposes. We assess compliance by comparing the privacy-label purpose for each data type against the inferred runtime purpose of actual data flows; any discrepancy constitutes a *purpose*

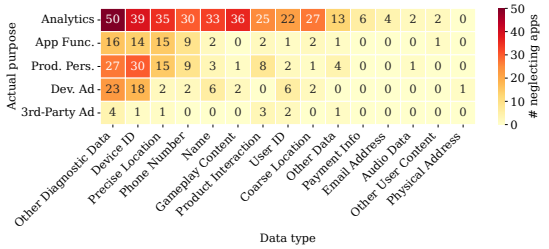


Fig. 10: Inadequate disclosures across different purposes.

neglect violation. To infer runtime purposes, we employ the purpose classifier from Lalaine [42], which predicts Apple's five privacy purposes (*App Functionality*, *Analytics*, *Product Personalization*, *Developer's Advertising or Marketing*, *Third-Party Advertising*) using app metadata and network-level features.

Fig. 9 shows that the most common omissions involve *Other Diagnostic Data* and *Device ID*, with further gaps in *Precise Location* and *Phone Number*. Across these prevalent types, *Analytics* emerges as the dominant inferred purpose, suggesting overlooked disclosures often stem from measurement and telemetry infrastructure rather than app-specific logic. Notably, data types such as *Device ID* and *Other Diagnostic Data* are linked not only to Analytics, but also to App Functionality, Product Personalization, and Developer Advertising. Regardless of data type, collection typically serves multiple purposes, especially *Analytics* and *Product Personalization*, with a substantive *Third-Party Advertising* component. Diagnostic data, while frequently omitted, is widely monetized by analytics and ad-tech SDKs. Fig. 10 further details violations by purpose and data type. The concentration is highest in *Analytics*. spanning both diagnostic and user-centric data: *Other Diagnostic Data* (50), *Device ID* (39), *Gameplay Content* (36), *Precise Location* (35), *Name* (33), *Phone Number* (30). Many data types, such as *Device ID*, support multiple purposes (e.g., Analytics, Product Personalization, Developer Advertising, and App Functionality), revealing that the problem lies not only in whether data types are disclosed, but also in whether their purpose is accurately represented.

Concretely, data that developers file under *Diagnostics* in our traffic includes Crashlytics installation identifiers and OS build versions (`x-crashlytics-installation-id`, `x-crashlytics-os-build-version`), the device model string `RealityDevice14,1` (sent to TelemetryDeck and Crashlytics), SDK and app versions (`sdk_ver=2022.3.47f1` to Unity; `$app_version_string` to Mixpanel), screen resolution, and network type (`networkType=wifi`). Data filed under *Usage Data* includes session identifiers (`$mp_session_id` to Mixpanel; `sessionID` to TelemetryDeck) and per-event timestamps. These examples illustrate that "diagnostics" and "usage" collection routinely carries stable identifiers suitable for cross-app linkage. More broadly, AVP-Inspect operationalizes several goals of the NIST Privacy Framework [43] for AVP apps: identifying actual data processing through runtime traffic, comparing observed flows with disclosed privacy practices, and helping developers govern third-party SDK behavior. It thus serves as a practical privacy-auditing tool that bridges abstract privacy-risk goals with concrete evidence of what data leaves AVP apps and whether those flows are properly disclosed.

6.6 Responses from Apple and Developers

Following responsible disclosure practices, we reported our findings to Apple and app developers. We summarize their responses as follows.

Apple's Responses. We reported our findings to Apple via the Feedback Assistant and App Store Review channels, providing affected apps, supporting evidence, and a summary of the most prevalent violations. After receiving our reports, Apple acknowledged, validated, and reproduced our findings.

Developer Responses. We searched for developer contact information from App Store information and developer websites, and were able to find the contact information for 119 apps out of 188 apps with violations. We reported our findings to them, and we have received 10 responses so far. We categorize the responses into four categories (some of them belong to multiple categories):

(i) Acknowledging disclosures. Two developers agreed to update or refine their privacy disclosures. For example, *Alpenglow Sunset Predictions* acknowledged that User ID transmission (via RevenueCat and Unleash SDKs) was previously underdisclosed, and revised the privacy label to declare User ID under "Analytics / Not Linked to You". Similarly, *Space Vision* stated that its existing "Crash Data – App Functionality" label covers Firebase Crashlytics, but committed to making the disclosure more precise in the next release.

(ii) Relying on SDK documentation or SDK-provided semantics. Four developers based their disclosures on SDK documentation or information provided by SDK vendors regarding data semantics. For instance, *1Blocker* cited RevenueCat's official privacy documentation to support its current label assignments, and *Alpenglow* argued that RevenueCat's use of Apple Search Ads attribution leverages Apple's AdServices framework rather than persistent device identifiers. These responses suggest that developers often defer to SDK vendors' data-handling descriptions, rather than directly verifying runtime data flows.

(iii) Arguing that existing labels already cover observed behavior. Five developers stated that their current privacy-label disclosures adequately cover the reported data flows. For example, *Space Vision* mapped Firebase Crashlytics to "Crash Data – App Functionality," and *Spatial Physics Playground* contended that its first-party requests were already labeled as "Diagnostic Data, Not Linked to Identity." Similarly, *Sequel Media Tracker* stated that its privacy label already includes purchases, location, identifiers, and usage data. These cases highlight the broad and sometimes ambiguous categorization permitted by Apple's privacy-label taxonomy.

(iv) Disputing data-type classification or reported flows. Four developers contested our traffic analysis or the classification of specific data types. For example, *Alpenglow* argued that an attribution token should not be classified as a persistent Device ID; *Cardhop* requested clarification of what "Browsing" represented. This indicates that the classification of data types is not always clear-cut, and there are cases where developers may have different interpretations of the data types.

In the disputed cases, developers often categorized specific data items into broader labels such as "Diagnostics" or "Usage Data," even when the transmitted fields more precisely matched Apple-defined categories such as identifiers or browsing activity. For example, one developer argued that a Device ID sent to Mixpanel was covered under "Usage Data," but Apple requires Device ID to be declared as such even when marked as not linked to identity. Our categorization follows Apple's published data-type definitions and reflects how these items should be disclosed to end users.

6.7 Summary of Findings

We summarize our findings from large-scale analysis as follows.

Prevalence of Non-Compliance. 188/324 (58.0%) AVP apps violate one or more privacy disclosure requirements and 61.1% of observed privacy-related data flows occur without proper disclosure. This is a conservative lower bound, as we only performed 20-minute exploration.

Policy, Label and Manifest Non-Compliance. Omissions and inaccuracies are common in both privacy policies and App Store privacy labels. Adoption of privacy manifests remains inconsistent and frequently superficial; nearly half of apps with manifests omit required elements or fail to accurately represent observed data flows.

SDKs as Primary Compliance Bottleneck. Third-party SDKs account for the majority of undisclosed network traffic, typically outside the direct control or awareness of app developers.

Purpose Misalignment. Substantial discrepancies persist between declared purposes and actual runtime data usage, undermining the integrity of App Store privacy disclosures.

Developer Uncertainty. Ambiguities in Apple's data taxonomy, coupled with insufficient tooling for SDK attribution, contribute to confusion and inconsistent compliance among developers.

AVP-specific leakage. AVP-specific data account for only a small fraction of observed violations. Of 3,452 violations, 46 (1.3%) map to AVP-specific taxonomy nodes. Manual inspection confirms that none of the 46 contains raw XR sensor measurements. Importantly, our static analysis of all 374 app binaries shows that 54 declare an XR sensor usage string and 39 link an authorization-gated ARKit provider, indicating that such capabilities are present in the ecosystem, yet we observe no corresponding raw sensor transmission. This finding is consistent with visionOS's privacy architecture: Apple states that the system mediates camera, sensor, gaze, and hand inputs without directly exposing them to apps, while more sensitive providers require explicit authorization in restricted immersive contexts [44]. Thus, the absence of raw XR telemetry provides encouraging empirical evidence that visionOS's system-mediated sensing boundary limits direct network exposure of raw sensor inputs.

7 Discussion

We discuss the implications and recommendations. We further discuss our limitations and future work in Appendix J.

Implications. AVP-specific sensing data can reveal physical behavior and surrounding context. Even where such streams remain on-device, the identifiers that are sent through network (e.g., User ID and Device ID shared with third parties) enable cross-app linkage and user profiling, and the small, distinctive AVP install base makes that linkage easier than the equivalent on a phone.

Quantitatively, our non-compliance rate is comparable to OVRSeen's Meta Quest findings (58.0% of AVP apps with at least one violation vs. 58.6% on Quest). VisionOS providing richer privacy disclosures

and manifests, but these additional disclosure mechanisms do not translate into more consistent compliance. We attribute this to two factors. First, the AVP ecosystem is still relatively immature, and developers may not yet have established reliable practices for accurately documenting the privacy data. Second, visionOS is a closed platform that provides limited visibility into application behavior: without root access or third-party auditing tools, developers have few practical ways to independently verify how privacy data flows through their applications. AVP-INSPECT addresses this gap by providing an external method to audit these flows.

Recommendations. Based on our findings, we offer recommendations for three stakeholder groups in the AVP ecosystem:

For AVP Users. Users should not rely solely on App Store privacy labels or manifest disclosures, as our findings show these are frequently incomplete or inaccurate. Users are advised to routinely review and manage app permissions, restrict access to sensitive device features, and uninstall or avoid apps with a history of privacy violations. Where feasible, users should favor applications from developers with strong, transparent privacy practices.

