Screen Spies: Semantic Usage Analysis and Triangulated Risk Assessment of Session Replay Services in Mobile Apps

Jiawei Guo
University at Buffalo, SUNY
Buffalo, NY, USA
jiaweigu@buffalo.edu

Zongze Li
University at Buffalo, SUNY
Buffalo, NY, USA
zongzeli@buffalo.edu

Haipeng Cai*
University at Buffalo, SUNY
Buffalo, NY, USA
haipengc@buffalo.edu

Abstract

Session Replay (SR) is a powerful tool that reconstructs a user's in-app session from fine-grained UI states and interaction events collected over time. Although SR is legitimately used by app developers, it may introduce significant security and privacy risks by recording sensitive on-screen interactions. While these risks have been studied on the web, their prevalence and nature in the mobile ecosystem remain largely unexplored. In this paper, we present the first large-scale, systematic dissection of SR in Android apps. To enable the study, we developed TapSpy, a novel auditing framework that synergistically combines program analysis with schema-guided, evidence-grounded Large Language Model (LLM) reasoning to perform a deep semantic audit of SR usage.

Using TapSpy, we conducted a measurement study on 47,855 high-impact Android apps. Our findings reveal a systemic failure in the mobile ecosystem's approach to SR security and privacy. Transparency is the first casualty: the vast majority of apps using SR fail to disclose this data collection in their Data Safety sections, creating a significant gap between their declared and actual privacy practices. We uncover a stark asymmetry in developer effort, where developers meticulously enrich session data with user identifiers and custom events for analytics but overwhelmingly rely on coarse, default masking for privacy. Consequently, risks rarely occur in isolation but instead form dangerous bundles, such as failure in redacting sensitive UI content (i.e., masking), logging sensitive information, linking it to a user's identity, and failing to disclose any of it. Our work provides the first large-scale empirical study of SR risks of these widespread issues in mobile apps and offers actionable insights for developers, SR vendors, and platform operators to mitigate the risks associated with this powerful technology. In line with responsible disclosure, we reported high-severity risks to the developers of affected apps.

CCS Concepts

• **Security and privacy → Software and application security**.

Keywords

Session replay, mobile apps, security and privacy risks, session replay usage specification (SRUS), triangulated auditing

*Haipeng Cai is the corresponding author.

CCS '26, The Hague, The Netherlands

ACM ISBN 979-8-4007-0050-7/23/11
https://doi.org/10.1145/3576915.xxxxxxx

ACM Reference Format:
Jiawei Guo, Zongze Li, and Haipeng Cai. 2026. Screen Spies: Semantic Usage Analysis and Triangulated Risk Assessment of Session Replay Services in Mobile Apps. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, The Netherlands.* ACM, New York, NY, USA, 25 pages. https://doi.org/10.1145/3576915.xxxxxxx

1 Introduction

Session Replay (SR) is a powerful technology that helps app developers reconstruct how users interact with an app by recording fine-grained UI states and interaction events over time in modern application (app) development, providing legitimate merits in debugging and user experience optimization. Notably, SR is *fundamentally different* from traditional mobile analytics and logging services. First, while conventional analytics collect coarse-grained telemetry (e.g., predefined events), SR services reconstruct entire user sessions with rich context, capturing detailed UI state and user inputs as they occur. Second, SR systems expose developer-configurable redaction semantics (e.g., redacting UI content by masking controls at global or element granularity), where misconfigurations can silently disable default protections. Third, the privacy impact is further amplified when replay data is associated with persistent identifiers (e.g., user accounts or profile attributes), enabling recorded interactions to be attributed to specific individuals. Fourth, SR is frequently framed as "analytics" or "debugging", despite recording content and interactions that users typically perceive as private. This *uniqueness* of SR also makes it *riskier than general third-party services* (as detailed in Appendices A and B).

In fact, risks induced by flawed SR service usage have already been found in high-profile real-world apps [1, 2]. In 2019, for example, several popular iOS apps were found to be recording entire user screens through SR services without proper disclosure [49]. Some of these apps employed the Glassbox SR service [6], resulting in sensitive data (e.g., payment information and private messages) being inadvertently captured and transmitted. Following media reports, Apple intervened and required developers to remove the offending code or face App Store removal, citing violations of policies mandating explicit user consent and visible indicators for recording [48]. In some cases, these SR-induced risks even triggered legal action [9].

However, extant relevant research has mainly focused on, hence being limited to, *SR usage in web platforms* [22, 23, 41, 54]. Apart from the platform coverage difference, the dominant methodology is runtime- or traffic-centric rather than code-based: prior web studies largely infer SR behavior from dynamic observations (e.g., monitoring script actions and outbound requests, detecting exfiltration, or instrumenting form interactions). This is because the end-to-end SR usage logic is encoded in large, minified first-party bundles

and opaque third-party scripts, and may further depend on server-side decisions that are not directly auditable from the client code alone [22, 41]. As a result, these studies typically do not (and often cannot) systematically examine the first-party integration logic that governs when SR is enabled, what data is unmasked or enriched, and under what guards these behaviors are triggered—precisely the logic that determines SR's effective security and privacy impact.

This leaves a *significant gap* in understanding the semantic behaviors and associated specific security and privacy risks of *SR usage in mobile apps*—mobile SR studies to date focus on limited features of SR in only a specific category of apps. This gap is particularly concerning given the intimate nature of mobile device usage and the potential for capturing sensitive user interactions that may not occur in web browsing contexts.

As it stands, there is no at-scale, systematized study that (i) *detects SR usage* in mobile apps, (ii) *analyzes SR semantics* governing data-capture and masking at code level, and (iii) *assesses real-world risks* (including, but not limited to, leakage and transparency inconsistencies) induced by SR when the usage is flawed.

In this paper, we aim to bridge this critical gap, presenting the first large-scale, systematic dissection of SR services in mobile apps to characterize their usage and assess their risks. Given the scale of the mobile ecosystem, manual analysis is clearly infeasible, necessitating an automated approach. However, achieving such automation is non-trivial and faces several fundamental challenges.

First (*Challenge 1*), there is a large semantic gap between the low-level code patterns of SR usage and the high-level, abstract security and privacy principles and policies. An effective, automated analysis must be able to infer the functional intent of code (e.g., "this code unmasks a login UI") and connect it to a reference standard (e.g., "do not record credentials"). Moreover, in line with their uniqueness, SR behaviors, which may have security and privacy implications or consequences, are composed of (a) the relevant set of SR APIs the app invokes, (b) the control-flow structure and data-flow context of (i.e., program dependencies among) those API calls, and (c) constraints or conditions (e.g., lifecycle callbacks, conditional branches, global configurations) that govern when and where those APIs are invoked. These SR behaviors, referred to as *compositional SR semantics*, model and determine what SR services actually record and how they annotate or link session recordings. Understanding these semantics from low-level code, which is not a property of any single SR call in isolation, nor point-to-point reachability [17] as in source-sink or taint flow [25], is inherently difficult.

Second (*Challenge 2*), the mobile software ecosystem is marked by immense heterogeneity: e.g., developers use SR APIs in varied and unpredictable ways, and the security and privacy documentation of both apps and SR services are often irregular, incomplete, or ambiguous. This wide variety makes it difficult to apply a single, static set of rules for risk assessment against various SR service vendors and apps. Then, the compositional nature of SR semantics exacerbates this heterogeneity-induced challenge. Different SR service vendors expose distinct API surfaces and defaults (e.g., masking models, consent interfaces, identity primitives), leading to diverse composition patterns in real apps. As a result, SR auditing must both recover composition-level usage context and normalize vendor-specific APIs into comparable semantic concepts.

The advent of large language models (LLMs) seems to bring hope for cracking the above challenges. Yet LLMs have their own inherent limitations (*Challenge 3*), including susceptibility to hallucination and practical constraints on context window size, making them struggle to offer reliable results and prohibitive for analyzing entire apps. Moreover, despite their general potential for code understanding, LLMs may not have contextual reasoning capabilities specific to compositional SR semantics and factual knowledge necessary for SR risk analysis (which involves various information sources such as app disclosures or vendor requirements).

Due to these challenges and uniqueness of SR services and their risks, existing relevant solutions fall short when applied to assess SR risks (as we elaborate in Appendix C and highlighted in Table 7).

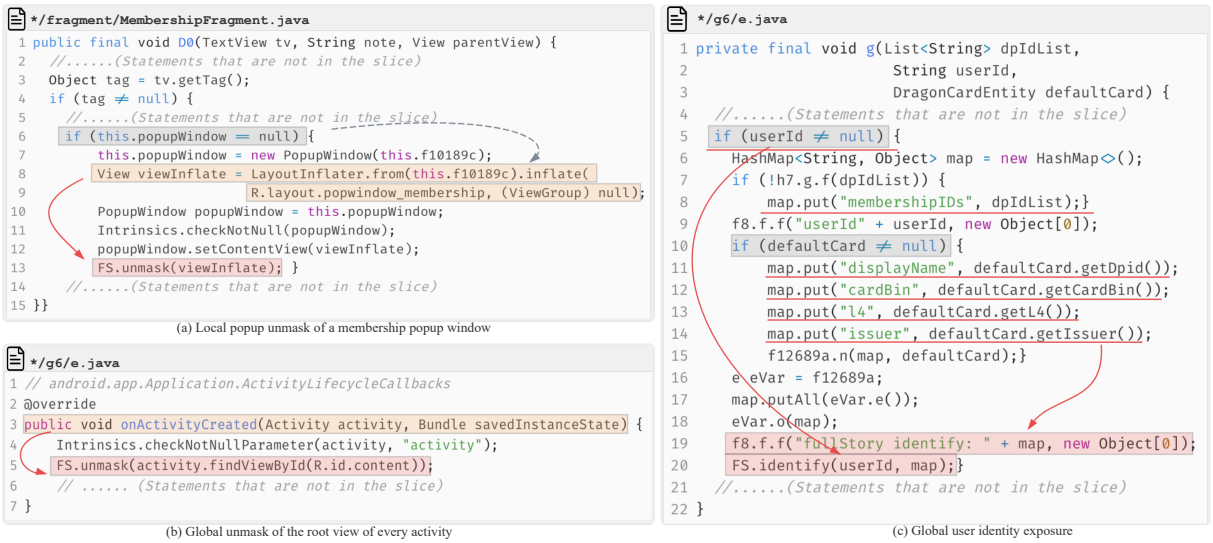
To address such challenges, our work is guided by several key insights. First, we leverage LLMs' reasoning capabilities to interpret the high-level functional semantics of low-level code and link them to natural-language security and privacy concepts defined in our framework, mitigating *Challenge 1*. Second, we introduce a structured approach, using program analysis to first abstract an app's complex codebase into a concise, code-level representation, referred to as the *Session Replay Usage Specification* (**SRUS**). We then develop a global schema for SRUS to normalize LLM's output as SRUS's natural-language representation. Meanwhile, we model SR vendors' policies (e.g., terms of use) and apps' disclosure/documentation (e.g., data-safety sections) as external references in a principled manner, while applying the same global schema to unify those references but refined specifically to each SR service and individual app. These together overcome *Challenge 2*, while further addressing *Challenge 1*. The code-level abstraction via SRUS dramatically reduces the code analysis scope for the LLMs, mitigating their context window constraints; we further introduce a triangulation-based approach to risk auditing, grounding LLM-based assessment with verifiable evidence based on a multi-faceted reference model, which counters model hallucination to overcome *Challenge 3*.

Based on these insights, we developed TapSpy, a novel, automated technique that analyzes compositional SR semantics hence assesses security and privacy risks of SR services as used in mobile apps. Using TapSpy, we then conducted a large-scale study of 47,855 real-world apps to provide the first empirical view of SR-induced risks in the mobile ecosystem. Our evaluation shows that TapSpy is highly effective, achieving 89.47% precision and 94.44% recall in identifying the risks, at low costs of <3 minutes and <$0.1 per app.

Our study reveals a systemic failure in the mobile ecosystem's approach to SR security and privacy. While transparency violations are widespread, we show that SR risks in mobile apps are driven less by the mere presence of recording and more by how developers *configure and enrich* SR sessions in first-party code. In particular, risky deployments frequently combine (i) weak or overridden masking, (ii) identity binding, and (iii) verbose enrichment (custom events or properties), creating replay artifacts that are both information-rich and readily attributable to a specific user.

Among others, our noteworthy findings are summarized below.

- We identify **551** Android apps in our dataset that integrate SR services, spanning 14 unique SR service vendors.
- Among these apps, **312** (**56.6%**) exhibit at least one risk, spanning *all* app functionality categories, and *all* risks in our taxonomy



(a) Local popup unmask of a membership popup window

(b) Global unmask of the root view of every activity

(c) Global user identity exposure

Figure 1: Examples of privacy-risking uses of the SR service by FullStory in a real-world app. (a) A membership popup window is explicitly unmasked, allowing its contents to be recorded. (b) User identity and payment-related attributes (user ID, membership IDs, card BIN, last four digits, issuer) are bundled into a map and sent to FullStory via FS.identify(). (c) A global lifecycle hook unmasks the root view of every activity, effectively disabling masking protections app-wide and exposing all UI contents.

appear in the wild. Most critically, we observe SR-specific implementation failures that can directly expose sensitive on-screen content and bind it to user identities:

(1) *(Un)Masking Misconfiguration* includes insufficient masking, overly permissive rules (e.g., wildcard-style unmasking), and even explicit unmasking of sensitive UI;
(2) *Logging of Sensitive Info* is largely driven by developers logging PII through enrichment APIs; and
(3) *PII Leakage in Identification* spans multiple identifier types, including email, user IDs, phone numbers, and names, enabling straightforward re-identification of replay sessions.

- For a small subset of apps, we instrument SR API entry points at runtime and have validated that the risky semantics detected statically do occur during the apps' executions. Further captured outbound traffic and observable end-to-end SR upload or transmission workflow show concrete evidence that SR artifacts are exported off-device to SR vendor infrastructure.
- To evaluate the real-world impact of our findings, we initiated a responsible disclosure process focusing on 64 apps that exhibited both technical vulnerabilities (e.g., masking misconfigurations) and a failure to disclose such practices in their privacy policies. To date, developers of 16 unique applications, spanning the Business, Social, Entertainment, Shopping, and Maps & Navigation categories, have formally confirmed our findings. These developer-validated cases substantiate the four primary risk categories identified by our tool: masking misconfigurations, identification leakage, enrichment logging, and consent bypass.

In summary, our work makes the following contributions:

- A novel auditing framework: we design and implement TapSpy, a new methodology that synergistically combines program analysis with schema-guided, evidence-grounded LLM reasoning to perform a deep semantic audit of SR usage in mobile apps (§3).
- A large-scale measurement: we present results of the first large-scale study of SR services in Android, quantifying their prevalence and revealing widespread risky implementation patterns and policy inconsistencies in real-world apps (§5).
- Actionable insights: we provide insights for multiple stakeholders, including developers, SR vendors, and platform operators, to help mitigate the security and privacy risks associated with this powerful and popular technology (§6).

2 Background and Motivation

2.1 Session Replay (SR)

Session replay refers to the practice of reconstructing a user's experience within an app—capturing interactions like taps, scrolls, view changes, and other UI events—to visualize how users navigate and interact with the app. It is widely valued in product development, UX research, and customer support as it provides qualitative context to quantitative analytics, enabling teams to understand why users behave in certain ways, not just what they do.

On mobile platforms, SR services typically do not rely on video-like screen capture. Instead, they operate by capturing snapshots of the app's view hierarchies—structures of UI elements (e.g., views, text boxes, images, input fields) that represent the app's state—and by logging interaction events over time. These snapshots, often represented as "frames," are transmitted (frequently as diffs) to the vendor's backend, where the UI is reconstructed for playback, recreating a user's journey with greater efficiency and control. These legitimate uses do not make SR inherently unsafe; rather, the risk depends on how developers configure SR in first-party code, as SR services often expose developer controls over what is captured and how sessions are contextualized. These controls include masking or unmasking UI content at different scopes (e.g., element, screen, or global), as well as enrichment and identification APIs that attach business events or user attributes to a replay. We provide more background details in Appendix A.

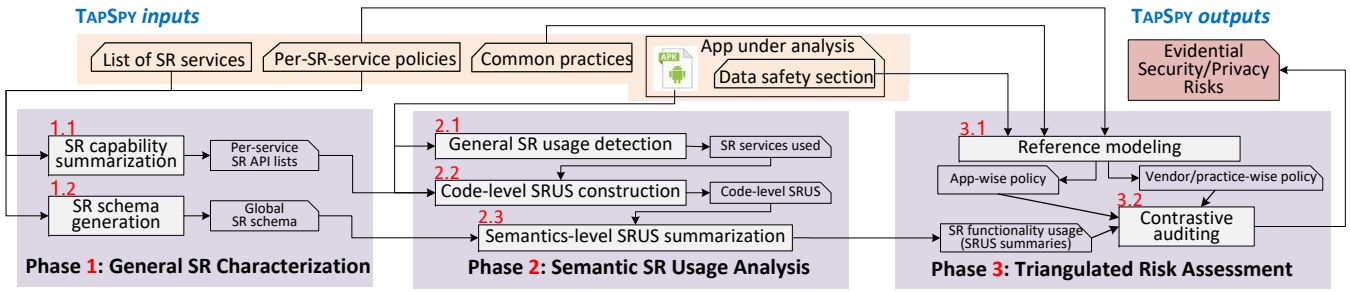


Figure 2: Overview of our TapSpy framework, including its inputs, three key working phases, and outputs. Starting from the app under analysis together with SR documentation, policies, common practices, and the app's Data Safety disclosures, TapSpy derives a unified SR schema (Phase 1), recovers app-specific SR usage semantics (Phase 2), and contrasts the recovered behavior against disclosure- and policy-based references to identify evidence-backed security and privacy risks (Phase 3).

2.2 Motivation

Despite its justifiable merits and prevalence, SR comes with security and privacy risks, which are often subtle yet severe. To illustrate, we consider a high-profile popular travel rewards app with over a million downloads and a stellar 4.8-star rating from over 60k users. It integrates a leading SR service, FullStory [8], and in doing so, implements several privacy-risking practices, as shown in Figure 1.

Figure 1(a) depicts a seemingly benign feature: a membership popup window. However, the code explicitly disables the default privacy protections for this UI element. After inflating the popup view (`viewInflate` at Line 10), the developer passes it to the `FS.unmask()` API (Line 15). This single call instructs the SR service to record the full UI content of the popup, which may include the user's name, membership status, or other personal details, defeating the SR service's default redaction.

While unmasking a single element is concerning, Figure 1(b) reveals a far more systemic issue. The app registers a global `ActivityLifecycleCallbacks` (Line 1) that triggers every time a new screen (Activity) is created. Within this callback, the code finds the root view of the new screen (Line 4) and calls `FS.unmask()` on it (Line 5). The consequence is profound: every single screen in the app is unmasked, effectively nullifying the SR service's privacy safeguards app-wide and exposing all on-screen content to recording.

Since most SR services can enrich the captured sessions with more information beyond capturing the user interface, SR can be used to exfiltrate sensitive data directly. In Figure 1(c), the app gathers user and payment-related attributes, including a user ID, membership IDs, and payment card details like the Card BIN and issuer (Lines 3, 12-15). This data is then bundled into a Map object, and as the data-flow arrows indicate, passed to `FS.identify()` (Line 23). This explicitly sends a detailed profile of the user and their payment instrument to a third-party analytics service. This kind of instrumentation may be intended to support richer analytics or replay segmentation. Notably, even if some values are visually masked in the UI, the risk here remains because the data is passed directly to the SR service through an identification API rather than captured from the rendered screen. Such behaviors are particularly concerning given that the app's Data Safety section claims "No data shared with third parties" and does not declare any data as being collected for "Analytics" purposes, putting its implementation in contradiction with its public disclosures.

The effective semantics in this example is the *composition* of three SR behaviors, each enabled by a specific API call under a specific guard or context to conduct (i) local unmasking, (ii) global unmasking, and (iii) identity or profile linkage. Together, these calls yield a composed semantics where the SR (a) records UI content that would otherwise be redacted (now including all screens due to the lifecycle-guarded global unmask), and (b) links those recordings to an identified user enriched with sensitive attributes—amplifying privacy risk beyond what any single call would suggest on its own.

Manually auditing thousands of apps for these nuanced issues is intractable. This motivates the need for an automated framework capable of operating at the scale of the app ecosystem. In the following sections, we present our technical design to address these very problems, as defined in our **threat model** (Appendix B).

3 TapSpy Design

We start with an overview, followed by details of each component.

3.1 Overview

Figure 2 gives a high-level overview of TapSpy highlighting its overall design. In addition to (1) the Android app (APK) under analysis, **TapSpy Inputs** also include (2) a *list of SR services* (names) targeted and their official API documentation and security and privacy policies (*per-SR-service policies*); (3) common security, privacy, and compliance requirements (*common practices*)—individual SR service vendors do not always include these common policies (e.g., "session recordings should not lead to PII leaks"); and (4) data collection and sharing disclosure of the given app (*data safety section*)—the Google Play Data Safety section of the app's listing.

With these inputs, in **Phase 1**, TapSpy builds a general understanding of SR capabilities from the target SR services' documentation, yielding a comprehensive *list of SR APIs* for each service. It then summarizes all security and privacy aspects relevant to SR capabilities, resulting in a *global schema* that defines a unified vocabulary of SR features and their associated privacy risks. The goal of this phase is to lay a *unified common foundation* for the consistent semantic analysis in **Phase 2** *across heterogeneous SR services and their diverse usages in various apps.*

Taking the global SR schema and per-service API lists, in **Phase 2**, TapSpy detects and characterizes actual SR usage in the given app. It first detects the presence of any SR libraries in this app. If the app package (APK) does not even include any SR library, TapSpy may end here. Otherwise, it proceeds by constructing the Session Replay Usage Specification (SRUS) of the app, a code-level

abstract representation that isolates the minimal, relevant program logic as a subgraph of each API caller's program dependence graph (PDG) [19], which includes each API callsite and its intraprocedural data- and control-flow context. The rationale is that such contextualized SR API calls collectively capture how the app uses SR functionalities of each included library. To enable high-level (semantic) understanding of SR usage, the SRUS is then converted to its natural-language form by an LLM via graph-based reasoning guided and output-regulated by the global schema. The goal is to allow for reasoning about risks induced by the SR usage at the same (i.e., natural-language) level (hence more accurately) in **Phase 3**.

Finally, in **Phase 3**, TapSpy audits the semantic SRUS summary for security and privacy risks and policy violations through a triangulation-based cross-referencing strategy. First, it constructs a reference model based on two data sources: the app-wise policy of the app (i.e., its Data Safety declarations) and the SR service vendor- and practice-wise policy (i.e., the vendor's terms of use or service and common security and privacy practices). Then, relevant violations are detected by triangulating the two data sources together with the app's SR functionality usage (i.e., natural-language/semantic SRUS summary)—any inconsistencies found by cross-referencing among the three entities signal risks and violations, with the contrasted references serving as evidence. Such audit reports of evidential risks constitute the main **TapSpy Outputs**.

3.2 General SR Characterization (Phase 1)

To build a foundational, vendor-agnostic understanding of SR capabilities, Phase 1 works in two steps below.

SR capability summarization (1.1). To understand the capabilities of SR, we construct a comprehensive corpus of public-facing APIs for each SR service. This is a prerequisite for understanding how app developers integrate and utilize these services. The primary challenge is that most SR services are commercial, closed-source products and often lack exhaustive, machine-readable documentation like Javadoc. Instead, their functionality is typically presented through tutorial-style developer guides.

We begin by manually curating an initial set of APIs from each SR service's official developer documentation. This is because these documents highlight the APIs that vendors intend for developers to use, effectively isolating the public API surface from internal implementation details. However, this documentation is often incomplete from a privacy auditing perspective. For instance, a tutorial might demonstrate how to tag a user session by showcasing a single API call (e.g., `setUserId()`), while omitting other semantically related methods within the same class (e.g., `setEmail()`, `setAddress()`, or `setName()`) that could also handle sensitive information such as PII.

To address this incompleteness, we employ a class-based expansion technique. For each API method curated from the documentation, we first identify its parent class. We then programmatically inspect this class to extract all other public methods it declares. This process systematically expands our initial seed set to include functionally adjacent APIs that are exposed to the developer but may not be explicitly mentioned in the tutorials.

SR schema generation (1.2). Next, to enable a consistent and scalable analysis across different apps and SR libraries, we abstract the discovered APIs and features into a global schema. This schema establishes a unified vocabulary of SR functionalities and their associated risks, serving as the canonical knowledge base for our entire auditing framework. To construct it, we first parse the developer documentation for each SR service and extract all distinct features. After manually reviewing these results for accuracy, we unionize SR features across all services to form a comprehensive catalog. This set was then filtered to retain only those features observable through static code analysis. Finally, we enriched the schema by manually analyzing and annotating the security and privacy risks of each feature, such as its capacity to handle PII or capture user input. The resulting global schema maps code-observable features to their potential risks, providing a robust foundation for our automated audit, as detailed in Appendix D Table 8 and Table 1.

3.3 Semantic Analysis of SR Usage (Phase 2)

Using the SR API lists from Phase 1, TapSpy then aims to analyze if and how the given app interacts with any of the targeted SR services. This is achieved by the three steps described below.

General SR usage detection (2.1). We first detect if the given app bundles a library with SR service. Thus, we extract all class definitions from the app APK and match against service-unique package namespaces. A positive match indicates the presence of an SR service usage in the app. Apparently, this process requires meaningful semantic information (e.g., method names, class structures) in the app. If the app is not obfuscated, the detection of explicit SR presence is efficient and accurate; otherwise, the information may be lost. Thus, we take a two-pronged approach, first checking explicit presence, followed by detecting implicit (obfuscated) usage and merging the results. The rationale is that an app may both explicitly *and* implicitly use SR services.

For the implicit presence checking, we leverage state-of-the-art (SOTA) obfuscation-resilient Android third party library (TPL) detection [50], obtaining a mapping between meaningful identifiers (e.g., class and method names) and their obfuscated counterparts. Apps may also dynamically load an SR service, followed by accessing SR APIs through reflection. Thus, we proceed with best-effort handling dynamic and reflective code loading by scanning uses of class-loading interfaces and, where feasible, resolving their target class names or bundled code paths; when these resolve to known SR namespaces or attributable code artifacts, we mark SR presence and carry the corresponding SR API callsites to the next step.

Code-level SRUS construction (2.2). With the curated API corpus, this step aims to characterize how these APIs are invoked. To that end, we introduce SRUS. For each user-defined method M that calls SR APIs, we formally define its SRUS as a subgraph of M's Program Dependence Graph (PDG). The rationale is that SRUS can capture the complete and minimal program logic that influences the app's interaction with the SR service.

The construction begins by identifying all user-defined methods that contain callsites to our target APIs. We iterate through every method in the app packages and match invoke instructions to our API corpus. When identifiers are obfuscated (e.g., renamed), we use the TPL detector's (same one as used in Step 2.1) internal mapping records to recover canonical class and method names and re-match. For reflective and dynamic invocations, we leverage the dynamically loaded SR library names resolved in Step 2.1 and further utilize a SOTA reflection resolver to identify the reflective call targets.

To focus exclusively on the app developer's integration choices, we exclude calls originating from the SR library's own packages or the standard Android framework. Using method `D0` in Figure 1(a) for example, once we found there is a library API call (Line 15) and the containing class name does not start with the SR library's package name, i.e., `com.fullstory` in this example, we label this method as a potential user-defined method. It is worth noting that we observed that for some general analytics service vendors, SR is an optional, distinct feature rather than the default behavior. For instance, Sentry [3], a widely used crash reporting and performance monitoring library, just recently introduced a stable SR feature. An app might integrate Sentry for its core features without ever enabling SR. To handle these cases, we manually identified the specific APIs required to **enable** SR functionality for each such SR service vendor. During our analysis, we only consider an app to be using SR if it explicitly invokes one of these **enablement** APIs. This ensures our study accurately targets apps that have actively opted into SR, not just including a library that offers it as a feature.

For each such user-defined method M, the SRUS subgraph is constructed by first performing a two-level union of backward slices to identify all relevant statements, and then *projecting* the method's full PDG to the union slice. Formally, the SRUS is constructed as:

$$SRUS(M) = PDG(M)\left[\bigcup_{c \in C_M} \bigcup_{a \in A_c} \text{Slice}(M, c, a)\right]$$

Where:

- $PDG(M)$ is the program dependence graph of method M.
- C_M is the set of all SR API callsites within M.
- A_c is the set of arguments for a given SR API callsite c.
- $\text{Slice}(M, c, a)$ is the static backward slice tracing the data and control dependencies that influence the value of an argument a at a callsite c.
- $G[V_{sub}]$ is an operator that yields the subgraph of a graph G that is induced by the vertex subset V_{sub}.

This definition ensures that the resulting SRUS is a concrete subgraph containing only the nodes (statements) and edges (dependencies) relevant to SR API interactions. Moreover, this code-level representation of SRUS corresponds to our notion of *compositional* SR semantics—as the PDG projection captures control flow structure, data flow dependencies, and constraints and conditions which together compose the SR semantics. To prepare the SRUS for downstream semantic analysis, we enrich this subgraph by numbering each statement and explicitly annotating the dependencies, for example, as $s_i \rightarrow s_j$. The resulting collection of these annotated PDG subgraphs constitutes the final SRUS for an app.

To illustrate, still consider Figure 1(a), where the API call `FS.unmask(viewInflate)` at line 15 is identified as the seed in the previous phase. The backward slice from this seed collects all statements that define or propagate its operands or guard its execution. Concretely, the slice retains the predicate input Line 5 `(tag = tv.getTag())`, the enclosing guards Line 6 `(if (tag != null))` and Line 8 `(if (this.popupWindow == null))`, the creation of the inflated view Line 10–11 and its propagation into the popup Line 14, the receiver chain through Line 9, 12, and 13, and the seed itself 15. Projecting the method's PDG onto these nodes yields a compact subgraph whose edges include control dependencies (6⟶15, 8⟶15), data dependencies along the argument chain (10⟶14⟶15), data dependencies along the receiver chain (9⟶12⟶13⟶14), and the flow into the control predicate (5⟶6). This SRUS thus captures what object is unmasked, how it flows to the API, which receiver instance is involved, and under which conditions the action occurs. If there are more seeds for this method, we repeat the same slicing procedure for each additional SR-API call and merge the resulting nodes and edges; their union constitutes the complete SRUS for the method.

It is important to note that this slicing and projection is performed on recompiled, Java-like source code rather than on a lower-level intermediate representation like Jimple. While Jimple is well-suited for traditional data-flow analysis, its three-address code format often introduces semantic redundancy by breaking down single high-level expressions into numerous simpler statements. This verbosity increases the analysis burden for both human auditors and the LLM. By operating on a more concise, source-level representation, we significantly reduce the number of input tokens required for the LLM, making our large-scale semantic audit more efficient and cost-effective.

Semantics-level SRUS summarization (2.3). With an app's SRUS constructed, the next step is to automatically generate its natural-language summary (i.e., natural-language representation of SRUS). This summary aims to answer what SR features an app uses and how it implements them. To achieve this at scale, we employ an LLM-based process guided by the schema defined previously.

The process begins with schema localization. With the global schema from the preceding step, we create a localized, service-specific schema by filtering for only those features relevant to the specific SR service present in the app. This crucial step focuses the LLM's analysis exclusively on the features that are actually available in the service being used. For example, in Figure 1(a), the app used FullStory, so the vendor-specific schema retained only the features supported by FullStory.

To generate the summary, we prompt the LLM to perform graph-based reasoning [28] on the SRUS. Instead of providing only the raw source code, we provide the entire SRUS structure: the numbered statements and the explicit data and control dependency edges (e.g., $s_i \rightarrow s_j$). The LLM is specifically instructed to utilize this graph to trace how data flows into API call arguments and to understand the conditions under which these APIs are invoked. This leverages the rich dependency information from the PDG to ground the LLM's interpretation in the underlying program logic, leading to more faithful and accurate summaries.