For the AVP Platform (Apple). The current self-reporting framework for privacy labels and manifests is inadequate for ensuring compliance. Apple should implement automated, ongoing runtime verification of privacy declarations, and systematically audit both in-house and third-party SDKs for undisclosed data collection. Policy reforms should require explicit data usage declarations for all bundled SDKs, enforce penalties for repeated non-compliance, and provide users with increased transparency, such as in-app notifications about ongoing data flows and results from independent privacy audits. For example, Apple can provide an option for users to see the ongoing data flows and the related privacy-sensitive data categories of the foreground app. Concretely, we recommend three classes of actions. *1) Documentation*: provide a finer-grained AVP-specific taxonomy for spatial and biometric disclosures, and treat `x-platform: visionOS` sent to third parties as tracking-relevant in privacy manifests. *2) Enforcement*: surface SDK-level privacy manifests to developers, and cross-check runtime traffic against machine-readable manifests during app vetting. *3) APIs*: add distinct runtime permissions and persistent indicators for spatial and biometric sensing APIs.

For AVP Developers. Developers must perform comprehensive audits of all integrated SDKs and explicitly disclose all collected data types and purposes in both App Store privacy labels and manifests. Automated privacy compliance checks should be integrated into the development lifecycle, with disclosures updated promptly in response to any code or SDK changes. Developers are responsible for ensuring that privacy documentation remains accurate and up-to-date, reflecting the actual behavior of both first- and third-party code within their apps. Concretely, we recommend four classes of actions: 1) audit third-party SDKs through runtime traffic inspection; 2) disclose third-party recipients and purposes, not only first-party collection; 3) keep privacy policies, labels, and manifests synchronized; and 4) update all privacy documents whenever the app or an embedded SDK changes.

8 Related Work

We discuss prior research on XR privacy analysis and XR dynamic testing. Additional related work on XR privacy attacks and defenses, as well as iOS privacy analysis, is covered in Appendix I.

XR privacy analysis. Ovrseen [17] is the first work to study inconsistencies between network traffic and privacy policies of VR applications on Oculus (Meta) platforms. VPVet [18] is a recent tool for vetting privacy policies across 10 mainstream VR platforms. Guo *et al.* [45] have analyzed 500 VR apps to study their security and privacy issues, and Simhadri *et al.* [46] performed a longitudinal analysis on 300+ firmware from Quest and Pico devices. Brunskong *et al.* [47] presented a network traffic analysis associated with child and adult accounts of six Meta Quest applications, raising concerns about insufficient age-appropriate data practices on VR platforms. Different from those works which focus on Meta devices, we perform the first privacy analysis on AVP devices based on network traffic using automated testing.

Apple privacy labels. Prior work has studied Apple privacy labels from user, developer, and ecosystem perspectives, showing that users may struggle to interpret labels [48, 49], developers face challenges mapping app and SDK behavior to Apple's data categories [50–52], and labels are not always updated as app behavior changes [50]. Prior work has also studied inconsistencies between privacy labels and privacy policies [53], as well as inconsistencies between iOS mobile app traffic and privacy labels using either passive/no-touch traffic collection [54] or UI-driven dynamic analysis [42]. Our work complements this line of research by extending privacy-label compliance analysis from traditional mobile apps to Apple Vision Pro apps. Our results further show that disclosure inconsistencies are also prevalent in the AVP ecosystem. We attribute this in part to two factors: the relative immaturity of the AVP ecosystem, and the closed nature of the platform. Without root access or mature third-party auditing tools, developers cannot easily verify what their own apps and embedded SDKs transmit, which is the gap AVP-Inspect fills.

XR dynamic testing. Prior XR testing work targets platforms with source access or standard testing suites. White-box and model-based tools such as VRTest [55], VRGuide [56], and VRExplorer [57] automate scene exploration on Unity/SteamVR; Youkai [58] and VR-ReST [59] address unit testing and requirements-driven test generation. Other efforts detect visual defects such as stereoscopic inconsistency [60] and AR object-placement issues [61], and Gu *et al.* [62] survey this landscape. Recently, AutoVR [63] was introduced to test Unity-based VR applications and detect sensitive data exposure using Frida. Different from our work, these approaches 1) target platforms (e.g., SteamVR) with standard testing suites, 2) rely on source code, root capability, or use dynamic instrumentation tools, which are not applicable to AVP due to the challenges mentioned in §3.

Non-intrusive GUI testing. RoboTest [64] is closest to our Controller: it physically actuates smartphones with a robotic arm and recovers widgets from screenshots. We share its non-intrusive, vision-based premise, but AVP has no touch surface and no absolute screen-to-actuator mapping. AVP-Inspect therefore uses a low-cost hybrid controller: Bluetooth HID for pointer/keyboard input and

servo motors for hardware buttons. Its control problem is also different: AVP cursor motion is relative and depth-dependent, requiring feedback-guided binary search and perceptual state deduplication under head-pose variation. Finally, RoboTest targets crash/compatibility bugs, whereas AVP-Inspect audits privacy compliance from network traffic.

9 Conclusion

In this paper, we presented AVP-Inspect, a novel testing framework that enables automated privacy analysis of Apple Vision Pro applications, without requiring root access or application source code. We first evaluated AVP-Inspect using a 50-app ground-truth dataset to demonstrate its effectiveness, and we performed a large-scale analysis on 324 AVP applications, which reveal that 188 apps have non-compliance issues. We have reported our findings to Apple and app developers, and we offer recommendations to help them mitigate such issues. Our findings highlight the need for improved privacy enforcement mechanisms in emerging XR platforms and provide a foundation for future privacy research in immersive computing environments.

Acknowledgment

The authors from George Mason University (GMU) are supported in part by 1) a seed funding and GRA awards from the CAHMP (now CHAIS) Center at GMU, and 2) a seed funding from 4-VA, a collaborative partnership for advancing the Commonwealth of Virginia. Yue Xiao is supported in part by the Commonwealth Cyber Initiative (CCI-HC-2Q26-037).