For each feature in the localized schema, the LLM generates a structured output containing: (1) Usage: a boolean determination of whether the app utilizes the feature; (2) Description: a natural language summary, derived from its graph-based reasoning, of how the feature is implemented; and (3) Evidence: the concrete arguments and key statement excerpts from the SRUS that support its conclusion. Using the same example in Figure 1, for Figures 1(a) and (b), the LLM will detect the feature presence of Element Level and Screen Level redaction control and output *true* for two features, with a natural language description—for example:"*The app invokes* `FS.unmask()` *in two distinct ways: (i) it conditionally unmasks a membership popup view when a specific UI or feature condition holds; and (ii) it registers* `ActivityLifecycleCallbacks` *and, during lifecycle events, repeatedly accesses each activity's root view and applies* `FS.unmask()`*, effectively performing systematic unmasking across screens.*" and the corresponding SRUS statement excerpts as

Table 1: Risk Taxonomy for SR Practices

Risk Category	Risk	Description
Disclosure Transparency (RC1)	Omitted Data Practices (R1)	The app's code behavior shows it collects or shares a specific type of data (e.g., location) via SR, but this practice is never mentioned in its Data Safety section.
Behavioral Implementation (RC2)	(Un)Masking Misconfiguration (R2)	The developer fails to mask enough sensitive information in a UI page or deliberately unmasks sensitive information.
	Logging of Sensitive Info (R3)	The developer misuses a Data Enrichment API (e.g., `logEvent`, `setCustomEvent`, or `log`) to pass sensitive, unencrypted information.
	PII Leakage in Identification (R4)	The app writes PII, credentials, or sensitive information when identifying the user's identity associated with the session.
	Erroneous Privacy Configuration (R5)	The developer calls the privacy APIs, but in the wrong order.
Consent & User Control (RC3)	Disregard User Choice (R6)	The user has explicitly denied consent, but the app or SR service proceeds to collect or share data anyway.
	Hard-Coded Consent (R7)	The developer hard-codes a "true" or "opt-in" value when calling an SR's privacy API, completely ignoring any preference the user might have selected in the UI.

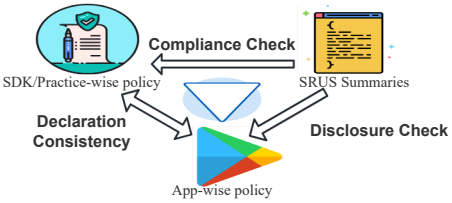


Figure 3: Reference modeling and triangulated risk auditing.

evidence. The LLM prompt template used is given in Figure 14 in Appendix I.

3.4 Triangulated Risk Assessment (Phase 3)

Leveraging the natural-language summaries from the preceding phase, the final step in our methodology is to audit these behaviors for potential privacy risks and policy violations. Our approach is a triangulated audit (Figure 3) that positions the app's observed behavior, the app's own disclosures, vendor and common privacy mandates [33, 36, 37, 45, 55, 59] as three vertices of a reference triangle. By examining the relationships across each edge, TapSpy systematically checks for mismatches and omissions. The final output of this process is a detailed privacy audit report for each app, identifying discrepancies between its actual implementation of SR and the policies it is expected to follow.

Reference modeling (3.1). To ground the LLM's reasoning, we build a triangulation model from three distinct sources. The rationale is two-fold. First, by providing the LLM with explicit reference policies, we mitigate the risk of hallucination and ensure its judgments are based on documented ground truths. Second, it allows us to account for developer transparency; as we detail in our threat model, a potentially risky data practice may not constitute a violation if it is clearly disclosed to users. These sources, shown as three vertices of Figure 3, represent distinct kinds of evidence:

- **SRUS Summaries (SR functionality usage):** The semantic summary of how the app actually uses SR features, as derived from the SRUS constructed in Phase 2.
- **App-wise policy:** The developer's self-declared data practices from the Google Play Data Safety section, specifically data claimed as collected or shared for "Analytics" purposes.
- **Vendor/practice-wise policy:** A unified model synthesized from SR service vendor terms of service and widely acknowledged privacy principles (e.g., GDPR, CCPA), capturing contractual requirements and common best practices.

Together these three sources form our reference model against which contrastive checks are performed. This triangulated view mitigates LLM hallucination by grounding analysis in verifiable inputs, and it accounts for both developer-facing obligations (vendor mandates) and user-facing transparency (Data Safety declarations).

Contrastive auditing (3.2). We leverage LLMs for auditing, which takes the above three kinds of evidence as its inputs.

Given these inputs, the LLM performs a structured contrastive analysis. Rather than assessing the code in isolation, it evaluates whether the observed behaviors are consistent with the expectations instantiated in the reference model. In practice, this involves checking each edge of Figure 3:

- **Disclosure Check (SRUS↔App-wise policy):** Does the app's SR usage involve data not disclosed in its Data Safety section?
- **Compliance Check (SRUS↔Vendor/practice-wise policy):** Does the app's SR usage violate vendor-imposed rules or widely acknowledged privacy practices?
- **Declaration Consistency (App-wise policy↔vendor/practice-wise policy):** Shown in the model for completeness, this edge represents the conceptual relationship between what developers disclose and what vendors or privacy principles would expect to be disclosed. TapSpy currently does not directly audit this edge, but it highlights potential for systematic under-reporting.

The output of contrastive auditing is a detailed audit report for the app. Each report enumerates the specific risks identified, including the violated policy dimension, a natural-language justification of why the behavior is risky, and the supporting SRUS evidence (API calls and arguments) that triggered the finding. This ensures that results are not only reproducible but also interpretable, grounding high-level assessment in concrete code-level facts. The LLM prompt template used is given in Appendix I Figure 15. Moreover, to enrich the audit report, we developed a taxonomy of SR risks as per the global SR schema. We first synthesize privacy guidelines from official vendor documentation, established issues from prior literature, and standard security best practices, before instantiating the taxonomy for specific SR services; we then use this taxonomy to associate each risk item in the report with an SR risk category.

Table 1 presents the *risk taxonomy* for session replay (SR) practices, which systematically organizes privacy and security concerns into three **R**isk **C**ategories (RC1–RC3) and seven distinct risks (R1–R7). Each risk highlights a concrete way in which SR implementations can misalign with transparency expectations, data protection principles, or user consent requirements.

- **Disclosure Transparency (RC1)** focuses on mismatches between declared and actual data practices. For example, an app may use SR service to capture sensitive data (e.g., location) without ever disclosing such practices in its Data Safety section (R1).
- **Behavioral Implementation Risks (RC2)** capture problematic developer practices when configuring or invoking SR APIs. These include insufficient or deliberately inverted masking (R2), logging of sensitive information through enrichment APIs (R3), leakage of personally identifiable information (PII) during session identification (R4), and misordered or incorrect invocation of privacy controls (R5).
- **Consent and User Control Risks (RC3)** address violations of user expectations around consent. Apps may disregard explicit denial of consent (R6), or worse, hard-code consent to always "true" regardless of user input (R7).

This taxonomy provides a systematic lens for identifying how SR practices can create privacy and compliance risks, complementing this global schema (as shown in Appendix D Table 8).

To illustrate, we revisit the example in Figure 1, following the output from Step 2.2. Using the NL summaries that indicate unmasking of (i) a membership popup and (ii) activity root views via lifecycle callbacks, TapSpy matches these behaviors to the reference model's notion of redaction control and default masking. Because both summaries describe actions that disable or override redaction (albeit at different scopes), the audit flags them as *(Un)Masking Misconfiguration (R2)* with scope-specific evidence: the report distinguishes the localized popup unmask in (a) from the systematic, lifecycle-driven root-view unmask in (b), while grounding both in the SRUS slices that expose the corresponding targets and guards.

These examples demonstrate the granularity of TapSpy's outputs: each risk is labeled with its taxonomy category, grounded in evidence from the SRUS (natural-language representation), and accompanied by a natural-language explanation of the privacy implication. This allows the results to serve both as a quantitative dataset (in our large-scale study) and as actionable evidence for developers and auditors.

Technique generalizability. The core analysis engine (SRUS graph extraction and guided-LLM reasoning via graph-based prompting) is decoupled from the pluggable SR-specific API schema. By swapping this schema, our method can be retargeted to perform a similar semantic audit on other mobile app infrastructure (e.g., analytics, ads, privacy-configurable SDKs). Our Phase-3 triangulated audit (code semantics ↔ vendor guidance ↔ app disclosure) is also fully general, providing a new way to find deep code-documentation mismatches in any Android app ecosystem.

4 Implementation and Evaluation

4.1 Implementation of TapSpy

We implemented TapSpy for Android apps, as further detailed in Appendix E. To support consistent analysis across SR services, we first constructed a global SR feature schema by manually curating SR capabilities from vendor documentation, unionizing them across services, and retaining only features observable in app code. We then manually derived the risk taxonomy by synthesizing privacy guidelines from official vendor documentation, established issues from prior literature, and standard security best practices. For the LLM-involved steps, we used Gemini-2.5-Pro [7].

For static analysis, we use Soot [46] for SR library presence detection and API call-site identification. We use LibScan [50] to recover meaningful library identifiers. To construct code-level SRUS, we decompile APKs with Jadx [43], resolve reflective calls with DroidRA [44], and use Joern [53] for dependence analysis and backward slicing.

4.2 Evaluation of TapSpy

In this section, we evaluate the performance of TapSpy in terms of both effectiveness and efficiency. In addition to the end-to-end violation detection results, we explicitly evaluate the two LLM-involved steps in our pipeline: (i) generating a natural-language (NL) representation of the SR usage from SRUS, and (ii) performing risk assessment grounded by our triangulated reference model.

4.2.1 Setup. **Reliability of the global schema and risk taxonomy.** To assess the global SR schema and the SR risk taxonomy curated through collective author decisions (i.e., which features and risks belong in the canonical vocabulary), we quantify inter-author agreement on inclusion and exclusion decisions. Concretely, we first form a closed candidate pool by taking the union of items proposed by documentation parsing and manual inspection; then multiple authors independently vote whether each candidate should be included. We compute Fleiss's Kappa on these pre-adjudication votes to measure agreement beyond chance, and resolve disagreements through discussion to produce the final schema (taxonomy).

To assess the effectiveness of TapSpy, we established a ground truth by manually inspecting a statistically significant subset of our dataset. From a population of 551 unique apps confirmed to use SR services, we randomly selected 84 app–service pairs. This sample size was determined using a 90% confidence level, a 5% margin of error, and an estimated population proportion of 10%.

Ground truth for SRUS NL representation. For each selected pair, annotators inspected the SR-relevant code paths and the SRUS slice, then produced a reference annotation of the key SR semantics that should appear in an accurate NL representation (e.g., whether SR is enabled or disabled, masking/unmasking scope, identity linkage, and activation guards such as consent/lifecycle/region logic). We then evaluated whether the generated NL representation from each approach correctly captured these annotated semantics, reporting precision/recall/F1. We also report the underlying ground-truth reliability using inter-annotator agreement, denoted by Fleiss Kappa (FK) score.

Ground truth for risk assessment. For the same 84 pairs, annotators determined the presence and type(s) of SR-related privacy/security risks by inspecting the corresponding code paths and vendor documentation where needed. This ground truth serves as the basis for measuring precision/recall/F1 of the final risk assessment stage, and we again report FK to quantify labeling reliability.

Baseline. To quantify the performance contribution of SRUS and the reference model (beyond the LLM itself), we implement a standalone LLM baseline that takes as input the same SR-invoked API callsites but without access to SRUS or our triangulated reference model. The baseline uses a fixed prompt to (i) summarize the SR behavior and (ii) judge whether the callsite indicates any privacy/security risk. We compute its precision/recall/F1 against the same ground truth.

To assess efficiency, we measured execution time, peak memory usage, and LLM API costs. We additionally break down runtime across the three primary phases of TapSpy to identify bottlenecks.

4.2.2 Results. For the two curated knowledge bases, SR feature schema and risk taxonomy, on independent include or exclude votes over the closed candidate pools, we obtain a Kappa of 0.83 for the SR schema and 0.91 for the risk taxonomy. These scores indicate high agreement beyond chance, suggesting that both the schema (what SR functionalities are code-observable and should be included) and the taxonomy (what risk types should be represented) are reliable foundations for the downstream auditing stages.

Table 2 reports results for both LLM-involved subtasks. For **SRUS NL representation**, TapSpy achieves high precision and recall (F1=0.941), while the baseline performs substantially worse (F1=0.436). This indicates that graph-guided prompting grounded in SRUS is critical for producing faithful, semantically complete descriptions of SR behavior, rather than relying on the LLM to infer missing context from isolated callsites.

For the **final risk assessment**, TapSpy again substantially outperforms the baseline (F1=0.919 vs. 0.250). This performance gap supports our central claim that TapSpy's accuracy comes from combining program-analysis-grounded SRUS with a triangulated reference model, which enables composition-aware reasoning about SR semantics (e.g., masking scope, identity linkage, and activation logic) that is difficult to recover from callsites alone. Finally, the FK values in Table 2 show that the manual ground truth for both subtasks is reliable, reducing the likelihood that the observed performance differences are artifacts of annotation noise.

In Table 3 the average execution time of TapSpy is less than 3 minutes per app with a reasonable memory load. The LLM API cost and token cost are also minimal, which are key advantages of our approach to be scalable in practice.

In Table 4 the most resource-intensive phase is Phase 2, which is responsible for the code-level construction and semantics-level summarization of the SRUS. The high resource usage in Phase 2 is expected, as it involves detailed static analysis to build a comprehensive dependency graph of the app's codebase. This process is crucial for providing the granular, grounded context needed to prevent LLM hallucination and ensure the accuracy of the final risk detection. Our analysis demonstrates that while the system requires significant computational resources during the SRUS construction phase, the overall cost of the approach is low, making it a viable solution for large-scale security audits.

5 Measurement Study

Using TapSpy, we conducted a large-scale measurement study, aiming to answer four research questions (RQs).

- **RQ1**: What is the prevalence and feature landscape of SR services in Android apps?
- **RQ2**: What are the common feature usage patterns of SR in Android apps?
- **RQ3**: To what extent do real-world SR implementations introduce privacy risks?
- **RQ4**: What do in-depth case studies reveal about real-world privacy risks introduced by SR?

5.1 Methodology

Dataset. To build our initial list of SR service providers, we consulted public software review platforms, starting with G2's "Best Session Replay Software" list [4]. For each candidate SR service vendor, we manually verified from its official website whether it supported the Android platform, as many are web-exclusive. This process yielded a set of potential SR services for our study. However, we encountered a practical limitation: several enterprise-focused SR service vendors (e.g., Glassbox [6], Quantum Metric [5]) do not provide public developer documentation or accessible AAR/JAR files. Access to their technical details typically requires direct contact with a sales team. As this prevented us from systematically mapping their API surface, we excluded these services from our analysis. For app collection, we scraped the Google Play Store to collect high-impact apps, defined as those with one million or more downloads. This resulted in a dataset of 47,855 apps.

Metrics. In our large-scale measurement study, we use the following metrics: (1) Prevalence and Usage: We measure the adoption of SR service libraries/enablement of SR support in the library and the usage of specific SR features (as defined in our schema in Table 8) as a percentage of the apps in our dataset. (2) Co-occurrence Patterns: To understand how features and risks are combined, we analyze their co-occurrence. We use the Jaccard similarity for pairs of items (SR features or risks) and Support (the proportion of apps containing a specific itemset) for identifying common combinations of three or more items. (3) Feature-Risk Association: To identify feature sets that are predictive of privacy risks, we use standard association rule mining metrics: Support, Confidence (the conditional probability of a risk occurring given a feature set), and Lift (the ratio of the observed confidence to the expected confidence).

Procedure. First, to answer RQ1, we began by characterizing the capabilities of 14 prominent SR services against our global feature schema. We then executed Phase 1 of TapSpy on our dataset of 47,855 apps to detect the presence of these services. We aggregated results to determine the market share of each service and analyzed their distribution across different Google Play Store categories.

Next, to address RQ2, we processed the apps identified in the previous step through the complete TapSpy pipeline. The semantic summaries generated in Phase 2 allowed us to identify which of the 16 features each app implemented. We then aggregated this data to calculate overall feature usage rates and performed the co-occurrence analysis to identify common feature bundles. We also categorized the specific arguments passed to APIs of features in FC2 and FC4 to understand developer implementation choices.

For RQ3, we used the final output from TapSpy's triangulated risk audit (Phase 3). This provided specific privacy risks present in each app. We calculated the prevalence of each risk across the dataset and performed a co-occurrence analysis to find which risks tend to appear together. Finally, we conducted an association analysis to identify the feature combinations most strongly correlated with specific privacy risks. To answer RQ4, we manually selected illustrative examples from the apps analyzed for in-depth analysis.

5.2 General SR Usage (RQ1)

To understand the capabilities and diversity of the SR ecosystem, we first analyze the feature sets of 14 prominent SR services; the full comparison is given in Appendix F.

Table 2: Effectiveness of TapSpy on two LLM-involved subtasks: (i) SRUS natural-language (NL) representation generation, and (ii) final risk assessment

Approach	Graph-guided prompting for SRUS NL representation				Risk assessment grounded by triangulated reference model			
	Ground Truth's Fleiss Kappa	P	R	F1	Ground Truth's Fleiss Kappa	P	R	F1
TapSpy	0.78	0.952	0.931	0.941	0.92	0.895	0.944	0.919
Baseline (LLM)	–	0.454	0.412	0.436	–	0.273	0.231	0.250

Table 3: Efficiency of TapSpy

Metric	Min	Max	Mean
Execution Time (s)	41.51	245.39	166.69
Peak Memory (MB)	2987.60	8730.09	3879.28
LLM API Tokens Used	3257	6802	4743
LLM API Cost ($)	0.03	0.07	0.06

Table 4: Average Execution Time per Phase of TapSpy

Phase	Sub-Phase	Time (s)
Phase 1	SR presence detection	6.14
Phase 2	Code-level SRUS	97.66
	Semantics-level SRUS	40.25
Phase 3	Contrastive detection	22.64

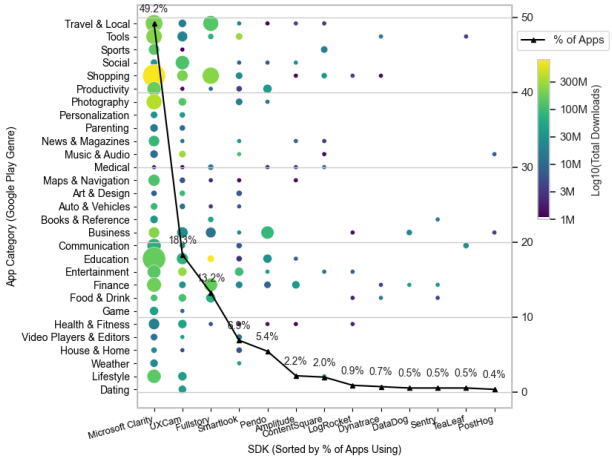


Figure 4: Distribution of SR service presence in studied apps.

Figure 4 shows that SR adoption is highly uneven across services and app categories. Each bubble represents a vendor–category pair, where size indicates how many apps in that category use the SR service and color indicates the relative scale of installs. For example, the Shopping–Clarity bubble is large and bright, meaning that many Shopping apps integrate Clarity and together account for substantial installs. These examples show that SR adoption can produce either broad exposure via many mid-sized apps or concentrated exposure through a few blockbuster apps. Weather or Dating categories show faint or no bubbles, indicating minimal use.

Our results also show that the SR adoption is concentrated in a handful of providers: Microsoft Clarity alone appears in nearly half of all apps, with UXCam and FullStory trailing far behind. Together with the bubble matrix, this shows that SR exposure is shaped both by a few services that dominate the ecosystem and by a few high-impact category–vendor pairings.

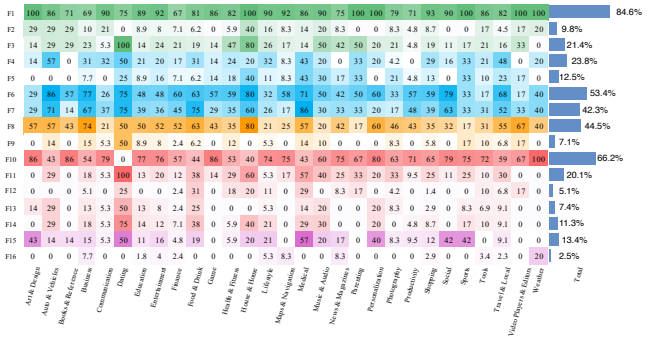


Figure 5: Distribution of SR Feature usage in the studied apps.

The category distribution is similarly skewed. Shopping, Education, and Travel & Local are the most prominent rows, each showing dense activity across multiple services. Finance also stands out: although its row is less dense, its bubbles are consistently large, implying that SR-using apps in this domain reach especially broad audiences despite smaller adoption.

This distinction between adoption count and audience reach is critical. The prevalence rate of 1.2% (551/47,855 apps) should be contextualized by the nature of these tools: unlike ubiquitous free analytics services, SR services are often paid, enterprise-grade products. Consequently, their impact is better captured by user exposure than raw app counts. The apps in our dataset using SR average over 15 million downloads each, creating a cumulative exposure footprint exceeding 8 billion downloads. Furthermore, SR adoption spans 29 distinct categories, penetrating sensitive domains such as Finance, Medical, and Business. Thus, while ecosystem-level prevalence is modest, the combination of high-volume user exposure and broad deployment in sensitive domains shows that SR has substantial real-world reach.

Concentration in a handful of SR services, and concentration of popularity in specific categories, imply that misconfigurations or weak defaults in just a few services can propagate widely, particularly in high-stakes app categories such as Finance.

5.3 Semantic SR analysis (RQ2)

In this RQ, we show how SR features are distributed in the apps analyzed. Figure 5 shows that some SR features are nearly universal, while others are rare. Initialization Setup (F1) dominates, reflecting that most apps explicitly configure SR on startup. Default Masking (F10) and User Identification (F6) are also widespread, showing frequent reliance on built-in privacy controls and common linkage of sessions to user profiles.

Across categories, the chart reveals systematic gaps. Some genres never employ particular features—for instance, F10 is absent in Dating. These omissions suggest that developers in some domains

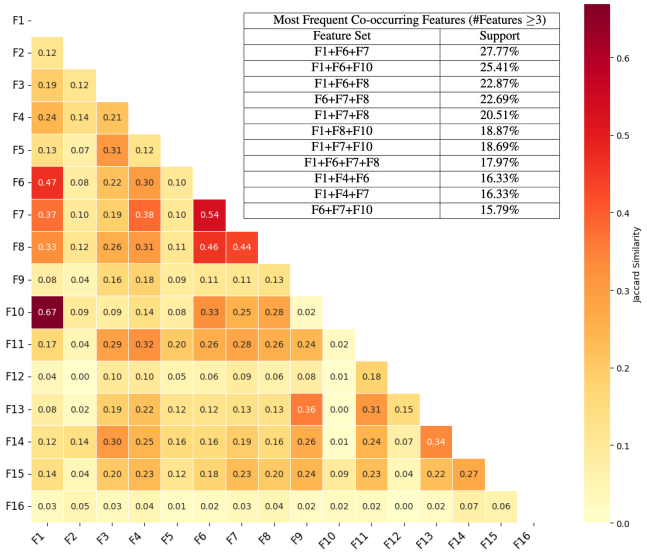
place less value on such controls, either because the perceived sensitivity is lower or the workflows are simpler.

Notably, F1 is not present in every app. This does not mean SR is running without initialization; rather, setup may occur in ways not visible to our code analysis, such as through configuration files outside the decompiled package or remote configuration from a vendor dashboard. In such cases, no explicit initialization call appears in the app code even though the service is active.

Developers often use mandatory or default features, while optional controls (e.g., consent and fine-grained masking) remain uncommon.

Figure 6 presents pairwise co-occurrence together with the most frequent ≥ 3-feature sets. The column for F1 is especially informative: its brightest cells align with F10, F6, Custom Event Tracking (F7), and Session Control (F8), indicating that once SR is initialized, developers commonly enable identification, log business events, and control session boundaries while leaving default masking in place. The table confirms this "business-context bundle": the top-ranked triad is F1+F6+F7, with support covering over a quarter of apps, showing that identification and event logging are frequently deployed together after initialization rather than in isolation.

The higher-order co-occurrence table clarifies patterns that a pairwise Jaccard view cannot capture. Several top triads include F1 and at least one of F6, F7, F8, demonstrating that value-generating features are activated in bundles. For example, F1+F6+F10 and F1+F6+F8 indicate that, alongside initialization, developers either keep default masking or add session controls; F6+F7+F8 shows that identification, event logging, and lifecycle control often co-occur even without F1 in the triple.



Most Frequent Co-occurring Features (#Features ≥3)	
Feature Set	Support
F1+F6+F7	27.77%
F1+F6+F10	25.41%
F1+F6+F8	22.87%
F6+F7+F8	22.69%
F1+F7+F8	20.51%
F1+F8+F10	18.87%
F1+F7+F10	18.69%
F1+F6+F7+F8	17.97%
F1+F4+F6	16.33%
F1+F4+F7	16.33%
F6+F7+F10	15.79%

Figure 6: Co-occurrence of SR Features in the studied apps.

Equally important is what does not cluster. Privacy add-ons that require extra engineering—granular masking at component/screen scope (F12/F13) and explicit consent (F16)—do not appear among the most common triads, despite being potent mitigations. Their absence in the top higher-order patterns, together with their weak pairwise associations in the heatmap, suggests that developers largely rely on defaults (F10) and seldom layer fine-grained protections or consent logic on top of identification and tracking.

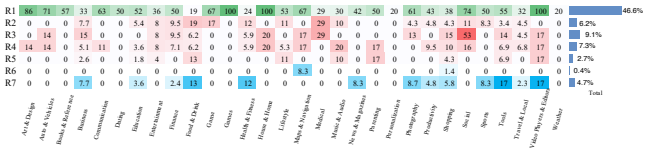


Figure 7: Distribution of SR Risks in the studied apps.

For the distribution of specific arguments passed to APIs of features in FC2 and FC4, we show them in the Appendix G.1.

SR deployments are built around tightly coupled "analytics bundles" (initialize → identify users → track custom events → manage sessions), while optional privacy controls remain peripheral.

5.4 SR Risk Assessment (RQ3)

In this section, we show how SR risks are distributed in analyzed apps. Out of 551 apps that have been found with SR usages, 312 apps (56.6%) are assessed to have at least one type of risks, covering all app functionality categories. This conditional risk is important for interpreting the significance of our findings. While only 0.65% of all apps in the full dataset (312/47,855) exhibit at least one SR-related risk, among SR-using apps the proportion rises to 56.6%. Combined with the popularity of these apps and their presence in sensitive domains, this indicates that the problem is not merely that SR exists, but that risky SR usage is common once SR is adopted. Additionally, all type of risks in the taxonomy are found in the apps analyzed. Figure 7 shows that Omitted Data Practices (R1) are by far the most prevalent risk, often exceeding half of apps in many categories and reaching near-universal presence in others.

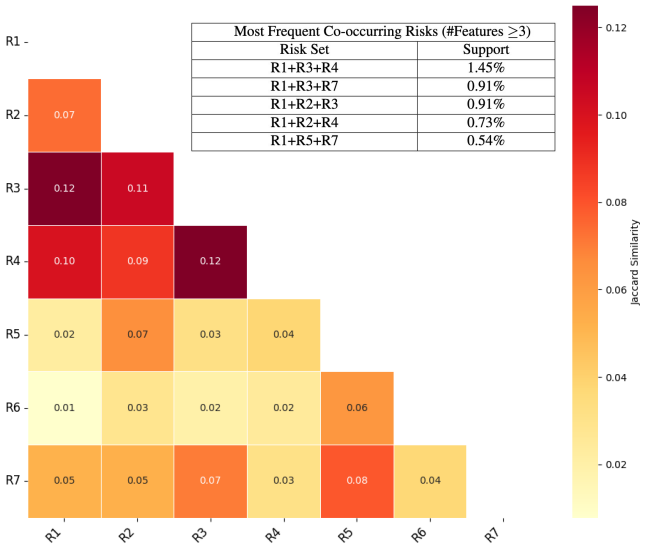
By contrast, Unmasking Misconfiguration (R2) and Logging of Sensitive Info (R3) are more scattered. R2 appears in only a handful of categories such as Business, Finance, or Shopping, suggesting that most apps do not attempt to override masking defaults but some do so in ways that introduce leakage. R3 shows a similarly uneven profile: while absent in most genres, it spikes in categories like Medical and Shopping, where custom event tracking is more used and sensitive details are more likely to be logged.

PII Leakage in Identification (R4) shows modest but widespread presence, with rates in the 5–20% range across categories like Lifestyle and Social. This suggests that while not universal, a meaningful minority of apps explicitly attach identifiers such as emails to SR sessions. Erroneous Privacy Configuration (R5) is rarer still, surfacing only sporadically in Business, Entertainment, and Travel & Local, reflecting configuration mistakes rather than deliberate design. The remaining risks are close to absent, such as Disregarding User Choice (R6) and Hard-Coded Consent (R7).

SR usage risks are prevalent. Transparency failures are systemic and dominate the landscape, while enrichment-driven risks appear more selectively in categories where developers actively track user actions or identifiers. Second, consent-related violations are rare, because they generally do not implement any explicit consent logic at all, leaving little opportunity to misconfigure it.

Figure 8 highlights how privacy risks tend to co-occur within the same app. The heatmap shows that R3 frequently overlaps with both R1 and R4, with Jaccard similarity exceeding 0.1 in each case.

This indicates that once developers start enriching sessions with sensitive fields, they are often simultaneously failing to disclose these practices or attaching identifiers, amplifying privacy risk.



Most Frequent Co-occurring Risks (#Features ≥3)	
Risk Set	Support
R1+R3+R4	1.45%
R1+R3+R7	0.91%
R1+R2+R3	0.91%
R1+R2+R4	0.73%
R1+R5+R7	0.54%

Figure 8: Co-occurrence of SR Risks in the app.

The higher-order table further confirms this combination. The most frequent triad is R1+R3+R4—particularly concerning since it both increases exposure and undermines transparency. Other recurring sets, such as R1+R3+R2/R7, show that omitted practices almost always accompany enrichment-driven risks. By contrast, risks like R5 or R6 rarely co-occur with others, appearing only in weaker combinations. This reflects that such misconfigurations are isolated mistakes rather than systematic patterns. For associations of feature co-occurrence and risks, we show them in the Appendix H.

For the distribution of specific arguments passed to API Calls identified with risks, we show them in the Appendix G.2.

The SR ecosystem is not just with random, independent risks. The most damaging patterns arise from bundles of enrichment and identification features that systematically co-occur with nondisclosure.

5.5 Case Studies (RQ4)

This section presents in-depth analyses of a small set of real-world apps identified by TapSpy. We organize the case studies into two parts. First, we show how our static analysis surfaces SR usage and risk-relevant implementation details from app code. Second, we validate a subset of these findings via dynamic instrumentation and network observation, showing that SR APIs are invoked with concrete runtime values and that SR session artifacts are subsequently exported off-device to SR vendor infrastructure.

5.5.1 Static Code Demonstration. We first present static, code-level evidence for representative apps to illustrate the spectrum of SR implementation practices. For each app, we trace the integration of SR service and highlight the specific SR APIs invoked with the surrounding application logic.

A Cascade of Risks in a Government App. The first case is an official digital wallet from a major Brazilian city, with 10M+ downloads and used to manage licenses and registrations. Given this

```
1 public final class SplashActivity extends AppCompatActivity {
2   @Override protected void onCreate(final Bundle savedInstanceState) {
3     DynatraceSessionReplay.setConfiguration (Dynatrace.applyUserPrivacyOptions(
4       UserPrivacyOptions.builder()
5         .withDataCollectionLevel(DataCollectionLevel.USER_BEHAVIOR)
6         .withCrashReportingOptedIn(true)
7         .withCrashReplayOptedIn(true)
8         .build()));
9     Configuration.INSTANCE.builder().withMaskingConfiguration(
10      new MaskingConfiguration.Custom().removeAllMaskedViews()).build();}}
11 ...
12 public final class LoginCheckActivity extends PushNotificationsActivity {
13   public final void redirect(String cpfLogado) {
14     Application application = getApplication();
15     Usuario user = new LoginDao(application).getUser();
16     Dynatrace.identifyUser(user.getCpf());}}
17 ...
18 public void onclick(SelecioneVeiculo CompradorAdapter.ServiceModel obj) {
19   DTXAction dTXActionEnterAction = Dynatrace.enterAction("CRV");
20   String numeroCrvVeiculo = obj.getVeiculo().getNumeroCrvVeiculo();
21   dTXActionEnterAction.reportValue("CRV", numeroCrvVeiculo);}
```

Figure 9: Severe privacy risks via SR in a government app.

high-trust context, the privacy failures are striking. TapSpy uncovered a chain of high-severity risks (Figure 9). The app hard-codes consent (Line 7), forcing session recording without user choice (R7). It then calls `removeAllMaskedViews()`, a global unmask that disables all redaction. Next, it leaks national identifiers by passing a citizen's CPF number (the Brazilian equivalent of a Social Security Number) to `identifyUser()` (Line 16), leak PII via identification(R4). Finally, it logs sensitive data by reporting each user's Vehicle Registration Certificate number (`numeroCrvVeiculo`, Line 21) as a custom event.