References

[1] Mordor Intelligence, "Extended Reality Market Size & Share Analysis (2024 - 2029)." https://www.mordorintelligence.com/industry-reports/extended-reality-xr-market, 2024.
[2] Meta Newsroom, "Metaverse technologies are creating new opportunities for teachers to inspire students." https://about.fb.com/news/2023/09/metaverse-technologies-education-opportunities/, 2023.
[3] US Chamber of Commerce, "How virtual reality is transforming healthcare." https://www.uschamber.com/technology/how-virtual-reality-is-transforming-healthcare, 2022.
[4] US Department of Defense, "Virtual, augmented reality are moving warfighting forward." https://www.defense.gov/News/Feature-Stories/Story/Article/2079205/virtual-augmented-reality-are-moving-warfighting-forward/, 2020.
[5] D. Hancko, A. Majlingova, and D. Kačíková, "Integrating virtual reality, augmented reality, mixed reality, extended reality, and simulation-based systems into fire and rescue service training: Current practices and future directions," *Fire*, vol. 8, no. 6, 2025.
[6] Apple, "Apple Vision Pro." https://www.apple.com/apple-vision-pro/, 2025.
[7] FOX business, "Apple vision pro headset becomes tool in operating room." https://www.foxbusiness.com/technology/apple-vision-pro-headset-becomes-tool-operating-room, 2024.
[8] Y. Zhang, C. Slocum, J. Chen, and N. Abu-Ghazaleh, "It's all in your head (set): Side-channel attacks on {AR/VR} systems," in *USENIX Security'23*, 2023.
[9] Y. Wu, C. Shi, T. Zhang, P. Walker, J. Liu, N. Saxena, and Y. Chen, "Privacy leakage via unrestricted motion-position sensors in the age of virtual reality: A study of snooping typed input on virtual keyboards," in *IEEE S&P'23*.
[10] S. Luo, X. Hu, and Z. Yan, "Hollogger: Keystroke inference on mixed reality head mounted displays," in *IEEE VR'22*.
[11] C. Slocum, Y. Zhang, N. Abu-Ghazaleh, and J. Chen, "Going through the motions:{AR/VR} keylogging from user head motions," in *USENIX Security'23*, 2023.
[12] U. Meteriz-Yıldıran, N. F. Yıldıran, A. Awad, and D. Mohaisen, "A keylogging inference attack on air-tapping keyboards in virtual environments," in *IEEE VR'22*, 2022.
[13] D. Cayir, R. Mohamed, R. Lazzeretti, M. Angelini, A. Acar, M. Conti, Z. B. Celik, and S. Uluagac, "Speak Up, I'm listening: Extracting speech from zero-permission VR sensors," in *NDSS'25*, 2025.
[14] Z. Ye, A. T. Mahdad, Y. Wang, C. Shi, Y. Chen, and N. Saxena, "BPSniff: Continuously Surveilling Private Blood Pressure Information in the Metaverse via Unrestricted Inbuilt Motion Sensors," in *S&P'25*, 2025.
[15] GDPR, "General data protection regulation." https://gdpr-info.eu/, 2018.
[16] CCPA, "California consumer privacy act." https://oag.ca.gov/privacy/ccpa, 2020.
[17] R. Trimananda, H. Le, H. Cui, J. T. Ho, A. Shuba, and A. Markopoulou, "{OVRseen}: Auditing network traffic and privacy policies in oculus {VR}," in *USENIX Security'22*, 2022.
[18] Y. Zhan, Y. Meng, L. Zhou, Y. Xiong, X. Zhang, L. Ma, G. Chen, Q. Pei, and H. Zhu, "Vpvet: Vetting privacy policies of virtual reality apps," in *CCS'24*, 2024.
[19] MalwareBytes, "iPhone Jailbreaking." https://www.malwarebytes.com/iphone-jailbreaking, 2025.
[20] Frida, "iOS | Frida | A world-class dynamic instrumentation toolkit." https://frida.re/docs/ios/, 2025.
[21] OWASP Mobile Application Security, "MASTG-TECH-0067: Dynamic Analysis on iOS." https://mas.owasp.org/MASTG/techniques/ios/MASTG-TECH-0067/, 2025.
[22] B. Andow, S. Y. Mahmud, W. Wang, J. Whitaker, W. Enck, B. Reaves, K. Singh, and T. Xie, "{PolicyLint}: investigating internal privacy policy contradictions on google play," in *USENIX Security'19*, 2019.
[23] B. Andow, S. Y. Mahmud, J. Whitaker, W. Enck, B. Reaves, K. Singh, and S. Egelman, "Actions speak louder than words:{Entity-Sensitive} privacy policy and data flow analysis with {PoliCheck}," in *USENIX Security'20*, 2020.
[24] H. Cui, R. Trimananda, A. Markopoulou, and S. Jordan, "{PoliGraph}: Automated privacy policy analysis using knowledge graphs," in *USENIX Security'23*, 2023.
[25] Apple, "Submit your apps to the App Store for Apple Vision Pro." https://developer.apple.com/visionos/submit/, 2025.
[26] Apple, "App privacy details on the App Store." https://developer.apple.com/app-store/app-privacy-details/#data-type, 2025.
[27] Apple, "Privacy manifest files." https://developer.apple.com/documentation/bundleresources/privacy-manifest-files, 2025.
[28] Macaca, "Macaca: Solution for automation test with ease." https://macacajs.github.io, 2025.
[29] Sauce Labs, "Getting started with xcuitest." https://saucelabs.com/resources/blog/getting-started-with-xcuitest, 2022.
[30] Android Developers, "Android debug bridge (adb) | android studio." https://developer.android.com/tools/adb, 2025.
[31] 9To5Mac, "Hands-on with the $299 Apple Vision Pro Developer Strap – stabler Mac Virtual Display, beta downgrades, and more." https://9to5mac.com/2024/02/06/hands-on-with-the-299-apple-vision-pro-developer-strap-stabler-mac-virtual-display-beta-downgrades-and-more/, 2024.
[32] Apple, "Accessory Design Guidelines for Apple Devices." https://developer.apple.com/accessories/Accessory-Design-Guidelines.pdf, 2025.
[33] Apple, "Close an app on Apple Vision Pro." https://support.apple.com/en-us/118514, 2024.
[34] Wikipedia, "Servomotor." https://en.wikipedia.org/wiki/Servomotor, 2025.
[35] https://en.wikipedia.org/wiki/Circle_Hough_Transform.
[36] Apple, "Move, resize, and close app windows on Apple Vision Pro." https://support.apple.com/guide/apple-vision-pro/move-resize-and-close-app-windows-dev009366408/visionos, 2024.
[37] Microsoft, "A simple screen parsing tool towards pure vision based GUI agent." https://github.com/microsoft/OmniParser, 2025.
[38] Vision Directory Developers, "Vision Directory." https://vision.directory/, 2025.
[39] X. Wang, X. Qin, M. B. Hosseini, R. Slavin, T. D. Breaux, and J. Niu, "Guileak: Tracing privacy policy claims on user input data for android applications," in *ICSE'18*, 2018.
[40] S. Zimmeck, P. Story, D. Smullen, A. Ravichander, Z. Wang, J. Reidenberg, N. C. Russell, and N. Sadeh, "Maps: Scaling privacy compliance analysis to a million apps," *Proceedings on privacy enhancing technologies*, 2019.
[41] L. Yu, X. Luo, J. Chen, H. Zhou, T. Zhang, H. Chang, and H. K. Leung, "Ppchecker: Towards accessing the trustworthiness of android apps' privacy policies," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, 2018.
[42] Y. Xiao, Z. Li, Y. Qin, X. Bai, J. Guan, X. Liao, and L. Xing, "Lalaine: Measuring and characterizing {Non-Compliance} of apple privacy labels," in *USENIX Security'23*, 2023.
[43] National Institute of Standards and Technology, "NIST privacy framework: A tool for improving privacy through enterprise risk management, version 1.0," 2020.
[44] Apple, "Apple Vision Pro." https://developer.apple.com/documentation/visionos/adopting-best-practices-for-privacy/, 2025.
[45] H. Guo, H.-N. Dai, X. Luo, Z. Zheng, G. Xu, and F. He, "An Empirical Study on Oculus Virtual Reality Applications: Security and Privacy Perspectives," in *ICSE'24*, ACM, Apr. 2024.
[46] V. S. Simhadri, Y. Xiong, H. Farrukh, and X. Zhang, "Virtual Reality, Real Problems: A Longitudinal Security Analysis of VR Firmware," in *CCS'25*, ACM, 2025.

[47] E. Brunskog, S. Knutas, S. Mirzakhani, and N. Carlsson, "A network-level comparison of privacy risks for children vs. adults in virtual reality," in *WPES'25*, 2025.

[48] S. Zhang, Y. Feng, Y. Yao, L. F. Cranor, and N. Sadeh, "How usable are ios app privacy labels?," *Proceedings on Privacy Enhancing Technologies*, 2022.

[49] M. Tahaei, R. Abu-Salma, and A. Rashid, "Stuck in the permissions with you: Developer & end-user perspectives on app permissions & their privacy ramifications," in *CHI'23*, 2023.

[50] T. Li, K. Reiman, Y. Agarwal, L. F. Cranor, and J. I. Hong, "Understanding challenges for developers to create accurate privacy nutrition labels," in *CHI'22*, 2022.

[51] A. Alsahdi, J. Yan, M. Kodwani, M. Fassl, C. Kanich, and A. J. Aviv, ""because i didn't touch these and even don't know why i should to change these": Why app developers do (not) update apple's privacy labels," *Proceedings on Privacy Enhancing Technologies*, 2026.

[52] Y. Xiao, C. Zhang, Y. Qin, F. F. S. Alharbi, L. Xing, and X. Liao, "Measuring compliance implications of third-party libraries' privacy label disclosure guidelines," in *CCS'24*, 2024.

[53] A. Jain, D. Rodriguez, J. M. Del Alamo, and N. Sadeh, "Atlas: Automatically detecting discrepancies between privacy policies and privacy labels," in *EuroS&PW'23*, IEEE, 2023.

[54] S. Koch, M. Wessels, B. Altpeter, M. Olvermann, and M. Johns, "Keeping privacy labels honest," *Proceedings on Privacy Enhancing Technologies*, 2022.

[55] X. Wang, "Vrtest: an extensible framework for automatic testing of virtual reality scenes," in *ICSE Companion'22*, 2022.

[56] X. Wang, T. Rafi, and N. Meng, "Vrguide: Efficient testing of virtual reality scenes via dynamic cut coverage," in *ASE'23*, IEEE, 2023.

[57] Z. Zhu, H.-N. Dai, H. Guo, Z. Liao, and Z. Zheng, "Vrexplorer: A model-based approach for semi-automated testing of virtual reality scenes," in *ASE'25*, 2025.

[58] T. Figueira and A. Gil, "Youkai: A cross-platform framework for testing vr/ar apps," in *HCII'22*, Springer, 2022.

[59] A. C. Correa Souza, F. L. Nunes, and M. E. Delamaro, "An automated functional testing approach for virtual reality applications," *Software Testing, Verification and Reliability*, vol. 28, no. 8, 2018.

[60] S. Li, C. Gao, J. Zhang, Y. Zhang, Y. Liu, J. Gu, Y. Peng, and M. R. Lyu, "Less cybersickness, please: Demystifying and detecting stereoscopic visual inconsistencies in virtual reality apps," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, 2024.

[61] T. Rafi, X. Zhang, and X. Wang, "Predart: Towards automatic oracle prediction of object placements in augmented reality testing," in *ASE'22*, 2022.

[62] R. Gu, J. M. Rojas, and D. Shin, "Software testing for extended reality applications: a systematic mapping study," *Automated Software Engineering*, vol. 32, no. 2, 2025.

[63] J. Y. Kim, C. Zuo, Y. Zhao, and Z. Lin, "Autovr: Automated ui exploration for detecting sensitive data flow exposures in virtual reality apps," in *USENIX Security'25*.

[64] S. Yu, C. Fang, M. Du, Y. Ling, Z. Chen, and Z. Su, "Practical non-intrusive GUI exploration testing with visual-based robotic arms," in *ICSE'24*, 2024.

[65] A. G. Moore, R. P. McMahan, H. Dong, and N. Ruozzi, "Personal identifiability and obfuscation of user tracking data from vr training sessions," in *ISMAR'21*, IEEE, 2021.

[66] V. Nair, W. Guo, J. Mattern, R. Wang, J. F. O'Brien, L. Rosenberg, and D. Song, "Unique identification of 50,000+ virtual reality users from head & hand motion data," in *USENIX Security'23*, 2023.

[67] M. R. Miller, F. Herrera, H. Jun, J. A. Landay, and J. N. Bailenson, "Personal identifiability of user tracking data during observation of 360-degree VR video," *Scientific Reports*, vol. 10, no. 1, 2020. Publisher: Nature Publishing Group UK London.

[68] M. R. Miller, E. Han, C. DeVeaux, E. Jones, R. Chen, and J. N. Bailenson, "A Large-Scale Study of Personal Identifiability of Virtual Reality Motion Over Time," Mar. 2023. arXiv:2303.01430 [cs].

[69] R. Miller, N. K. Banerjee, and S. Banerjee, "Using siamese neural networks to perform cross-system behavioral authentication in virtual reality," in *IEEE VR'21*.

[70] R. Miller, N. K. Banerjee, and S. Banerjee, "Within-system and cross-system behavior-based biometric authentication in virtual reality," in *VRW'20*, IEEE, 2020.

[71] K. Pfeuffer, M. J. Geiger, S. Prange, L. Mecke, D. Buschek, and F. Alt, "Behavioural Biometrics in VR: Identifying People from Body Motion and Relations in Virtual Reality," in *CHI'19*, ACM, 2019.

[72] C. Rack, K. Kobs, T. Fernando, A. Hotho, and M. E. Latoschik, "Extensible Motion-based Identification of XR Users using Non-Specific Motion Data," *arXiv preprint arXiv:2302.07517*, 2023.