Exemplary Privacy-Aware Implementation. In contrast, we also observed privacy-preserving SR use. The Taco Bell app [11] illustrates how SR can support UX insights while safeguarding data (Figure 10). In its `ProfileFragment`, the birthday field is conditionally handled. When empty (Line 3), the app calls `FS.unmask()` (Line 6), allowing capture of the placeholder state for usability analysis. When populated, it first applies `FS.mask()` (Line 8) before inserting the user's birthdate (Line 9), ensuring PII is never recorded. Unlike the global unmasking in the government wallet, this design reflects "privacy by design": granular, context-aware masking embedded directly into app logic. It demonstrates that SR's analytic value and user privacy can be reconciled, offering a practical model for responsible development.

```
1 public class ProfileFragment extends BaseFragment implements f57,an3,nr6 {
2   @Override public void f1(String str) {
3     if (TextUtils.isEmpty(str)) {
4       Q5(this.C.J, getString(R.string.add_birthday),
5          R.color.tlp_banner_text, new View.OnClickListener() { ... });
6       FS.unmask(this.C.J); }
7     else {
8       FS.mask(this.C.J);
9       this.C.J.setText(str); }}}
```

Figure 10: A privacy-preserving SR usage.

Targeted Unmasking in a Medical Reference App. A popular medical reference app for millions of healthcare professionals demonstrates a deliberate, conditional SR strategy (Figure 11). SR activates only if the user is a US-based physician (Lines 3–4), suggesting a calculated attempt to apply different standards across regions. For this group, the app disables default masking via `setDefaultMasking(false)` (Line 5), then calls `unMask(EditText.class)` (Line 6), exposing all text input fields. It also hard-codes consent with `optIn(context)` (Line 8), preventing refusal.

```
public final void startSessionIfEnabledMain (Context context) {
  ...
  if (ProfileUtil.INSTANCE.isCountryUS()
   && ProfileUtil.INSTANCE.isPhysician()) {
    Contentsquare.setDefaultMasking (false);
    Contentsquare.unMask((Class<?>) EditText.class);
    Contentsquare.start(context);
    Contentsquare.optIn(context);
    Contentsquare.send("Launch screen");
    Settings.singleton(context).saveSetting(
      com.medscape.android.Constants.IS_CONTENTSQUARE_SDK_ENABLED, true);}}
```

Figure 11: Targeted, high-risk data collection logic via SR.

Table 5: App cases exercised with verified risks found in SRUS

Cases	Run-time Stack Trace	Risk Description	Risk
1	... → ...fragments.onViewCreated → com.fullstory.FS.addClass(**getView()**, **FS.UNMASK_CLASS**);	Unmasks bottom-sheet view, exposing on-screen text/inputs; undisclosed.	R1,R2
2	... → ...MainViewModel$1$1.emit → ...com.uxcam.UXCam.setUserProperty(user_email, **dummy@email.com**)	Links user email via setUserProperty (identity binding); undisclosed.	R1,R4
3	... → ...main.more.MoreFragment.initView → ...main.more.g.onClick → ... → ...UXCam.logEvent("Signout", **map**)	Logs "Signout" with user-profile PII key–value map, enriching the session; undisclosed.	R1,R3
4	... → ...application.App.onCreate → ...com.fullstory.FS.consent(**true**)	Hard-codes consent(true) at startup; undisclosed.	R1,R7

This illustrates a more complex privacy challenge. It does not stem from developer negligence, but a deliberate strategy to monitor a specific user base. The conditional logic, while potentially designed to navigate legal compliance, results in a two-tiered privacy system where one group of users is subjected to intense monitoring without their explicit consent. This example highlights how SR can be used for targeted surveillance and raises ethical questions about the practice of segmenting users for different levels of privacy.

5.5.2 Dynamic Behavior Validation. To strengthen the evidence of SR risks identified by TapSpy, we perform dynamic validation on a small subset of apps by executing them and instrumenting SR API entry points at runtime. Using Frida-based hooks [39], we record the arguments supplied to SR APIs revealed by TapSpy when they are invoked during interaction, and we compare these observed values against the risk-relevant arguments reported by TapSpy. We also capture the app's outbound network traffic to confirm that SR artifacts are uploaded to the SR vendor's infrastructure.

Table 5 shows part of the apps we exercised. We redacted app package names as the studied apps are closed-source, proprietary products. In Case 1, the app de-redacts a bottom-sheet during `onViewCreated` by calling `...`, exposing on-screen text/inputs. Case 2 links identity by setting `UXCam.setUserProperty(...)`, binding our input dummy email address to the session. Case 3 enriches the session timeline by logging a "Signout" event via `UXCam.logEvent(...)`, where *map* denotes a key–value dictionary of user-profile PII. Case 4 pre-enables recording at app startup by invoking `FS.consent(true)`.

Network traffic analysis. To further corroborate that SR data leaves the device, we captured outbound traffic from the Case 2 app during execution and observed the canonical UXCam upload workflow. The app first issues a verification request to `https://verify-{{region_code}}.uxcam.com/v4/verify`, after which the server returns a session identifier, capture settings (including `videoRecording=true`), and pre-signed upload parameters that authorize the client to upload two session artifacts: a `video.zip` and a `data.zip`. Immediately afterward, the device performs multipart uploads to UXCam-controlled storage endpoints (`https://prod-video-zip-{{region_code}}-1.uxcam-storage.com/` and `https://prod-data-zip-{{region_code}}-1.uxcam-storage.com/`), both returning HTTP 200 OK. The uploaded files are encrypted at the application layer (e.g., `video.mp4.aes` and `data.json.gz.aes`), so we do not inspect plaintext contents; nevertheless, the verify → upload sequence provides end-to-end evidence that SR session artifacts (screen recording and associated interaction/state data) are exported off-device to the SR vendor infrastructure. For more details, we have put the detailed scheme of each request and response in Appendix J.

Dynamic validation confirms that risk-relevant SR behaviors do occur at runtime. Network captures further show that these replay artifacts are subsequently uploaded off-device to SR vendor endpoints.

6 Discussion

6.1 Implications of Results

Our findings show actionable implications for developers, SR vendors, and platform operators, requiring coordinated measures to mitigate privacy risks of SR usage.

For Application Developers. The asymmetry between aggressive data enrichment and coarse, default masking indicates that apps often maximize analytics while investing little in privacy hygiene. To reduce leakage risk, developers could adopt a *deny-by-default* masking strategy: mask all UI elements at startup and explicitly unmask only the non-sensitive components needed for analysis, rather than relying on SDK defaults. Developers might also routinely audit the data sent through enrichment APIs to prevent PII from being logged as custom properties. Finally, the prevalence of R1 suggests that Data Safety disclosures may be maintained as living documents and reviewed as part of the release cycle to ensure SR collection is transparently reported to users [13, 31].

For SR Vendors. Although vendors often advertise SR as "private by default," their APIs can make it easy to disable protections globally. For example, broad unmasking primitives (e.g., unmasking an activity's root view) effectively provide a one-line way to nullify masking, the primary safeguard. Vendors can improve safety by adopting safer vendor defaults and discouraging risky patterns: deprecate or harden global unmasking in favor of APIs that require explicit, fine-grained unmasking of specific elements, introducing intentional friction against wholesale privacy disablement. Vendors could also build in privacy nudges and detection: emit high-priority debug warnings when sensitive UI (e.g., password fields) is unmasked or when identify/enrichment arguments match common PII patterns, and augment dashboards with backend monitoring that flags newly observed sensitive fields in uploaded streams.

For Platform Operators. The widespread failure to disclose SR in Data Safety labels shows that self-reporting alone is unreliable. Platforms should therefore incorporate SR detection into app review: scan submissions for SR integrations and cross-check them against declared data practices; if SR is present but not disclosed, require correction before approval. To strengthen informed consent, a potential direction for further study is to explore whether platform-supported indicators for active SR recording, analogous to camera or microphone indicators, can improve real-time user awareness beyond policy text alone. However, such mechanisms would require careful design and evaluation, since prior usable-privacy research shows that icons and indicators do not necessarily convey privacy choices effectively and may risk habituation [29].

Moreover, deploying such support would likely require substantial platform-level adoption, which can be challenging to realize in practice. Apple's 2019 intervention [48] was a necessary step, but it did not solve the broader problem of under-disclosure.

More broadly, these mitigations are amenable to tool-supported enforcement: automated analyses such as TapSpy can help developers catch risky SR configurations before release, help vendors identify risky integration patterns in deployed apps, and help platforms prioritize apps for disclosure review.

6.2 Limitations

Impact of obfuscation and implicit usage. To ensure broad coverage, TapSpy incorporates best-effort defenses against common malicious evasion/benign protection techniques. However, limitations remain regarding advanced obfuscation and dynamic behaviors [18]. Techniques such as control-flow flattening or aggressive string encryption may still hinder Phase 2, resulting in incomplete or overly complex SRUS slices that can degrade the fidelity of the LLM's semantic reasoning. Moreover, while we resolve reflection and dynamic code loading involving constants, targets derived from runtime-external sources (e.g., network configuration payloads or complex user inputs) remain invisible to our static pipeline. Consequently, apps relying on fully dynamic remote code loading to deploy SR logic may still evade detection. Accordingly, our measured prevalence should be interpreted as estimates over the subset of SR ecosystems that are observable through analyzable SDK artifacts and recoverable app code; they may therefore underapproximate the true prevalence of SR deployment and SR-related risks in the broader Android ecosystem.

Geographical differences. Although our case studies include apps from different countries and demonstrate that SR-related privacy risks arise in diverse geographic contexts, it is not sufficient to support claims about geographic disparity or the impact of local regulations. A rigorous treatment would require reliably attributing each app (and its SR configuration) to developer location and user-region targeting [27], and then controlling for confounders such as app category, market segment, and vendor choice. We leave a large-scale, geographically analysis to future work.

Threats to validity. Our analysis targets native Android apps; services that support integrated via cross-platform frameworks (e.g., React Native, Flutter) could not be analyzed, limiting generalizability beyond the native ecosystem.TapSpy also examines only compiled APKs, missing "out-of-band" configurations such as vendor dashboard settings or Gradle build scripts. These external mechanisms may under-report certain SR features or risks.

For SR services which we rule out due to lack of public docs and packages, we cannot find corresponding package or class names, therefore it is hard to know their exact distribution in our dataset or real world, which might impact our default list of SR services currently collected and considered in our measurement study.

7 Related Work

Analysis of Android Third-Party Libraries. Work on detecting third-party libraries (TPLs) in Android apps is extensive. LibRadar clusters and fingerprints packages at scale [35], while LibScout matches versioned signatures [14]. Later systems improve obfuscation resilience: LibID [56] remains accurate under renaming, shrinking, and control-flow randomization, and LibScan [50] uses cost-effective fingerprints with over-approximated class correspondences. Beyond detection, studies examine risks from outdated or vulnerable TPLs, showing pervasive outdatedness and challenges of updating mobile dependencies [20, 47].

Risks Due to Third-Party Libraries. Research on SDK/TPL risks spans multiple threads. Network- and app-level studies reveal widespread third-party tracking and policy gaps [12, 13, 15, 34, 40, 42, 52, 57–59]. More broadly, default SDK choices strongly influence privacy outcomes before customization [32]. For detailed comparisons with related research on Android app and TPL privacy, we have shown them in Table 6 and Table 7 in Appendix C.

Session Replay (SR) Studies. SR research has focused on the web. Embedded SR scripts have been shown to exfiltrate keystrokes, unsubmitted form data, and credentials [22, 23]. Large-scale studies reveal pre-submit exfiltration of emails and passwords [41], SR deployment on 3.6% of hospital sites leaking sensitive patient data [54], and frequent nondisclosure of SR use in privacy policies [30].

Our work connects to a growing body of mobile app privacy research showing persistent gaps between declared and actual data practices, as well as recurring failures of notice and consent mechanisms. As prior work shows that privacy failures often stem from misalignment among disclosures [12, 13, 15, 51, 52, 59], implemented behavior[21, 55, 57], and meaningful user choice [36–38]. Our findings place SR within this broader pattern, but through a distinct instrumentation layer where the resulting exposure can be especially consequential. In this sense, SR does not create an entirely new class of privacy failure; rather, it introduces a new technical vector through which known problems—policy–practice inconsistency, weak transparency, and ineffective consent—can be amplified: they may manifest in especially rich and sensitive form as SR captures fine-grained UI state, user interactions, and identity-linked contextual data.

Finally, instead of addressing malware behaviors [16, 26], our threat model assumes benign developers and benign apps, with risks arising from unintended SR usage, misconfiguration, or disclosure inconsistencies rather than purposeful malicious intent.

8 Conclusion

We presented the first large-scale study of mobile Session Replay (SR) usages. To enable this study, we also developed TapSpy, a novel framework that synergizes program analysis with large language models (LLMs) to realize semantic usage analysis of SR services and evidence-guided risk assessment in mobile apps. Using TapSpy, we audited over 47,000 Android apps and uncovered systemic privacy failures: most apps omit disclosures, developers enrich data for analytics while neglecting privacy, and risks cluster into dangerous bundles of sensitive logging, identification, and nondisclosure. These findings call for coordinated action from developers, vendors, and platforms to enforce transparency and safeguard users.

Acknowledgment

We thank reviewers for their constructive comments. This work was supported by the National Science Foundation (NSF) under Grant CCF-2505223 and Office of Naval Research (ONR) under Grant N000142512252.

References

[1] 2025. App Analysis: Air Canada. https://theappanalyst.com/aircanada.html.

[2] 2025. App Analytics SDKs Could Expose Sensitive Data. https://www.security.com/expert-perspectives/app-analytics-sdks-could-expose-sensitive-data

[3] 2025. Application Performance Monitoring & Error Tracking Software. https://sentry.io/welcome/.

[4] 2025. Best Session Replay Software: User Reviews from August 2025. https://www.g2.com/categories/session-replay.

[5] 2025. Digital Analytics Platform. https://www.quantummetric.com.

[6] 2025. Digital Customer Experience Analytics - Glassbox. https://www.glassbox.com/.

[7] 2025. Gemini 2.5 Pro. https://deepmind.google/models/gemini/pro/.

[8] 2025. Intro to Fullstory for Mobile Apps. https://help.fullstory.com/hc/en-us/articles/4414642861463-Intro-to-Fullstory-for-Mobile-Apps.

[9] 2025. Papa John's Sued for 'wiretap' Spying on Website Visitors. https://www.theregister.com/2022/10/06/papa_johns_spying_lawsuit/.

[10] 2025. Session Replay for Mobile. https://docs.sentry.io/product/explore/session-replay/mobile/.

[11] 2025. Taco Bell Fast Food & Delivery - Apps on Google Play. https://play.google.com/store/apps/details?id=com.tacobell.ordering&hl=en_US.

[12] Benjamin Andow, Samin Yaseer Mahmud, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Serge Egelman. 2020. Actions speak louder than words: {Entity-Sensitive} privacy policy and data flow analysis with {PoliCheck}. In *USENIX Security*. 985–1002.

[13] Ioannis Arkalakis, Michalis Diamantaris, Serafeim Moustakas, Sotiris Ioannidis, Jason Polakis, and Panagiotis Ilia. 2024. Abandon All Hope Ye Who Enter Here: A Dynamic, Longitudinal Investigation of Android's Data Safety Section. In *USENIX Security*. 5645–5662.

[14] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in android and its security applications. In *CCS*. 356–367.

[15] Duc Bui, Yuan Yao, Kang G Shin, Jong-Min Choi, and Junbum Shin. 2021. Consistency analysis of data-usage purposes in mobile apps. In *CCS*. 2824–2843.

[16] Haipeng Cai, Xiaoqin Fu, and Abdelwahab Hamou-Lhadj. 2020. A study of runtime behavioral evolution of benign versus malicious apps in android. *IST* 122 (2020), 106291.

[17] Haipeng Cai and John Jenkins. 2018. Leveraging historical versions of Android apps for efficient and precise taint analysis. In *MSR*. 265–269.

[18] Haipeng Cai and Barbara Ryder. 2020. A longitudinal study of application structure and behaviors in android. *TSE* 47, 12 (2020), 2934–2955.

[19] Haipeng Cai and Raul Santelices. 2015. Abstracting program dependencies using the method dependence graph. In *QRS*. 49–58.

[20] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me updated: An empirical study of third-party library updatability on android. In *CCS*. 2187–2200.

[21] Xiaolin Du, Zhemin Yang, Jiapeng Lin, Yinzhi Cao, and Min Yang. 2024. Withdrawing is believing? detecting inconsistencies between withdrawal choices and third-party data collections in mobile apps. In *S&P*. 735–751.

[22] Steven Englehardt, Gunes Acar, and Arvind Narayanan. 2017. No boundaries: Exfiltration of personal data by session-replay scripts. *Freedom to tinker* 15 (2017).

[23] Steve Englehardt, Gunes Acar, and Arvind Narayanan. 2018. No boundaries for credentials: New password leaks to Mixpanel and Session Replay Companies.

[24] Huan Gui, Ya Xu, Anmol Bhasin, and Jiawei Han. 2015. Network a/b testing: From sampling to estimation. In *WWW*. 399–409.

[25] Jiawei Guo and Haipeng Cai. 2026. EvoTaint: Incremental static taint analysis of evolving Android apps. *TOSEM* 35, 4 (2026), 1–46.

[26] Jiawei Guo, Xiaoqin Fu, Li Li, Tao Zhang, Mattia Fazzini, and Haipeng Cai. 2025. Characterizing installation-and run-time compatibility issues in Android benign apps and malware. *TOSEM* 35, 1 (2025), 1–44.

[27] Jiawei Guo, Yu Nong, Zhiqiang Lin, and Haipeng Cai. 2025. Code Speaks Louder: Exploring Security and Privacy Relevant Regional Variations in Mobile Applications. In *S&P*. 3952–3970.

[28] Jiawei Guo, Haoran Yang, and Haipeng Cai. 2025. VerLog: Enhancing release note generation for Android apps using large language models. *PACMSE* 2, ISSTA (2025), 1910–1932.

[29] Hana Habib, Yixin Zou, Yaxing Yao, Alessandro Acquisti, Lorrie Cranor, Joel Reidenberg, Norman Sadeh, and Florian Schaub. 2021. Toggles, dollar signs, and triangles: How to (in) effectively convey privacy choices with icons and link texts. In *CHI*. 1–25.

[30] Daichi Kajima and Hiroaki Kikuchi. 2023. Failure of Privacy Policy for Session Replay Services Used for Monitor Your Keystroke. In *NBiS*. Springer, 123–129.

[31] Rishabh Khandelwal, Asmit Nayak, Paul Chung, and Kassem Fawaz. 2024. Unpacking privacy labels: A measurement and developer perspective on google's data safety section. In *USENIX Security*. 2831–2848.

[32] Simon Koch, Manuel Karl, Robin Kirchner, Malte Wessels, Anne Paschke, and Martin Johns. 2025. The Impact of Default Mobile SDK Usage on Privacy and Data Protection. *PETS* (2025).

[33] Shuai Li, Zhemin Yang, Nan Hua, Peng Liu, Xiaohan Zhang, Guangliang Yang, and Min Yang. 2022. Collect Responsibly But Deliver Arbitrarily?: A Study on Cross-User Privacy Leakage in Mobile Apps. In *CCS*. 1887–1900.

[34] Xing Liu, Jiqiang Liu, Sencun Zhu, Wei Wang, and Xiangliang Zhang. 2019. Privacy risk analysis and mitigation of analytics libraries in the android ecosystem. *TMC* 19, 5 (2019), 1184–1199.

[35] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. Libradar: Fast and accurate detection of third-party libraries in android apps. In *ICSE'Companion*. 653–656.

[36] Trung Tin Nguyen, Michael Backes, Ninja Marnau, and Ben Stock. 2021. Share first, ask later (or never?) studying violations of {GDPR's} explicit consent in android apps. In *USENIX Security*. 3667–3684.

[37] Trung Tin Nguyen, Michael Backes, and Ben Stock. 2022. Freely Given Consent?: Studying Consent Notice of Third-Party Tracking and Its Violations of GDPR in Android Apps. In *CCS*. 2369–2383.

[38] Midas Nouwens, Janus Bager Kristensen, Kristjan Maalt, and Rolf Bagge. 2025. A Cross-Country Analysis of GDPR Cookie Banners and Flexible Methods For Scraping Them. In *CHI*. 1–28.

[39] Ole André Vadla Ravnås. 2025. Frida: A world-class dynamic instrumentation toolkit. https://frida.re/.

[40] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, and Christian Kreibich Phillipa Gill. [n. d.]. Apps, trackers, privacy, and regulators. In *NDSS*, Vol. 2018.

[41] Asuman Senol, Gunes Acar, Mathias Humbert, and Frederik Zuiderveen Borgesius. 2022. Leaky forms: A study of email and password exfiltration before form submission. In *USENIX Security*. 1813–1830.

[42] Yun Shen, Pierre-Antoine Vervier, and Gianluca Stringhini. 2021. Understanding Worldwide Private Information Collection on Android. arXiv:2102.12869 [cs]

[43] skylot. 2025. Skylot/Jadx.

[44] Xiaoyu Sun, Li Li, Tegawendé F Bissyandé, Jacques Klein, Damien Octeau, and John Grundy. 2021. Taming reflection: An essential step toward whole-program analysis of android apps. *TOSEM* 30, 3 (2021), 1–36.

[45] Zeya Tan and Wei Song. 2023. PTPDroid: Detecting Violated User Privacy Disclosures to Third-Parties of Android Apps. In *ICSE*. 473–485.

[46] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[47] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In *ICSME*. 35–45.

[48] Zack Whittaker. 2019. Apple Tells App Developers to Disclose or Remove Screen Recording Code.

[49] Zack Whittaker. 2019. Many Popular iPhone Apps Secretly Record Your Screen without Asking.

[50] Yafei Wu, Cong Sun, Dongrui Zeng, Gang Tan, Siqi Ma, and Peicheng Wang. 2023. {LibScan}: Towards more precise {Third-Party} library identification for Android applications. In *USENIX Security*. 3385–3402.

[51] Yue Xiao, Zhengyi Li, Yue Qin, Xiaolong Bai, Jiale Guan, Xiaojing Liao, and Luyi Xing. 2023. Lalaine: Measuring and characterizing {Non-Compliance} of apple privacy labels. In *USENIX Security*. 1091–1108.

[52] Yue Xiao, Chaoqi Zhang, Yue Qin, Fares Fahad S Alharbi, Luyi Xing, and Xiaojing Liao. 2024. Measuring Compliance Implications of Third-party Libraries' Privacy Label Disclosure Guidelines. In *CCS*. 1641–1655.

[53] Fabian Yamaguchi. 2022. A platform for robust analysis of C/C++ code. https://joern.readthedocs.io/en/latest/installation.html.

[54] Xiufen Yu, Nayanamana Samarasinghe, Mohammad Mannan, and Amr Youssef. 2022. Got sick and tracked: privacy analysis of hospital websites. In *EuroS&PW*. IEEE, 278–286.

[55] Chang Yue, Kai Chen, Zhixiu Guo, Jun Dai, Xiaoyan Sun, and Yi Yang. 2025. What's Done Is Not What's Claimed: Detecting and Interpreting Inconsistencies in App Behaviors. In *NDSS*.

[56] Jiexin Zhang, Alastair R Beresford, and Stephan A Kollmann. 2019. Libid: reliable identification of obfuscated third-party android libraries. In *ISSTA*. 55–65.

[57] Xueling Zhang, Xiaoyin Wang, Rocky Slavin, Travis Breaux, and Jianwei Niu. 2020. How does misconfiguration of analytic services compromise mobile privacy?. In *ICSE*. 1572–1583.

[58] Yifan Zhang, Zhaojie Hu, Xueqiang Wang, Yuhui Hong, Yuhong Nan, XiaoFeng Wang, Jiatao Cheng, and Luyi Xing. 2024. Navigating the Privacy Compliance Maze: Understanding Risks with {Privacy-Configurable} Mobile {SDKs}. In *USENIX Security*. 6543–6560.

[59] Kaifa Zhao, Xian Zhan, Le Yu, Shiyao Zhou, Hao Zhou, Xiapu Luo, Haoyu Wang, and Yepang Liu. 2023. Demystifying Privacy Policy of Third-Party Libraries in Mobile Apps. In *ICSE*. 1583–1595.

[60] Bolin Zhou, Jingzheng Wu, Xiang Ling, Tianyue Luo, and Jingkun Zhang. 2025. Version-level Third-Party Library Detection in Android Applications via Class Structural Similarity. In *EASE*. 604–615.

Ethics Considerations

Our research was designed and conducted with a commitment to ethical principles, guided by a stakeholder-based analysis as recommended by the CCS and informed by The Menlo Report. We considered the potential impacts on all relevant stakeholders throughout the research process and upon publication.

Stakeholder-Based Analysis. We identified the following key stakeholders who could be impacted by our research: mobile app users, app developers, SR vendors, platform operators (e.g., Google), the research community, and society at large.

For mobile app users, our primary ethical considerations were Beneficence and Respect for Persons. Our research procedure did not directly involve or harm users, as we only analyzed publicly available APKs without interacting with user data. While publishing our findings might cause user anxiety, we mitigate this by not disclosing new, exploitable vulnerabilities. The primary benefit to users is increased transparency about covert data collection, empowering them and their advocates to demand better privacy. Furthermore, our responsible disclosure process enables developers to fix high-severity issues, directly reducing potential harm.

Regarding app developers, we focused on Beneficence and Justice. We acknowledge that developers of apps with poor privacy practices might face reputational harm from our findings. To mitigate this, we followed a responsible disclosure process, reporting high-severity violations to affected developers via emails before publication, giving them an opportunity to remediate the issues. Our disclosure emails described the observed behavior at a high level, summarized the relevant risk categories, and invited developers to request additional evidence or clarification where needed. To date, developers of 16 applications have confirmed our findings. The response include acknowledgments, requesting more information, etc. We also observed early remediation patterns in several cases, including updates to privacy-policy or disclosure language in subsequent or current app releases. For apps whose code that show privacy concerns and presented in the paper, we also anonymize the app's package to avoid proprietary issues as they are closed-sourced profit-driven commercial apps. The benefit to the broader developer community is a clear articulation of common pitfalls and actionable guidance for using SR tools responsibly, which supports the creation of more trustworthy products.

Our analysis of SR vendors was guided by the principle of Beneficence. While vendors could experience reputational harm if their tools are shown to be frequently misused, we mitigate this by focusing our study on the implementation of their SRs, not on blaming them for the technology itself. We acknowledge the legitimate benefits of SR and believe our findings offer valuable feedback to vendors on how their APIs can be improved to encourage safer defaults and prevent common misconfigurations.

For platform operators like Google, our work aligns with Respect for Law and Public Interest. The findings might be seen as a critique of current platform policies, such as the Data Safety section. However, the benefit is that our research provides these operators with crucial empirical data on a significant transparency gap. We offer actionable suggestions that can help platforms strengthen their policy enforcement and better protect all users within their ecosystem.

Decision to Proceed and Publish. After weighing the potential harms against the significant benefits, we concluded that both conducting the research and publishing its findings are ethically justified. The decision to proceed was based on the use of publicly available data (APKs), which ensured no individual's privacy was infringed upon. The decision to publish stems from the importance of exposing a largely uncharacterized problem with significant privacy implications for millions of users. We believe the benefit of bringing this systemic issue to the attention of all stakeholders substantially outweighs the mitigated risk of reputational harm. By providing empirical evidence and actionable recommendations, our work serves the public interest and contributes to a safer mobile ecosystem. Our responsible disclosure process was a critical prerequisite, ensuring we took concrete steps to reduce harm before disseminating our findings.

Open Science

In compliance with the CCS open-science policy, we are making our research artifacts available to support the evaluation and reproducibility of our findings. The artifacts include the source code for our TapSpy framework, the scripts used for our large-scale measurement study, and the resulting dataset of detected risks and feature usage. These artifacts are essential for verifying our methodology and reproducing our results. The publicly accessible artifact can be found at https://doi.org/10.5281/zenodo.20600860.

Acknowledgements

In preparing this manuscript, we used generative AI tools in a limited, assistance-only capacity to polish and refine our writing. Specifically, these tools were used to improve clarity, grammar, wording, and overall readability of text drafted by the authors. We did not use generative AI to create original scientific content, generate novel results or interpretations, produce figures or data, or substantively draft sections of the paper; all intellectual contributions, analyses, and conclusions are the authors' own.

A More Background on Session Replay

Session Replay (SR) has become an increasingly prevalent capability adopted in modern software development, driven by legitimate needs in debugging and user experience optimization. By recording fine-grained user interactions—such as screenshots, screen recordings, touch events, and navigation flows—SR enables developers to retrospectively observe how users interact with an app, diagnose usability issues, and refine interface design. Moreover, SR can also mitigate biases inherent in self-reported feedback or A/B testing by providing direct observational evidence of user behavior [24].

Typically, session replay services do not rely on pixel-level video recordings. This approach provides several key advantages. The first one is efficiency. The data footprint is considerably lower than video or screenshot capture—often just tens of kilobytes per minute—since SR services encode drawing operations and hierarchy updates rather than raw image data. The second one is privacy control. Because SR services operate at the element level, they support granular masking or exclusion of sensitive UI components (e.g., password fields, credit-card inputs), enabling data hygiene and user privacy preservation. The third one is performance. Without

the overhead of video capture, SR services minimize impact on app performance and bandwidth, while still delivering rich replay fidelity.

For example, FullStory [8], an SR service vendor, captures view hierarchy snapshots and renders "frames" for playback. Data is "Private by Default," with built-in redaction and element masking. The SR service is lightweight and often scans hundreds of times per minute at low bandwidth. As another example, Sentry [10], a popular open-source analytics SDK, also released SR integration recently. It combines view hierarchy snapshots with periodic screenshots (about one per second) to build replay segments enriched with debugging context like breadcrumbs and backend error traces. Screenshots are compressed and integrated into video-like segments.

B Threat Model

Our research addresses the privacy risks originating from the integration and configuration of SR within mobile apps. The primary asset we aim to protect is the user's sensitive data and their reasonable expectation of privacy. This includes, but is not limited to, personally identifiable information (PII), credentials, financial details, private messages, and any other data visible or entered within the app's user interface.

Threat Actors. The primary threat actor we consider is the app developer, who, while not necessarily malicious, may induce privacy risks through negligence, a lack of awareness of the SR's full capabilities, or overly aggressive data collection practices for analytics. This actor has full control over the app's source code and is responsible for implementing the SR SDK, including configuring its data capture and masking rules. We also acknowledge the role of SR service vendors in designing APIs that, while powerful, may be easily misused.

Threats. We focus on application-level threats that arise from the developer's implementation choices. The main kinds of threats we investigate include:

- **Inadvertent Sensitive Data Exposure.** Developers may fail to correctly configure the SR's privacy controls. This can range from neglecting to mask individual sensitive fields (like password inputs) to disabling protections globally, as illustrated in our motivating example (Figure 1(b)). The consequence is the unintentional recording and exfiltration of sensitive on-screen data to third-party SR vendors.
- **Policy and Transparency Violations.** An app's de facto data collection practices via SR may directly contradict its de jure public disclosures. Previous studies have shown that this often happens in Play Store's Data Safety Section [13] [31]. As seen in our motivating example, an app may claim in its Data Safety section that it shares no data with third parties for analytics, while its implementation actively sends detailed user and device information to an SR service. This discrepancy erodes user trust and may constitute a compliance violation.
- **Lack of Meaningful Consent.** Users are often unaware that their every interaction within an app is being recorded in a video-like manner. Even if mentioned in a lengthy privacy policy, this granular level of monitoring can violate a user's reasonable expectation of privacy and the principle of informed consent, as users may not understand the full extent of the data being captured.

Scope and Assumptions. Our threat model is focused on the application-level usage and configuration of SR service. We aim to identify risky implementation patterns and policy violations that are introduced by the app developer. We assume the SR services themselves function as designed by their vendors and are not compromised with malicious code. Accordingly, our analysis does not aim to discover traditional software vulnerabilities (e.g., memory corruption) within the SR service library binaries themselves. Furthermore, the security of the SR vendors' backend infrastructure and the data transmission pipeline are considered out of scope. Our work targets the "first-mile" problem: identifying what sensitive data is being captured from the user's device in the first place.

C Comparison with Related Work

Table 6 situates our work among Android privacy studies along seven axes (attack surface, objective, technique, scope, and three capability columns). Only our study simultaneously (i) audits first-party code usage of SDK/TPLs, (ii) recovers sequence/guard semantics across APIs, and (iii) models SR-specific functional APIs. Prior works that do analyze code (e.g., privacy-config or entity-sensitive flow papers) either focus on flows/purposes or on privacy flags and provide, at best, partial guard reasoning tied to specific settings—none reconstruct the functional API compositions that create SR risk.

Additionally, Table 7 summarizes what transfers from prior technique families to SR and where they break. Policy/label consistency can detect omissions (our R1) once SR presence is known but is semantics-blind to API misuse. Runtime data-flow may catch exercised identify/log leaks yet misses SR's static, guarded control surface. Privacy-config analyses check consent/opt-out flags but not the compositional SR semantics that drive SR risk—underscoring the need for our SRUS + semantics-guided audit.

D Session Replay (SR) Schema

Our main solution to the challenge that SR service vendors are diverse is to develop a common, unified schema that defines SR capabilities (functionality features).