[73] J. Liebers, C. Burschik, U. Gruenefeld, and S. Schneegass, "Exploring the Stability of Behavioral Biometrics in Virtual Reality in a Remote Field Study: Towards Implicit and Continuous User Identification through Body Movements," in *VRST'23*, ACM, Oct. 2023.

[74] J. Liebers, S. Brockel, U. Gruenefeld, and S. Schneegass, "Identifying Users by Their Hand Tracking Data in Augmented and Virtual Reality," *International Journal of Human–Computer Interaction*, vol. 40, Jan. 2024.

[75] J. Liebers, M. Abdelaziz, L. Mecke, A. Saad, J. Auda, U. Gruenefeld, F. Alt, and S. Schneegass, "Understanding User Identification in Virtual Reality Through Behavioral Biometrics and the Effect of Body Normalization," in *CHI'21*, ACM, May 2021.

[76] M. Sabra, N. V. Sureshkanth, A. Sharma, A. Maiti, and M. Jadliwala, "Exploiting Out-of-band Motion Sensor Data to De-anonymize Virtual Reality Users," Jan. 2023. arXiv:2301.09041 [cs].

[77] I. Jarin, Y. Duan, R. Trimananda, H. Cui, S. Elmalaki, and A. Markopoulou, "BehaVR: User Identification Based on VR Sensor Data," Aug. 2023. arXiv:2308.07304 [cs].

[78] Y. Meng, Y. Zhan, J. Li, S. Du, H. Zhu, and X. S. Shen, "De-anonymization Attacks on Metaverse," in *INFOCOM'23*, IEEE, 2023.

[79] G. M. Garrido, V. Nair, and D. Song, "SoK: Data Privacy in Virtual Reality," *Proceedings on Privacy Enhancing Technologies*, 2024.

[80] S. Stephenson, B. Pal, S. Fan, E. Fernandes, Y. Zhao, and R. Chatterjee, "Sok: Authentication in augmented and virtual reality," in *IEEE S&P'22*.

[81] J. Liebers, P. Laskowski, F. Rademaker, L. Sabel, J. Hoppen, U. Gruenefeld, and S. Schneegass, "Kinetic Signatures: A Systematic Investigation of Movement-Based User Identification in Virtual Reality," in *CHI'24*, ACM, 2024.

[82] V. Nair, L. Rosenberg, J. F. O'Brien, and D. Song, "Truth in motion: The unprecedented risks and opportunities of extended reality motion data," *IEEE Security & Privacy*, 2023. Publisher: IEEE.

[83] V. Nair, C. Rack, W. Guo, R. Wang, S. Li, B. Huang, A. Cull, J. F. O'Brien, M. Latoschik, L. Rosenberg, and D. Song, "Inferring Private Personal Attributes of Virtual Reality Users from Head and Hand Motion Data," June 2023. arXiv:2305.19198 [cs].

[84] S. Lee and W. Choi, "Eyes on your typing: Snooping finger motions on virtual keyboards," in *IEEE S&P'25*, IEEE, 2025.

[85] S. R. K. Gopal, D. Shukla, J. D. Wheelock, and N. Saxena, "Hidden reality: caution, your hand gesture inputs in the immersive virtual world are visible to all!," in *USENIX Security'23*, 2023.

[86] A. Nguyen, X. Zhang, and Z. Yan, "Penetration Vision through Virtual Reality Headsets: Identifying 360-degree Videos from Head Movements," in *USENIX Security'24*, 2024.

[87] Z. Yang, Z. Sarwar, I. Hwang, R. Bhaskar, B. Y. Zhao, and H. Zheng, "Can Virtual Reality Protect Users from Keystroke Inference Attacks?," *arXiv preprint arXiv:2310.16191*, 2023.

[88] S. Luo, A. Nguyen, H. Farooq, K. Sun, and Z. Yan, "Eavesdropping on Controller Acoustic Emanation for Keystroke Inference Attack in Virtual Reality," in *NDSS'24*, 2024.

[89] Z. Su, K. Cai, R. Beeler, L. Dresel, A. Garcia, I. Grishchenko, Y. Tian, C. Kruegel, and G. Vigna, "Remote keylogging attacks in multi-user {VR} applications," in *USENIX Security'24*, 2024.

[90] T. Ni, Y. Du, Q. Zhao, and C. Wang, "Non-intrusive and unconstrained keystroke inference in vr platforms via infrared side channel," in *NDSS'25*, 2025.

[91] A. Al Arafat, Z. Guo, and A. Awad, "Vr-spy: A side-channel attack on virtual key-logging in vr headsets," in *IEEE VR'21*.

[92] S. Son, C. Mukherjee, R. M. Aburas, B. Gulmezoglu, and Z. B. Celik, "Side-channel inference of user activities in ar/vr using gpu profiling," *arXiv preprint arXiv:2509.10703*, 2025.

[93] S. Aziz and O. Komogortsev, "Exploring the uncoordinated privacy protections of eye tracking and vr motion data for unauthorized user identification," in *IEEE VR'25*, IEEE, 2025.

[94] H. Wang, Z. Zhan, H. Shan, S. Dai, M. Panoff, and S. Wang, "Gazeploit: Remote keystroke inference attack by gaze estimation from avatar views in vr/mr devices," in *CCS'24*, 2024.

[95] V. Nair, W. Guo, J. F. O'Brien, L. Rosenberg, and D. Song, "Deep Motion Masking for Secure, Usable, and Scalable Real-Time Anonymization of Virtual Reality Motion Data," Nov. 2023. arXiv:2311.05090 [cs].

[96] V. C. Nair, G. Munilla-Garrido, and D. Song, "Going Incognito in the Metaverse: Achieving Theoretically Optimal Privacy-Usability Tradeoffs in VR," in *UIST'23*, ACM, Oct. 2023.

[97] J. Li, A. R. Chowdhury, K. Fawaz, and Y. Kim, "{Kaleido}:{Real-Time} privacy control for {Eye-Tracking} systems," in *USENIX Security'21*, 2021.

[98] B. David-John, K. Butler, and E. Jain, "Privacy-preserving datasets of eye-tracking samples with applications in xr," *IEEE TVCG*, 2023.

[99] M. Shoaib, A. Suh, and W. U. Hassan, "Principled and automated approach for investigating {AR/VR} attacks," in *USENIX Security'25*, 2025.

[100] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *NDSS'11*, The Internet Society, 2011.

[101] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "iris: Vetting private API abuse in ios applications," in *CCS'15*, ACM, 2015.

[102] Y. Li, Y. Zhang, J. Li, and D. Gu, "icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications," in *NSS'15*, Springer, 2015.

[103] M. Zheng, H. Xue, Y. Zhang, T. Wei, and J. C. Lui, "Enpublic apps: Security threats using ios enterprise and developer certificates," in *ASIACCS'15*, 2015.

[104] Y. Agarwal and M. Hall, "Protectmyprivacy: detecting and mitigating privacy leaks on ios devices using crowdsourcing," in *MobiSys'13*, 2013.
[105] T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz, "Psios: bring your own privacy & security to ios devices," in *ASIACCS'13*, 2013.
[106] K. Kollnig, A. Shuba, M. Van Kleek, R. Binns, and N. Shadbolt, "Goodbye tracking? impact of ios app tracking transparency and privacy labels," in *FAccT'22*, 2022.
[107] Z. A. Surma, S. Gowdar, and H. J. Pandit, "Examining the integrity of apple's privacy labels: Gdpr compliance and unnecessary data collection in ios apps," *Information*, vol. 15, no. 9, 2024.
[108] D. Liu, Y. Xiao, C. Zhang, K. Xie, X. Bai, S. Zhang, and L. Xing, "{iHunter}: Hunting privacy violations at scale in the software supply chain on {iOS}," in *USENIX Security'24*, 2024.
[109] J. Zhang, J. Wu, T. Yihua, M. Liao, N. Xu, X. Xiao, Z. Wei, and D. Tang, "Android in the zoo: Chain-of-action-thought for gui agents," in *Findings of EMNLP'24*, 2024.
[110] J. Ye, X. Zhang, H. Xu, H. Liu, J. Wang, Z. Zhu, Z. Zheng, F. Gao, J. Cao, Z. Lu, *et al.*, "Mobile-agent-v3: Fundamental agents for gui automation," *arXiv preprint arXiv:2508.15144*, 2025.
[111] J. Yoon, R. Feldt, and S. Yoo, "Intent-driven mobile gui testing with autonomous large language model agents," in *ICST'24*, IEEE, 2024.
[112] Z. Zhang, Y. Lu, Y. Fu, Y. Huo, S. Yang, Y. Wu, H. Si, X. Cong, H. Chen, Y. Lin, *et al.*, "Agentcpm-gui: Building mobile-use agents with reinforcement fine-tuning," *arXiv preprint arXiv:2506.01391*, 2025.
[113] OpenCV. https://docs.opencv.org/3.4/d4/d70/tutorial_hough_circle.html.
[114] https://apps.apple.com/us/app/airlauncher-app-launcher/id6477550630.
[115] https://apps.apple.com/us/app/twin-scans-for-passthrough/id6499172465.

A Ethical Considerations

Stakeholders. Our study involves three primary stakeholder groups. First, AVP users rely on privacy disclosures to make informed decisions about app installations and data sharing. Second, app developers may face scrutiny from identified privacy violations, though our analysis focuses on systemic compliance issues rather than individual fault attribution. Third, Apple as the platform operator may face increased attention regarding privacy enforcement mechanisms, which we view as an opportunity to strengthen user protection.

Data Collection. We collected publicly available information from the App Store, including app metadata, privacy manifests, privacy labels, and privacy policies. Our data collection process was designed to minimize disruption to the platform by implementing rate limiting. We did not access any user data or private information during our study.