Table 8 presents the *global SR schema* used by TapSpy, which systematically categorizes session replay (SR) implementation practices into five high-level **F**eature **C**ategories (FC1–FC5). Each category is decomposed into concrete features (F1–F16) that capture how developers configure and control SR in mobile apps.

- **Initialization Configuration (FC1)** captures setup practices, including SR initialization, video quality settings, and rules for data upload.
- **Data Enrichment (FC2)** covers features that add semantic context to recordings, such as screen tagging, WebView capture, user identification, and custom event logging.
- **Session Lifecycle Control (FC3)** refers to developer logic for programmatically starting, stopping, or pausing SR sessions or video capture.

Table 6: Detailed Comparison with Related Research

Study	Targeted SDK/Behavior	What to detect/check	Technique	Scope	1st-Party Audit?	Seq./Guard Semantics?	SR API Semantics?
Ours	SR SDKs	**semantic misuse** of SR in 1st-party code.	Usage Specs+Semantic Audit	61k	Yes	Yes (General)	Yes
[57]	Analytics ASMs (e.g., 'setUserId')	PII in **runtime parameters** of ASM calls.	Dynamic Hooking	1k	No	No	No
[15]	App Behavior vs. Policy **Purpose**	policy ↔ behavior **purpose** consistency.	Network Traffic+NLP	23.1k	No	No	No
[12]	**Entity-sensitive** Flow vs. Policy	policy ↔ behavior **entity** consistency.	Network Flows+NLP	13.8k	No	No	No
[58]	**Privacy-Configurable APIs**	privacy flag usage & propagation.	Static+Dynamic Analysis	48k	Yes	Partial	No
[34]	Analytics SDKs (event collection)	**what** data is collected by libraries.	Static+Dynamic Analysis	300	No	No	No
[59]	TPL Data Access vs. Policies	TPL behavior ↔ TPL policy consistency.	Static Analysis+NLP	642	Yes	No	No
[21]	**Withdrawal (Opt-Out) UIs**	if UI **choice** is honored in code.	Static Taint Analysis (UI-to-API)	70k+	Yes	Partial	No
[45]	**Entity-sensitive** TPL Flows	flow ↔ policy consistency.	Static Taint Analysis+NLP	1k	Yes(Partial)	No	No

Table 7: Comparison Between Results of Applying Related Technique to Our Problem

Prior Technique Family	What Transfers to SR "As-Is"	Where It Breaks on SR services
Policy/Label Consistency (e.g., [12], [15], [59], [45])	Our **Phase 3** check: Can detect Data Safety omissions (our R1 risk) if the SR SDK's presence is known.	**Semantically blind to API misuse**. They have no model of SR's functional APIs. They cannot tell if non-disclosure (R1) is accompanied by high-risk misuse (e.g., global 'unmask' + 'identifyUser').
Runtime Data-Flow Analysis (e.g., [57], [34], [12])	**Partial R4 detection**. Might catch PII in an 'identifyUser' call if the exact UI path is exercised *and* the parameter value is a clear, unencrypted string.	**SR risk is primarily static.** The core risks—'unmask' scope, hardcoded 'identify' keys, guard logic—are in the *code*, not in a runtime data flow. Dynamic analysis is unreliable for triggering these specific, often-gated, code paths.
Privacy-Config SDK Analysis (e.g., [58], [21])	**Partial flag checking.** Can check if privacy-specific APIs (e.g., consent/opt-out toggles) are called correctly.	**Audits privacy flags, not functional APIs.** These tools check if 'setConsent(false)' is honored. They have no model for SR's functional-API misuse. SR risk comes from the **semantic misuse of functional APIs**('unmask', 'identifyUser'), not just privacy-flag misconfiguration.

- **Masking & Occlusion (FC4)** describes privacy-related controls at different granularities, ranging from default masking to element-, component-, screen-, or global-level rules.
- **Consent and User Control (FC5)** concerns user-facing safeguards, including opt-in/out functionality and explicit consent requirements prior to SR initialization.

This schema provides a structured basis for analyzing SR usage, ensuring both functionality and privacy implications are systematically examined.

E Implementation of TapSpy

We implemented our three-phase methodology in a tool called TapSpy. This section details the specific tools and design choices made in its construction.

Static Analysis Toolchain. We use Soot [46] for initial, whole-apk-level analysis to extract the complete list of classes for SR service library presence detection and to identify all call sites for our target SR APIs. For obfuscated apps, we use LibScan [50] to recover meaningful identifiers, as it balances effectiveness and efficiency—more recent tool [60] does not scale well for apps in our dataset, as it cannot finish for over 48 hours on randomly selected apps in our dataset. For the construction of the code-level SRUS, our primary workflow involves first decompiling the APK into Java-like source code using Jadx [43]. For handling reflective calls, we used DroidRA [44], as it is the state-of-the-art available and usable tool. We then use Joern [53] to perform program dependence analysis and backward slicing on this human-readable source code. In cases where Jadx fails to decompile a specific method or where class naming conflicts arise, we have a fallback mechanism. To ensure complete coverage, we revert to using Soot to perform the PDG construction and program slicing directly on the Jimple intermediate representation for those specific methods.

Large Language Model. For all semantic analysis tasks—including schema generation, SRUS summarization, and the final triangulated audit—we utilize Gemini-2.5-pro [7]. We selected this model for its advanced reasoning capabilities, by the time our technique was implemented. During our interactive prompting trials, we also observed a particularly beneficial behavior: when provided with an app's package name, the model actively attempts to retrieve public information about the app, such as its name and purpose, enriching its analysis with valuable real-world context.

Extensibility. To support a new service S, no methodology changes are needed—only a new spec and light adapters. Concretely: (i) add a spec with namespaces/fingerprints (and optional endpoint hints) for Phase-1 discovery; (ii) define S's functional-API schema of S-specific equivalents plus argument mappers; (iii) register vendor-guidance anchors and policy/DSS field mappings for Phase-3; (iv) include any reflective/dynamic resolution hints so Step 1.1 can normalize targets. Once the spec is loaded, the pipeline reuses the same PDG slicing → SRUS construction → schema-guided LLM summaries → triangulated audit.

F SR Feature Usage Characterization

Table 9 provides a comprehensive comparison of these SR services against the 16 code-observable features defined in our global schema (Table 8) , which are grouped into five high-level categories (FC1-FC5).

This table can be interpreted as follows: each row corresponds to an SDK, and each column represents a specific feature. A filled circle (●) indicates that the SDK supports the feature, while an empty circle (○) means it does not. For example, examining the UXCam row, we see it supports nearly all features, including F11 (Element Level Masking), but lacks support for F16 (Explicit User Consent). The "Coverage" column aggregates these findings, showing that UXCam supports 94% of the features we analyzed. Conversely, the

Table 8: Global SR Schema used by TapSpy

Feature Category	Feature	Description
Initialization Configuration (FC1)	Initialization Setup (F1)	The developer has explicitly initialized the SR service with an API key or startup call
	Video Quality (F2)	The developer has configured video resolution or framerate settings
	Data Upload (F3)	The code includes rules for when (e.g., Wi-Fi only) or where (e.g. US) session data is uploaded
Data Enrichment (FC2)	Screen Tagging (F4)	The developer is naming screens to provide context to the visual recordings.
	WebView Capture (F5)	The developer has enabled the capture and replay of content within WebViews
	User Identification (F6)	The developer sets a user identifier and/or custom properties (e.g., subscription tier, name)
	Custom Event Tracking (F7)	The developer is logging specific in-app events to add business context to session replays
Session Lifecycle Control (FC3)	Session Control (F8)	The app includes code to programmatically stop, resume, or restart the entire session recording.
	Video Control (F9)	The app specifically pauses or resumes video capture *within* an active session
Masking & Occlusion (FC4)	Default Masking (F10)	No related API called or applying default masking
	Element Level(F11)	(Un)Masking specific UI elements (e.g., a password field).
	Component Level (F12)	(Un)Masking logical groups of elements (e.g., an address form).
	Screen Level (F13)	(Un)Masking entire screens (e.g., a checkout or payment page).
	Global/Pattern-Based (F14)	(Un)Masking all elements matching a certain type (e.g., all input fields) or all the content in the app
Consent User Control (FC5)	User Opt-In/Out (F15)	The developer has implemented logic to enable or disable recording based on user choice
	Explicit User Consent (F16)	The app contains logic that requires a user to provide consent *before* the SR service is initialized

Table 9: Feature Comparison of Session Replay (SR) services

Vendors	FC1			FC2				FC3		FC4					FC5		Cov.
	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	
Clarity	●	○	●	○	●	●	●	●	○	●	○	○	●	○	○	○	56%
UXCam	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	94%
FullStory	●	○	○	●	●	●	●	●	○	●	●	○	○	○	●	●	69%
Smartlook	●	●	●	●	●	●	●	●	●	●	●	○	●	●	○	○	88%
Pendo	●	○	○	●	○	●	●	●	○	●	●	●	○	●	○	○	63%
Amplitude	●	○	○	●	●	●	●	●	○	●	●	○	●	●	●	○	75%
ContentSquare	●	●	●	●	●	●	●	●	○	●	●	●	●	●	●	●	94%
Datadog	●	○	○	○	●	●	●	○	●	●	●	○	○	●	●	●	69%
LogRocket	●	○	●	●	●	●	●	●	○	●	●	○	●	○	○	○	69%
Dynatrace	●	○	●	●	●	●	●	●	○	●	●	○	○	●	●	●	75%
Sentry	●	●	○	○	●	●	●	●	○	●	●	●	●	●	○	○	75%
InstaBug	●	○	●	○	○	●	●	●	○	●	●	○	●	●	○	○	63%
TeaLeaf	●	○	○	○	●	○	●	●	○	●	●	○	○	○	○	○	44%
PostHog	●	○	○	●	○	●	●	●	○	●	○	○	●	○	●	○	56%
Support	100%	29%	50%	64%	79%	93%	100%	93%	21%	100%	86%	36%	71%	71%	57%	36%	**71%**

"Feature Support" row at the bottom indicates the prevalence of each feature across all service vendors; for instance, F7 (Custom Event Tracking) is a universally supported feature (100%).

Our analysis reveals a significant variance in the capabilities offered by different SR service vendors. On the high end, UXCam and ContentSquare emerge as the most comprehensive solutions, each supporting 94% of the analyzed features. This suggests a mature feature set designed to cover a wide array of analytics and debugging needs. In contrast, TeaLeaf offers the most focused or minimalist feature set, with a coverage of only 44%. This may indicate a design philosophy centered on core replay functionality rather than an all-encompassing data analytics suite.

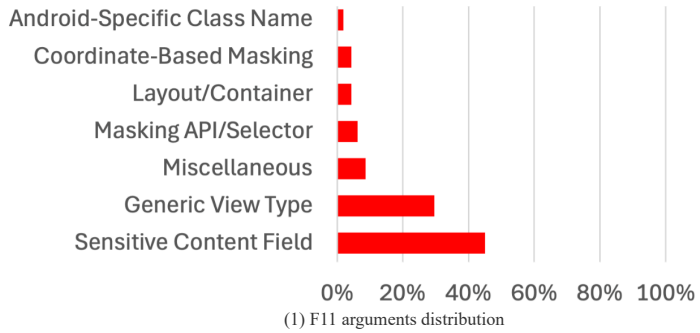
The prevalence of features across the ecosystem highlights what are considered core versus niche functionalities. Foundational features are universally available, with 100% of vendors supporting Initialization Setup (F1), Custom Event Tracking (F7), and Default Masking (F10). This establishes a baseline for modern SR tools: they must be configurable, able to integrate with application-specific business logic, and include basic privacy protections by default. However, support for more advanced or nuanced features is less consistent. For instance, fine-grained control over the recording itself, such as adjusting Video Quality (F2) or pausing only the visual capture via Video Control (F9), are among the rarest features, supported by only 29% and 21% of vendors, respectively. This suggests that while all services can record sessions, granular control over the recording process is a premium or less-demanded capability. From a privacy perspective, while element-level masking is common (F11 at 86%), explicit, vendor-enforced user consent mechanisms (F16) are surprisingly uncommon (36%), placing a greater onus on developers to implement their own consent flows.

G More details on SR Usage Characteristics

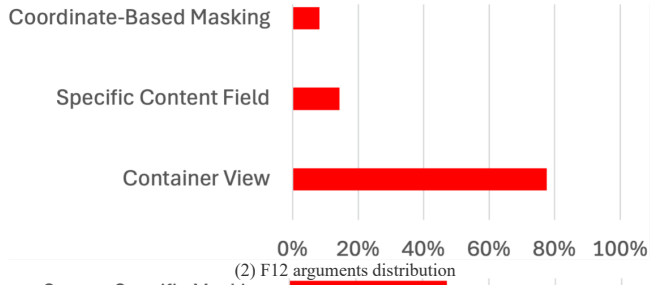
G.1 Arguments Distribution for SR API Call

Figure 12 visualizes how developers pass arguments into SR APIs, split between masking and data enrichment. On the masking side (left), developers overwhelmingly prefer broad, structural arguments rather than fine-grained ones. At the component level, container views dominate (over three-quarters of all arguments), indicating that masking is often applied wholesale to entire UI groups. At the element level, the largest share is sensitive content fields (≈45%), showing that when developers mask at finer granularity, they mostly focus on obvious inputs like passwords or credit card numbers. By contrast, categories such as layout/container references, coordinate-based masking, or Android-specific class names account for only a few percent of cases, revealing how rarely developers invest in low-level or maintenance-heavy masking rules. A similar pattern appears at the global/pattern-based level: view-class masking (≈49%) dominates, while more advanced toggles or niche categories such as masking levels or boolean flags remain marginal. At the screen level, masking is again concentrated into broad switches: either boolean/view-based rules (≈53%) or screen-specific toggles (≈47%), showing that coarse-grained controls are favored over nuanced configurations. In short, developers rely on broad, easily deployed arguments to achieve "good enough" masking, while specialized options remain underused.

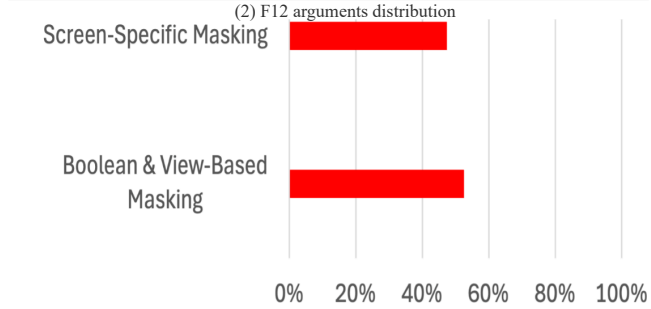
On the data enrichment (right) side, the pattern is reversed: arguments are far more diverse, and developers show deliberate effort to capture business context. In event tracking, the most common arguments are user actions & content (≈30%) and app/device information (≈16%), followed by event properties and event names. This indicates that replay is tightly tied to analytic pipelines that reconstruct user journeys with semantic detail. In screen tagging, developers frequently supply contextual or specific names (≈33%)



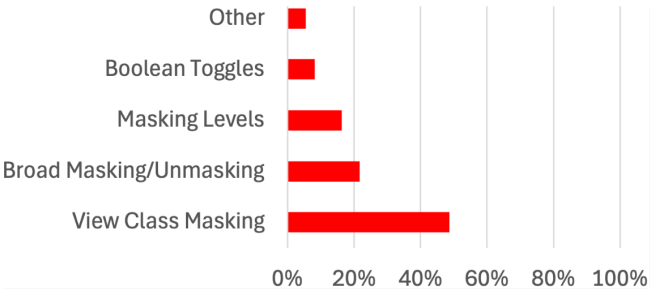
(1) F11 arguments distribution



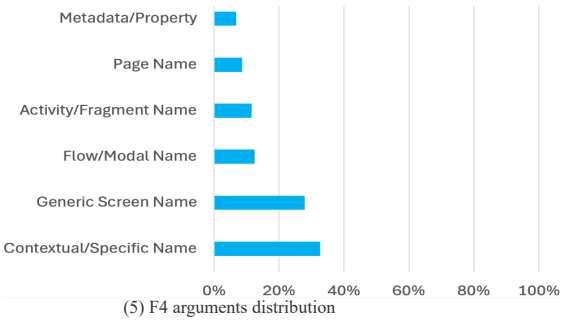
(2) F12 arguments distribution



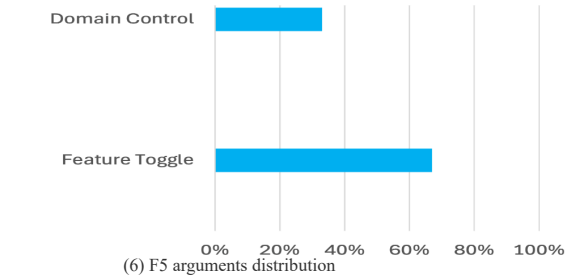
(3) F13 arguments distribution



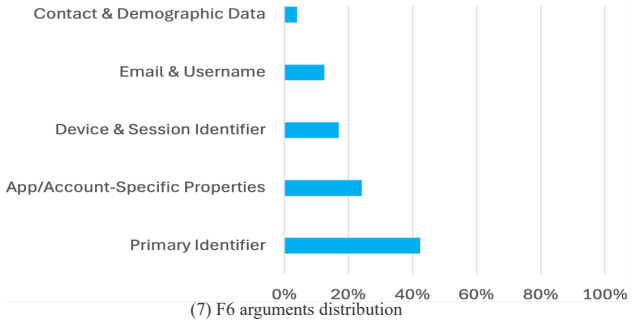
(4) F14 arguments distribution



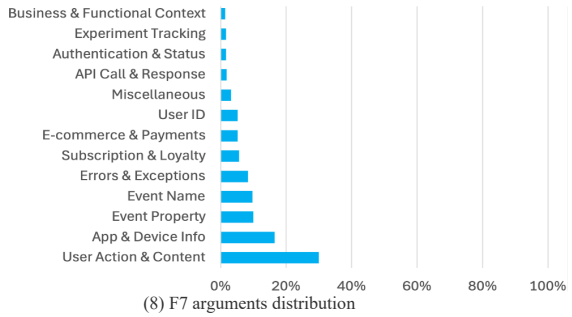
(5) F4 arguments distribution



(6) F5 arguments distribution



(7) F6 arguments distribution



(8) F7 arguments distribution

Figure 12: Distribution of developer-provided arguments for FC4 (left) and FC2 (right) API calls

and generic screen names (≈28%), reflecting the importance of linking replays to identifiable UI flows. In user identification, primary identifiers (≈42%) and account-specific properties (≈24%) dominate, supplemented by device/session identifiers and even direct inputs like emails or usernames (≈13%). These categories confirm that developers actively enrich sessions with user-level data, greatly increasing the potential for linkage between behavior and identity. Finally, in WebView capture, arguments cluster around feature toggles (≈67%) and domain controls (≈33%), showing that developers selectively choose which embedded web content to record.