Network Traffic Analysis. We analyzed network traffic generated during automated app exploration. All traffic was captured from apps we installed on our own AVP device. We did not intercept or analyze traffic from other users. The analysis focused on identifying privacy-related data transmissions to assess compliance with disclosed privacy practices.

Researcher Safety. Authors who tested AVP applications during the study did so voluntarily. Testing sessions were limited in duration and discontinued immediately upon experiencing discomfort.

Responsible Disclosure. We are in the process of reporting our findings to Apple and affected app developers to allow them to address identified privacy violations before public disclosure. We disclosed our findings before submission. We reported aggregate results to Apple through the Feedback Assistant and App Store Review channels, and contacted 119 developers. No user data was collected; all traffic originates from apps installed on our own device under our own accounts.

B Open Science

Our artifact can be found at: https://github.com/SECSAT-LAB-GMU/AVP-Inspect

C Generative AI Usage

AI tools (e.g., ChatGPT) have been used to check grammar and polish the sentences.

D Algorithms

We present the two algorithms: 1) App state exploration (§4.2); 2) Discovery of new data types (§4.3).

E Taxonomy of AVP-Inspect

Fig. 11 shows the taxonomy in full.

F Runtime Performance of AVP-Inspect

Exploration efficiency. Fig. 13 and Fig. 14 show the cumulative UI clicks and network requests over 20-minute exploration sessions, grouped by app category. For comparability, both figures use the same subset of ground-truth apps that generated observable network traffic during exploration. Exploration patterns vary significantly across categories. Entertainment apps show consistent

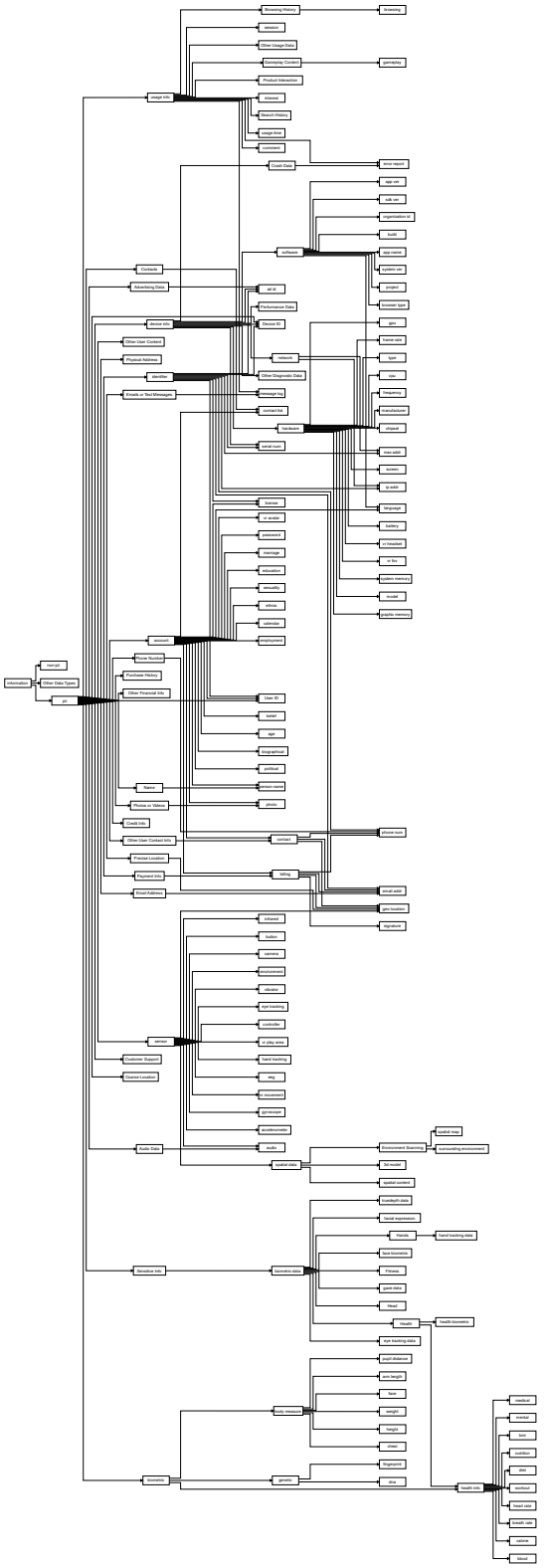


Fig. 11: Taxonomy

Algorithm 1: App State Exploration Algorithm

```
Function CheckIfStateIsKnown(state):
    if state ∈ G.nodes then
        return true;
    end
    return false;
Function CheckCurrentState():
    new_state ← GetElements(Screenshot());
    if CheckIfStateIsKnown(new_state) then
        ResumeExploreState(new_state);
    else
        G.nodes.append(new_state);
        G.edges.append(Button(id, pre_state, new_state));
    end
Function ResumeExploreState(state):
    while state.unexplored_buttons ≠ ∅ do
        button ← state.unexplored_buttons.pop();
        Click(button);
        CheckCurrentState();
    end
    TryClickBackButton();
    CheckCurrentState();
Function JumpToCurrentState(target_state):
    RestartApp(appid);
    foreach button ∈ GetPath(first_state, target_state) do
        Click(button);
    end
Function Main():
    first_state ← initial state of the app;
    G.nodes.append(first_state);
    ResumeExploreState(first_state);
    while ∃ state ∈ G with state.unexplored_buttons ≠ ∅ do
        state ← next state with unexplored buttons;
        JumpToCurrentState(state);
        ResumeExploreState(state);
    end
```

Algorithm 2: Iterative Discovery of New Data Types Using Embeddings

Input: Unmapped data phrases $D_{unmapped}$, similarity threshold $\theta = 0.8$
Output: Set of new data types N

$E \leftarrow$ GenerateEmbeddings($D_{unmapped}$);
$N \leftarrow \emptyset$ // New data types
$D_{remaining} \leftarrow D_{unmapped}$;
while $D_{remaining} \neq \emptyset$ **do**
  // Manually select a representative data type
  $t \leftarrow$ ManuallySelect($D_{remaining}$);
  $N \leftarrow N \cup \{t\}$;
  // Calculate similarity and filter similar phrases
  $D_{similar} \leftarrow \emptyset$;
  foreach *phrase* $d \in D_{remaining}$ **do**
    if *Similarity*$(d, t) \geq \theta$ **then**
      $D_{similar} \leftarrow D_{similar} \cup \{d\}$;
  // Remove filtered phrases from remaining set
  $D_{remaining} \leftarrow D_{remaining} \setminus D_{similar}$;
  if *no new data type can be manually identified* **then**
    break;
return N;

patterns, with apps like *Wet Your Beak*, achieving over 40 clicks due to well-structured UIs, and network traffic closely follows click pattern. In Utilities, *Broadcasts* shows the deepest UI exploration: its click count keeps increasing throughout the 20-minute session and reaches more than 70 clicks. In Education, *Foxar* generate substantial network traffic generate substantial network traffic relative to clicks, and it triggers multiple asset fetching per interaction.

Table 3: Per-app per-min unique network request comparison: AVP-Inspect's exploration (Auto) vs. idle baseline (Idle) for all 31 evaluation apps with captured traffic.

Category	App	Auto/min	Idle/min
Productivity	Rad Timer	0.13	0.04
	FreelanceKit	0.21	0.14
	Cardhop	0.12	0.05
	Focus - Timer	0.00	0.00
	FocusBeats	0.39	0.18
	Flippy	0.14	0.14
Utilities	Craft	1.80	0.36
	Broadcasts	10.82	0.05
	Qlone	0.10	0.05
	WWidgets	0.28	0.05
Games	Retrogram	0.81	0.31
	Things	2.50	7.85
Entertainment	Callsheet	119.60	3.12
	Turn Off the Lights	0.00	0.00
	Wet Your Beak	1.09	0.63
	Paramount+	12.58	0.31
Lifestyle & Health	alo Sanctuary	4.64	0.05
	Decathlon USA	0.41	0.05
	Mindr	0.06	0.04
	Bible	1.89	0.18
	Best Buy	2.51	0.14
Education	Chemistry	0.69	0.23
	Drawing Desk	0.71	0.00
	Foxar	3.75	0.36
	Inviewer	0.80	0.14
Creative	Kineo	0.14	0.05
	Spatial Station	2.36	0.00
	Theater	26.25	3.46
	Twin	0.57	0.14
	Cubes	0.40	0.14
Social	OverSoul	0.70	0.00
Total	**31**	**3.51**	**0.63**

Time breakdown. To understand the time cost of each exploration cycle, we analyze the breakdown of average time spent per click. As shown in Fig. 12, each click cycle consists of three parts: 1) UI state recognition by OmniParserV2, 2) virtual cursor movement, 3) state comparison to check whether a new state is reached. On average, Each click takes approximately 12.10s. The virtual cursor movement (9.76s) dominates due to the iterative visual calibration required for precise targeting, which is a necessary overhead for our hardware-based approach that works with any app without system modifications or root access.

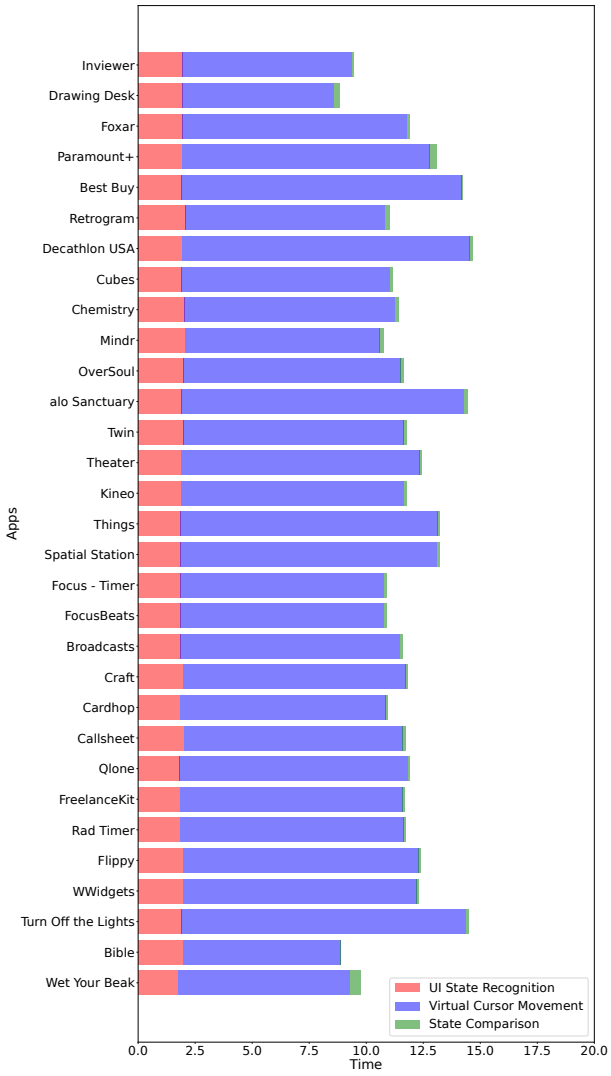


Fig. 12: Average time per click for 31 ground-truth apps.

G Large-Scale Analysis: Heatmaps

The heatmaps for privacy policy violation and manifest violation are presented in Fig. 15 and Fig. 16.

H SDK Analysis

Figures 17, 18, 19 show the SDK-related violations mentioned in §6.4.

I Additional Related Work

XR privacy attacks and defenses. Existing works mainly exploit machine learning models to perform side-channel attacks to breach user's privacy on XR platforms. Researchers have performed user identification based on motion data in XR [65–81]. Besides identifying users, motion data in XR has also been used to infer specific user attributes such as age and gender (*i.e.*, user profiling) [82, 83] from head and hand movement patterns. Other secrets such as keystrokes or speech in XR can also be inferred from motion data [8–14, 84], videos of user movements [85–87], acoustic signals [88], network traffic [89], infrared signals [90], WiFi signals [91] and GPU profiling data [92]. Aziz *et al.* [93] investigate how unprotected body motion data can weaken privacy safeguards for eye-tracking data, and conversely how eye-tracking data can compromise protections on motion data, ultimately facilitating user identification in VR. Wang *et al.* [94] demonstrated the first side-channel attack on AVP, where they used eye movement information exposed from AVP Persona to infer keystrokes. Recent works [95–98] have proposed injecting noises into motion data or identifiable anthropometrics to make inference attacks harder, which have predominantly used

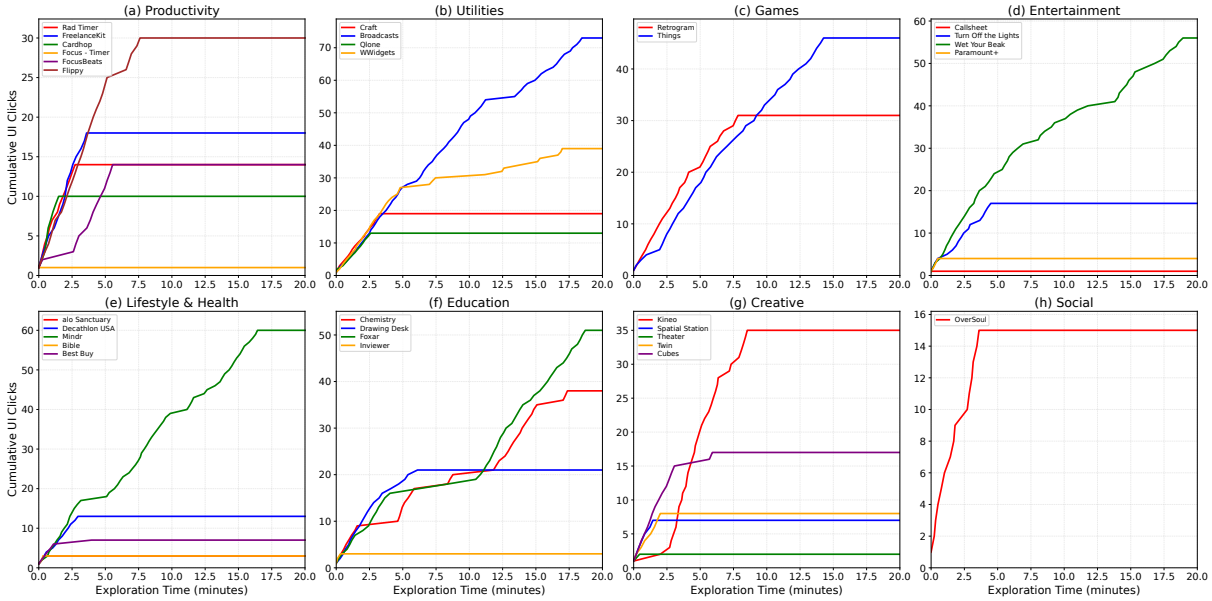


Fig. 13: Cumulative UI clicks over time during app exploration, grouped by app category.

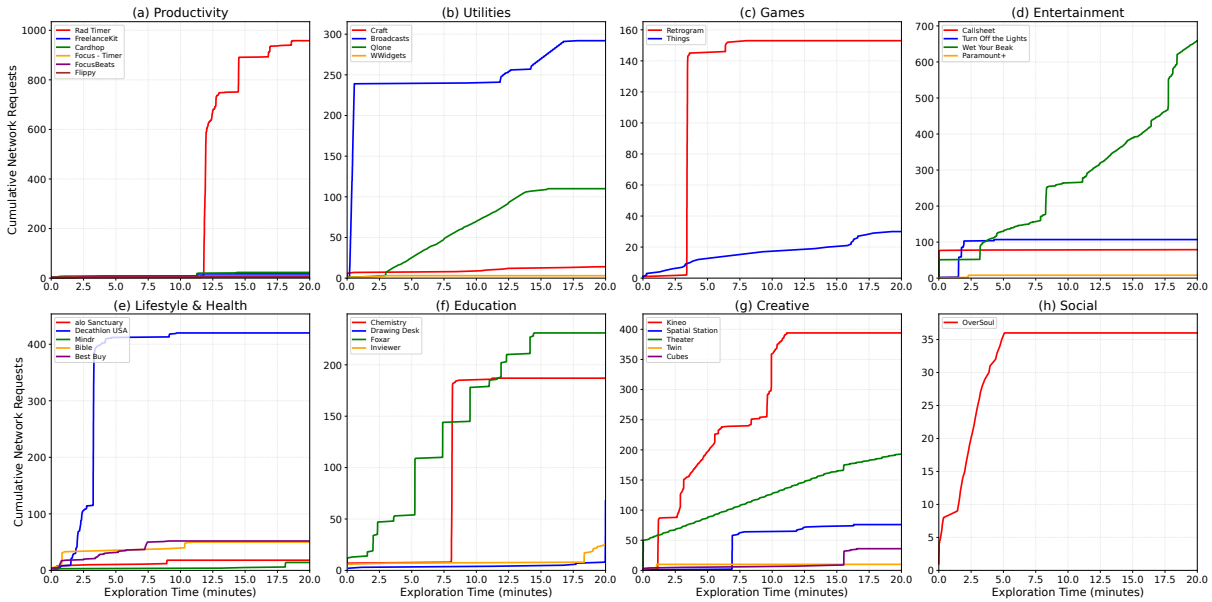


Fig. 14: Cumulative network requests over time during app exploration, grouped by app category.

differential privacy. Shoaib *et al.* proposed RealityCheck [99], a provenance-based auditing system and investigated 25 XR attacks. There are SoKs [79, 80] summarizing XR attacks and defenses.

iOS privacy analysis. PiOS [100] applied static analysis to Mach-O binaries to track data flows, revealing that about half of 1,400 analyzed apps leaked the device's unique ID. iRis [101] combined static and iterative dynamic analysis to detect security-critical private API usage, identifying 146 (7%) apps that accessed sensitive data. Other works have used source-to-sink analysis to detect cryptographic API misuse (iCryptoTracer [102]) or private API usage that exposes personally identifiable information (iAnalytics [103]). Additional studies include a crowdsourcing effort [104] showing the prevalence of privacy-sensitive resource access, and a proposal for fine-grained, user-driven sandboxing for runtime privacy protection [105]. Beyond invasive behavior detection, recent work has focused on privacy compliance analysis, assessing whether observed data practices align with privacy documentation such as Apple's

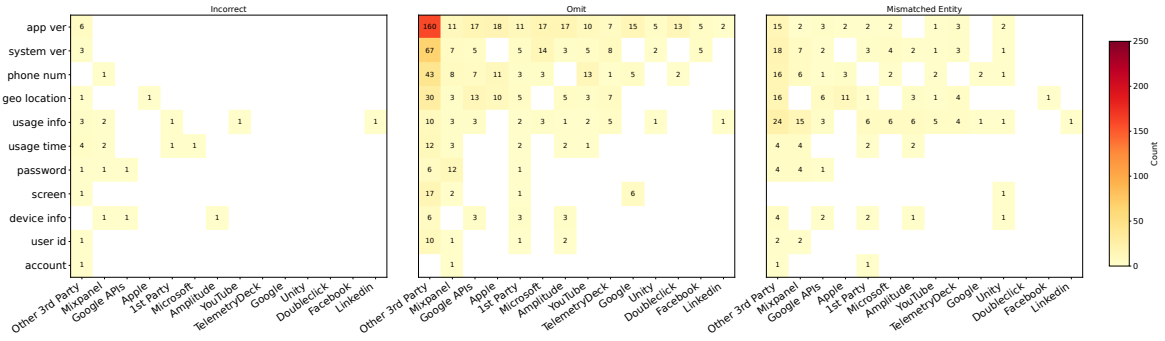


Fig. 15: Results: Privacy Policy (Partial)

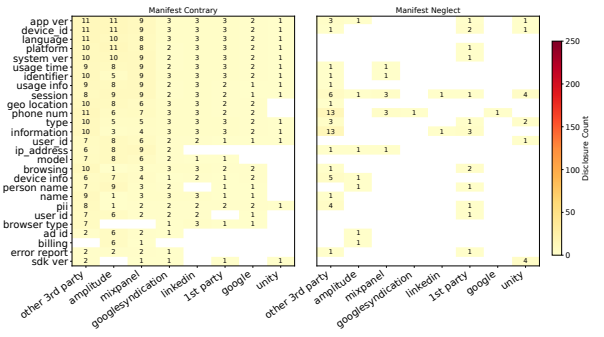


Fig. 16: Results: Privacy Manifest (Partial)

Table 4: Four apps have immersive scenes in our ground-truth dataset; our manual analysis showed that none of them generates network traffic.

App	Immersive functionality
Spatial Physics Playground	Rendering VR backgrounds
What If...?	Interacting with prebuilt Disney characters
3D Tic Tac Toe	Playing 3D interactive games
Breadpad	Breadboard simulator

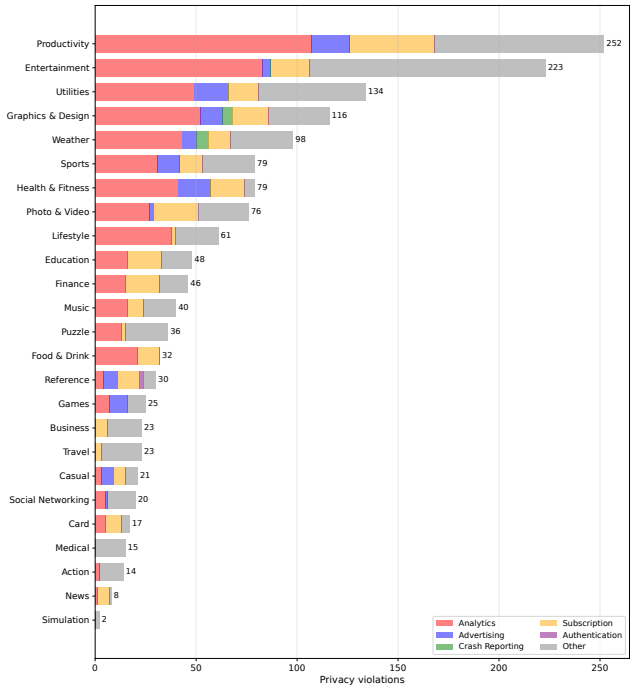


Fig. 17: Privacy-policy violations per app category w.r.t. SDKs; colors indicate the SDK category of the receiving entity.

privacy labels, privacy policies, and disclosure guidelines. Recent studies [42, 106, 107] examined the accuracy of Apple's privacy labels and their consistency with actual data handling behaviors. Other work has focused on third-party libraries: Colaine [52] leveraged NLP and dynamic analysis to verify compliance with privacy label disclosure guidelines, while iHunter [108] employed static taint analysis with iOS-specific symbolic execution and NLP-based taint rule generation to identify compliance violations in the iOS software supply chain. Unlike prior work that focuses on traditional iOS apps and SDKs, our work specifically investigates privacy risks and compliance issues in AVP apps.

J More Discussions

J.1 Traffic Analysis for Immersive Scenes

There are four apps (Table 4) in our ground-truth dataset that have immersive scenes, which AVP-Inspect cannot explore due to our limitations. However, through manual analysis on the immersive scenes in those apps, we found that they did not generate any network traffic. Therefore, the immersive scenes are unlikely to impact our final results.

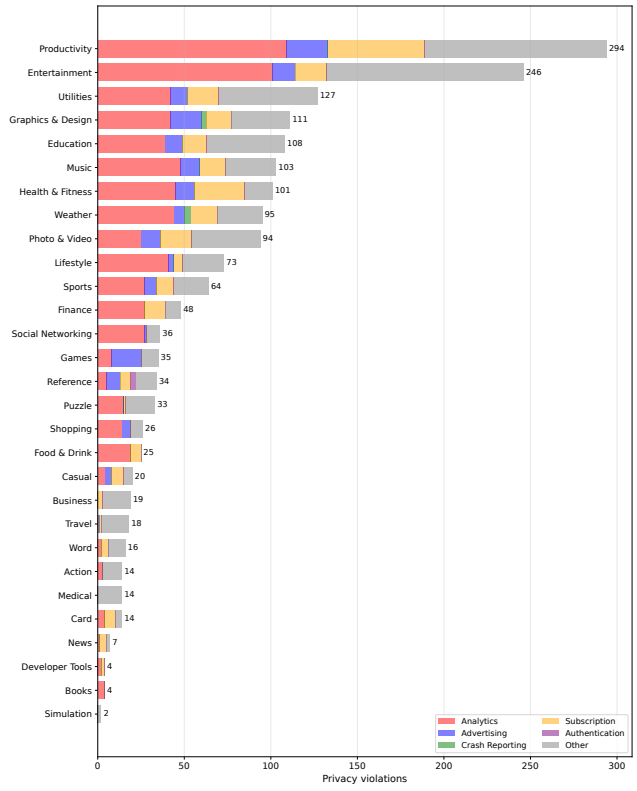


Fig. 18: Privacy-label violations per app category w.r.t. SDKs; colors indicate the SDK category of the receiving entity.

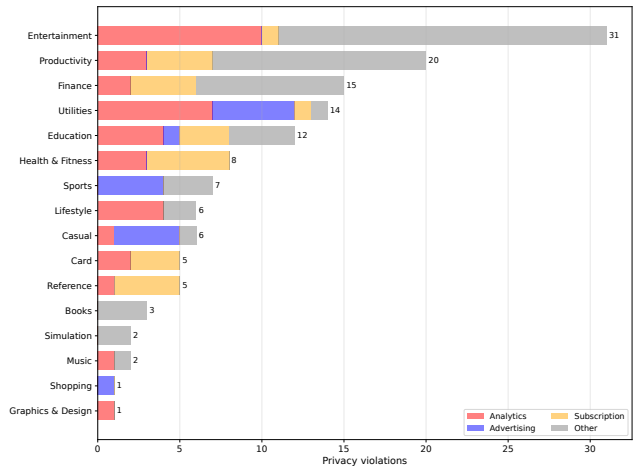


Fig. 19: Privacy-manifest violations per app category w.r.t. SDKs; colors indicate the SDK category of the receiving entity.

J.2 Limitations

The current design of AVP-Inspect has the following limitations. First, AVP-Inspect relies on the image recognition tool (`OmniParserV2` in our implementation) to recognize the clickable objects, which often generates randomness and prone to errors. Second, while AVP-Inspect can simulate user inputs such as clicks, it cannot simulate complex user interactions that require hand movements. As a result, we cannot handle complex immersive scenes. However, our manual exploration showed that only 4 out of 50 ground truth apps have such scenes, and those scenes do not send any traffic (Table 4 in Appendix §J.1).

Moreover, our UI-based exploration cannot create accounts or complete in-app purchases, so functionality behind such gates is unreachable regardless of exploration time. We manually audited the recorded sessions of all 50 ground-truth apps. Two apps place a login wall in front of essentially all functionality, so our traversal of those apps is limited to their pre-authentication screens. Five apps gate a subset of premium features behind an in-app purchase; because we did not transact, we cannot characterize what those paths would expose. The remaining 43 apps are fully explorable. Therefore, our method can already cover 48/50 apps.

Our taxonomy inherits its synonym lists verbatim from PoliCheck-/VPVet, and a few are broader than the AVP setting warrants. The clearest case is *password*, whose inherited synonym list includes the term `token`; because such tokens are typically used for authentication rather than being credentials themselves, flows matched through this term should be read as authentication-related rather than as transmitted passwords.

J.3 Future Work

Using GUI agents for exploration. GUI agents are very popular recently, and they have been used to automatically test Android apps [109–112]. One promising direction is to replace `Explorer` with a GUI agent for auto-exploration. We have attempted to use GUI agents, but we encountered the following challenges. First, existing GUI agents are trained using 2D images and trajectories collected from smartphones, which is not adaptable to AVP scenarios. Second, existing GUI agents often rely on debugging tools such as ADB to control the device, which is not available on AVP. As a result, in this paper, we chose not to use GUI agents.

Other applications of AVP-Inspect. While this paper only presents one concrete application of using AVP-Inspect to detect privacy violations in network traffic, AVP-Inspect (especially `Controller` and `Explorer`) can be easily extended to other applications. For example, AVP-Inspect can be repositioned to perform fuzz testing on XR devices to detect functional bugs in XR applications; it can also be used to perform performance testing for XR applications. With the recent advancements of vision language models (VLMs), AVP-Inspect can also be used to collect UI data and study the semantics of XR UI contents using VLMs. AVP-Inspect can also be extend to other devices beyond AVP by replacing `Controller` with device-specific hardware. We will open-source AVP-Inspect to facilitate future research.

K Implementation of AVP-Inspect

We implement a prototype of AVP-Inspect, which consists of 286 lines of ESP32 code and about 3,000 lines of Python code. In this section, we present details of our implementation.

K.1 `Controller`

Simulating User Inputs. We implement the customized HID using an Arduino ESP32-S3 microcontroller with built-in Bluetooth Low Energy support. We develop custom firmware in C that implements the HID profile to emulate mouse and keyboard inputs. The firmware adapts the connection protocol to satisfy AVP's strict device pairing requirements [32] and optimizes input command queuing and execution timing to match AVP's response characteristics.

Pressing Physical Buttons. We select servo motors based on torque requirements, positioning accuracy, and response time. Through empirical measurement, we determine that approximately 350g of force is required to press each physical button on the AVP device. We therefore select servo motors capable of providing equivalent force. To ensure system stability during automated operation, we design a 3D-printed bracket that mounts the servo motors onto the side arms of the AVP headset. The bracket securely holds the servo motors without interfering with normal headset operation. We integrate the servo motors with the ESP32 through GPIO pins with proper power management circuitry, ensuring reliable button actuation.

K.2 `Explorer`

Screen Capturer. For screen capture, we configured AirPlay mirroring with optimal resolution and frame rate settings to project the AVP screen to a Macbook Pro. We use the MacBook Pro's built-in screen recording APIs with programmatic control. For evaluation purposes, we actually recorded the entire exploration process, and we take screenshots for every action by `Explorer` to process. Our frame processing pipeline goes from capture to format conversion to analysis preparation. We optimized latency to balance capture quality with real-time interaction requirements.

Virtual Cursor Handler. We locate the virtual cursor in screenshots using a two-step computer vision pipeline. First, we apply an HSV color filter with hue range [36, 86] to isolate the green cursor from the background, effectively removing noise from the scene. Second, we detect circular shapes using the *Hough Circle Transform* [113] with parameters `minRadius=20` and `maxRadius=100`. To handle variations in cursor size, we employ a progressive matching strategy that gradually relaxes the radius tolerance from 20% to 80%. A key challenge arises from AVP's pointer control system, which restricts cursor movement to the boundaries of the currently focused window panel. Since cursor movement is only permitted within the bounds of the focused window, we must ensure proper window focus before attempting UI interactions. We address this by leveraging AVP's head tracking feature to center the headset on the target window, which causes newly opened windows to appear at the center of the 3D scene and automatically receive focus. This approach enables unrestricted cursor movement within the focused window boundaries.

State Explorer. To construct UI states from screenshots, we deploy a pretrained OmniParserV2 model [37]. The model inference runs on a GPU server through a REST API wrapper, which the MacBook queries directly for each screenshot. We configure a detection confidence threshold of 0.05 to ensure accurate button detection in 3D environments. For state comparison, we extract text labels from interactive UI elements detected by OmniParser and compute pairwise text similarity using normalized Levenshtein distance with OCR-confusion correction. Two states are considered equivalent when (1) their button counts differ by less than 30%, and (2) the average best-match text similarity exceeds 0.7. A key challenge arises when the exploration becomes trapped in complex 3D scenes that require hand gestures and head movements for interaction. Our manual analysis reveals that these immersive scenes typically do not trigger network traffic. To prevent indefinite exploration of such states, we implement a timeout detector that monitors UI state changes. When the UI remains unchanged for more than one minute, we mark the current state as *invalid* and invoke the "Force Quit" feature from `Controller` to terminate the application. Subsequently, any exploration paths leading to *invalid* states are excluded from future iterations.

K.3 `Detector`

Network Traffic Capture. We capture network traffic from the AVP device using a Man-in-the-Middle (MITM) proxy approach. Specifically, we deploy `Mitmproxy` on the MacBook and configure the AVP device to connect through the MacBook's Wi-Fi hotspot. To enable HTTPS traffic decryption, we install `Mitmproxy`'s root certificate on the AVP device and mark it as fully trusted in visionOS's Certificate Trust Settings. This setup allows us to intercept, decrypt, and analyze all network communications originating from the AVP device during application testing.

L Detector Details

Matching mechanisms. We map extracted keys to taxonomy nodes using three mechanisms. *Synonym matching* uses the synonym lists of our taxonomy; these are inherited from VPVet and PoliCheck [18, 23], then augmented with the phrase groups extracted from network traffic. *Key-pattern matching* uses a manually curated dictionary of common SDK/API field names (e.g., `lat`/`lon`) and hand-written key regular expressions, extracted from the field names we observed across the collected traffic. *Value-pattern matching* uses manually defined regular expressions that recognize canonical value formats: email addresses, IP addresses, UUIDs, and latitude/longitude pairs.

First-party vs. third-party attribution. Following OVRSeen [17], we classify a flow as first-party when its destination domain matches the app's own domain, a domain declared in its privacy manifest, or the domain hosting its privacy policy. All remaining destinations default to third-party; we manually labelled the most frequently observed third-party entities and maintain that list as part of our artifact. Because a first-party backend hosted on an independently registered domain still defaults to third-party, our first-party counts are a lower bound.

Why the disclosures are not ground truth. Neither the privacy policy nor the privacy label can serve as ground truth: both are developer-provided disclosures and can be incomplete or incorrect, and ground truth can only be obtained from app behavior. Our work therefore compares observed runtime data collection against each privacy document separately. Pure policy–label inconsistency is orthogonal to our goal and can be studied with existing privacy-document analysis tools such as PolicyLint [22].

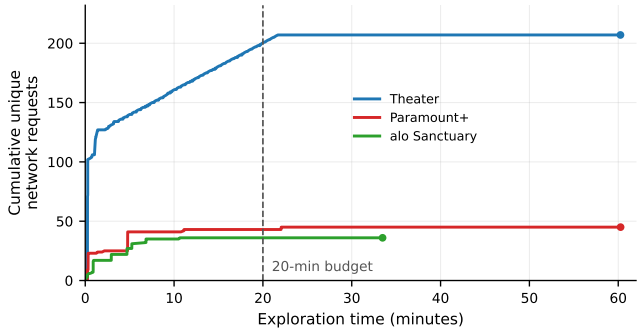


Fig. 20: Cumulative unique network requests during extended 60-minute sessions. All three apps saturate before the 20-minute budget expires.

M Long-running Exploration

To assess both how much AVP-Inspect captures and how stably it behaves over time, we re-ran 3 apps from different category with a 60-minute budget.

Coverage. In all three, the cumulative number of unique requests saturates well before the 20-minute budget expires (Fig. 20): 96.6%, 95.6%, and 100% of the requests seen over the full session are already present at 20 minutes.

Reliability. The plateau reflects saturation of the reachable surface rather than the explorer stalling. `Theater` keeps interacting throughout the remaining 39 minutes—243 clicks across 124 states over the full hour—while discovering no new request after minute 21, and a second 60-minute session issued 217 clicks across 33 states without interruption. Exploration therefore does not degrade over time.

N TLS Interception Coverage

We re-examined the raw `mitmproxy` captures for the full large-scale dataset to determine whether any traffic escaped interception.

Decryption. Every intercepted connection completed the TLS handshake against our proxy certificate. We observed no certificate rejections and no undecryptable payloads, and all 336 distinct destination hosts yielded successfully decrypted requests.

Errored flows. Of the 201 errored flows, 165 originate from transient failures on the local proxy hop and the remainder are ordinary mid-stream resets. None exhibits the behavior characteristic of certificate pinning, i.e., a handshake refused with a certificate error, or a connection established but immediately dropped before any request is sent.

What cannot be observed. The one behavior invisible on-path is silent evasion, in which an app declines to connect at all, or routes around the proxy, and therefore leaves no flow to analyze; detecting it would require observing runtime symptoms such as error dialogs or broken functionality. Our screen recordings show no such symptoms, but because silent evasion is undetectable by construction, we treat any flow we cannot decrypt as unknown rather than benign and present our measurements as a lower bound.

N.1 Case Studies

We present two representative case studies that illustrate the privacy disclosure violations discovered in our analysis. The first case demonstrates a severe *Contrary Disclosure* violation where an app explicitly claims to collect no data while actually collecting extensive user information. The second case illustrates a *Neglect Disclosure* violation in a Vision Pro-specific spatial computing application.

Contrary Disclosure (AirLauncher App Launcher). AirLauncher [114] is a utilities app designed exclusively for AVP. The app serves as a shortcut widget that allows users to quickly launch websites, apps, contacts, and shortcuts with a single pinch gesture. Critically, the app's App Store privacy label explicitly declares **"`Data Not Collected`"**, stating: *"The developer does not collect any data from this app."* However, our dynamic analysis reveals that the app transmits *9 distinct privacy-related data items* to RevenueCat's subscription management API, including *Device Identifier*, *App Fingerprint*, *Usage Behavior*, and *Performance Metrics*. RevenueCat, a third-party analytics SDK, can support cross-app attribution and identifier linkage, which increases the risk of user profiling and tracking and directly contradicts the app's stated disclosure.

Neglect Disclosure (Twin: Scans for Passthrough). Twin [115] is a spatial computing productivity app that exemplifies privacy concerns unique to the AVP ecosystem. The app utilize visionOS capabilities including photogrammetry scanning, passthrough visualization, and immersive environment placement to create *digital twins* of physical objects. Users scan real-world items (such as keyboards or desk decorations) using their iPhone or iPad, then place these 3D models in Vision Pro's virtual environments, enabling them to see physical objects while fully immersed. This use case raises heightened privacy concerns specific to Vision Pro: users scan their physical surroundings and synchronize data across multiple devices (iPhone/iPad for scanning, Vision Pro for viewing), creating a multi-device data trail that could reveal information about users' physical spaces and daily activities. The app's privacy label declares only *Data Not Linked to You* without specifying categories, leaving users unaware of the extent of data collection. Our analysis reveals *8 undisclosed data collection practices* spanning 4 privacy categories, including *Identifiers*, *Location*, *Usage Data* and *Diagnostics*, sending to two third-party SDKs: RevenueCat (subscription management) and Paywalls (UI presentation). The visionOS-specific headers in the traffic (`x-platform: visionOS` and `x-platform-version: Version 2.1`) explicitly identify the device as Apple Vision Pro, enabling platform-specific user profiling. This case highlights a growing Vision Pro privacy issue: as apps use spatial computing features, sensitive data collection rises, but privacy disclosures often overlook platform-specific risks and third-party SDK behavior.