A sharp asymmetry exists between privacy protection and data enrichment. Masking arguments are overwhelmingly concentrated in broad categories, with fine-grained safeguards seldom adopted. By contrast, enrichment arguments are highly diverse and richly populated, spanning user identifiers, device/session attributes, and contextual metadata. This imbalance suggests that developers systematically invest in maximizing analytic value, while relying on coarse defaults to manage privacy, leaving nuanced masking opportunities largely underutilized.

G.2 Arguments Distribution for API Call Identified with Risks

Figure 13 dissects the subcategories of arguments that drive each type of privacy risk, showing that many violations are not abstract but arise from concrete, repeated developer choices.

For Erroneous Privacy Configuration (R5), the largest share comes from overly permissive configurations, followed by conflicting API calls and flawed privacy logic. This indicates that developers often enable capture too broadly or call APIs in contradictory ways, weakening privacy safeguards. Disregarding User Choice (R6) is almost entirely explained by hardcoded or unconditional opt-ins: every instance traces to developers forcing recording regardless of user preference. This confirms that violations here are not accidental but stem from bypassing consent logic outright. For Masking Misconfiguration (R2), the data spread reveals multiple problematic patterns. The most common are insufficient masking and overly permissive WebView configurations, followed by wildcard rules that unmask broadly, and even explicit unmasking of sensitive data. While less common, conflicting masking rules and global unmasking also appear. These patterns show that even when masking is implemented, developers frequently weaken it by misusing or over-generalizing rules. Omitted Data Practices (R1) is dominated by omissions around app interactions, which dwarf other categories. Much smaller but still noteworthy omissions include personal data, device IDs, and financial information, highlighting that disclosure gaps extend beyond general behavior tracking into sensitive categories. In Hardcoded Consent (R7), arguments cluster around hardcoded opt-ins and unconditional initialization, with smaller shares for probabilistic opt-in, attribute-based opt-in, and cases where no user control or consent check is performed. These reflect different developer shortcuts that effectively nullify user autonomy.

Finally, enrichment-driven risks show more detail. Logging of Sensitive Info (R3) is primarily due to logging of PII, with a smaller share of overly verbose logging, confirming that excessive instrumentation is the main driver. PII Leakage in Identification (R4) distributes across multiple identifiers: most prominently email and user IDs, followed by phone numbers, names, and account IDs, and tapering down to tokens, roles, and national IDs. This breadth demonstrates how SR usage extends beyond technical identifiers to personal and financial details, creating strong potential for re-identification.

Risks are not evenly distributed accidents, but stem from recurring, classifiable implementation practices.

H Feature-Risk Association

Table 10 examines which feature combinations are most strongly associated with specific privacy risks. The top-lift patterns reveal that certain enrichment and control features, though relatively rare in absolute terms, almost deterministically lead to high-risk outcomes when they co-occur. For instance, the combination of Custom Event Tracking (F7) with Explicit User Consent (F16) consistently triggers Hardcoded Consent (R7), with lift values of 8.0. This suggests that when developers introduce explicit consent features, they often implement them incorrectly—hardcoding consent instead of enabling genuine user control. Similar patterns appear with triads like F3+F4+F2 or F11+F4+F2, which also link strongly to R7. These findings indicate that consent-related risks are not random but are systematically tied to particular combinations of tracking and identification features.

Table 10: Associations of SR features and risks

Feature Set	Risk	Support (%)	Confidence (%)	Lift
Top Co-occurrence Patterns (By Lift)				
F7 + F16	R7	1.28	66.67	8.00
F7 + F16 + F1	R7	1.28	66.67	8.00
F3 + F4 + F2	R7	1.28	57.14	6.86
F11 + F4 + F2	R7	1.28	57.14	6.86
F12 + F7 + F13	R2	1.60	71.43	6.55
F12 + F11 + F13	R2	1.60	71.43	6.55
F12 + F13 + F6	R2	1.28	66.67	6.12
F12 + F13 + F8	R2	1.28	66.67	6.12
F12 + F11 + F8	R2	1.92	66.67	6.12
High Support Patterns				
F10 + F1	R1	52.24	84.46	1.03
F1 + F4	R1	19.87	84.93	1.03
F3 + F1	R1	18.59	84.06	1.02
F4 + F6	R1	15.38	84.21	1.02
F7 + F1 + F4	R1	15.06	82.46	1.01

By contrast, high-support patterns largely revolve around Omitted Data Practices (R1). The most frequent pair is Initialization Setup (F1) with Default Masking (F10), covering more than half of all apps (52.24%). Other high-support sets—like User Identification (F4) paired with F1, or Event Tracking (F3) with F1—also consistently map to R1. While their lift values are modest (≈1.0), the sheer prevalence of these patterns indicates that omission of disclosure is a systemic problem across nearly every major feature bundle.

Rare but high-lift feature combinations (e.g., involving consent logic) almost always produce severe risks like R7, and frequent but low-lift combinations (e.g., F1+F10) drive the ubiquity of R1. This contrast highlights that the ecosystem faces both widespread structural risks and concentrated high-severity risks, each requiring different mitigation strategies.

I LLM Prompt Templates Used by TapSpy

This appendix provides the complete prompt templates used to guide the LLM in the two key semantic analysis phases of TapSpy: Semantics-level SRUS Summarization (Phase 2.2), as shown in Figure 14, and Triangulated Risk Auditing (Phase 3), as shown in Figure 15.

J Network Traffic Evidence of SR

Listing 1 provides network-level evidence that UXCam-integrated apps export SR artifacts off-device to vendor infrastructure. The trace begins with a verification request to `verify-<REGION_CODE>.uxcam.com/v4/verify` (left), which returns a server-issued `sessionId` and capture configuration (e.g., `videoRecording=true`). Crucially, the response also includes pre-signed upload parameters under the `s3` field, indicating that the client is authorized to upload session artifacts to UXCam-controlled storage endpoints. We redact app- and device-specific identifiers (e.g., package identifiers, device IDs, session URLs) as well as credential material (e.g., policy/signature fields) for ethical considerations.

Figure 13: Distribution of developer-provided arguments for risky API calls

Following the verify handshake, the client performs a multipart upload to `prod-video-zip-<REGION_CODE>-1.uxcam-storage.com` (right), submitting `video.zip` with an application-layer encrypted payload (e.g., `video.mp4.aes`), and receives an HTTP 200 OK response, confirming successful upload. UXCam also uploads a corresponding `data.zip` artifact (containing encrypted session metadata such as interaction/state logs) using the analogous endpoint `prod-data-zip-<REGION_CODE>-1.uxcam-storage.com` and the same pre-signed form structure; for brevity, we omit the nearly identical `data.zip` packet from the figure and report it in the appendix.

````
You are an expert AI assistant specializing in Android SDK functionality analysis. Your task is to analyze a
**Session Replay Usage Specification (SRUS)** and report which SDK features are used and how they are
implemented.

## 1. Input Format

You will be provided with one JSON input with: Package name of the App and the SRUS. For example:

```json
{
  "packageName": "com.example.app",
  "SRUS": [
    {
      "methodSignature": "<com.example.app.MyClass: void doSomething()>",
      "SDKName": "com.some_sdk",
      "targetAPIsInvokedInMethod": [
        "<com.some_sdk.SDKClass: void startRecording()>",
        "<com.some_sdk.SDKClass: void maskView(android.view.View)>"
      ],
      "slicedMethodBody": [
        "// Decompiled Java-like or Jimple code **slice** showing SDK usage"
      ],
      "dataDependence": [
        "10 -> 14",
        "14 -> 15"
      ],
      "controlDependence": [
        "6 -> 15",
        "8 -> 15"
      ]
    }
  ]
}
```

## **2. Core Task & Analysis Instructions**

1. Examine each `slicedMethodBody` together with its explicit `dataDependence` and `controlDependence`.

2. Identify which SR features (from the global schema) are implemented.
   - Initialization, data enrichment, masking, user identification, session control, etc.

3. For each detected feature:

   - Provide a one-sentence **description** of how it is used.

   - Extract **arguments** when their semantic meaning is clear (e.g., `"user_id"`, `"password_field"`).
   - Provide **evidence** by citing the `methodSignature` and relevant code snippet(s).

## **3. Output Requirement & Schema**

Output must be a **single JSON object** with the following structure:

```json
{
  "analysisStatus": {
    "status": "COMPLETED | SKIPPED",
    "details": "A brief explanation if SKIPPED."
  },
  "functionalityUsage": {
    "Initialization_Configuration": {
        "InitializationSetup": {
          "isUsed": false,
          "description": "",
          "evidence": [],
          "arguments": []
        },
        ...
      },
      ...
  }
}
```

- Evidence must include the methodSignature and code snippet.
- Leave arguments empty if the semantic value cannot be determined.
- Do not include any privacy audit in this step.

## **4. Rules**

- Only set isUsed: true if there is **clear evidence** in the slice.
- Every true must have supporting evidence.
- Leave arguments empty if the semantic meaning cannot be confidently inferred.
- Skip unanalyzable or obfuscated methods without semantics
````

Figure 14: Prompt template for Phase 2 Step 2.2.

```
{
  "request": {
    "method": "POST",
    "url": "https://verify-<REGION_CODE>.uxcam.com/v4/verify",
    "headers": {
      "Accept-Encoding": "gzip",
      "Connection": "Keep-Alive",
      "Content-Type": "application/x-www-form-urlencoded",
      "Host": "verify-<REGION_CODE>.uxcam.com",
      "User-Agent": "okhttp/4.12.0",
      "Content-Length": "<LENGTH>"
    },
    "body": {
      "buildIdentifier": "<REDACTED_APP_ID>",
      "deviceId": "<REDACTED_DEVICE_ID>",
      "osVersion": "16",
      "platform": "android",
      "deviceType": "Phone",
      "deviceModelName": "<REDACTED_DEVICE_MODEL>",
      "appVersion": "14.27",
      //... More device and app metadata
    }
  },
  "response": {
    "status": true,
    "resultCode": 1,
    "data": {
      "appIcon": false,
      "appName": false,
      "sessionId": "<REDACTED_SESSION_ID>",
      "videoRecording": true,
      "settings": {
        "maxOfflineVideos": 15,
        "textFieldPrivacy": [
          { "rule": "record", "isDefault": true }
        ],
        "activitiesToIgnore": [],
        "videoQuality": 3,
        "uploadNetwork": 2,
        "mobileDataLimit": 10,
        "filters": [],
        "filtersDataSession": [],
        "videoPrivacy": [
          { "rule": "record", "isDefault": true }
        ],
        "nativeScreenCapture": true,
        //.... More setting options
      },
      "misc": {
        "city": "<REDACTED_CITY>",
        "ll": "<REDACTED_LAT>",
        "lt": "<REDACTED_LON>",
        //.... More misc metadata
      },
      "s3": {
        "data": {
          "url": "https://prod-data-zip-...",
          "body": {
            "key": "<REDACTED_S3_KEY_DATA_ZIP>",
            "acl": "private",
            "X-Amz-Algorithm": "AWS4-HMAC-SHA256",
            "X-Amz-Signature": "<REDACTED_AWS_SIGNATURE>",
            "x-amz-server-side-encryption": "AES256",
            "file": ""
          }
        }
      },
      "sessionUrl": "<REDACTED_SESSION_URL>",
      "deviceUrl": "<REDACTED_DEVICE_URL>",
    }
  }
}
```

```
{
  "request": {
    "method": "POST",
    "url": "https://prod-video-zip-1.uxcam-storage.com/",
    "headers": {
      "Accept-Encoding": "gzip",
      "Connection": "Keep-Alive",
      "Content-Length": "<LENGTH>",
      "Content-Type": "multipart/form-data; boundary=<BOUNDARY>
          ",
      "Host": "prod-video-zip-1.uxcam-storage.com",
      "User-Agent": "okhttp/4.12.0"
    },
    "multipart_form": {
      "key": "<REDACTED_S3_KEY_VIDEO_ZIP>",
      "acl": "private",
      "success_action_status": "200",
      "X-Amz-Algorithm": "AWS4-HMAC-SHA256",
      "X-Amz-Credential": "<CREDENTIAL>",
      "X-Amz-Signature": "<SIGNATURE>",
      "file": {
        "filename": "<REDACTED_FILENAME>",
        "content_type": "video/mp4",
        "content_length": "<LENGTH>",
        "content_preview": "<REDACTED_PAYLOAD>"
      }
    }
  },
  "response": {
    "status_line": "200 OK",
    "headers": {
      "alt-svc": "h3=\":443\"; ma=86400",
      "cf-cache-status": "DYNAMIC",
      "CF-RAY": "<REDACTED_CF_RAY>",
      "Connection": "keep-alive",
      "Date": "<REDACTED_DATE>",
      "Location": "<REDACTED_LOCATION>",
      "Server": "cloudflare",
      "x-amz-request-id": "<REDACTED_AMZ_REQUEST_ID>",
      "x-amz-server-side-encryption": "AES256"
    },
    "body": ""
  }
}
```

Listing 1: Redacted UXCam network trace illustrating the SR artifact upload workflow

````
You are an expert AI assistant specializing in Android SDK privacy auditing. Your task is to analyze a summary of an
app's Session Replay functionality and its public Data Safety Section (DSS) to identify potential privacy violations.

------
## 1. Input Format

You will be provided with one JSON input containing the `functionalityUsage` summary (generated from a previous
analysis step) and the app's `DSS`. For example:

```
{
  "functionalityUsage": {
    // ... a complete functionalityUsage object from the previous phase ...
    "DataCapture_Enrichment": {
      "UserIdentification": {
        "isUsed": true,
        "description": "The app identifies the user with an email address.",
        "evidence": [{ "method": "<...>", "code": "FS.identify(\"user@example.com\");" }],
        "arguments": ["Email Address"]
      },
      // ... other features ...
    }
  },
  "DSS": {
    "Analytics": [
      "App interactions",
      "Crash logs"
    ]
  }
}
```

## 2. Core Task & Analysis Instructions

Your task is to analyze the inputs and generate a **single JSON object** that audits the application for privacy risks.

1. **Analyze Disclosure Violations:** Compare the SDK's behavior, as described in the `functionalityUsage` summary,
against the app's public disclosures in the `DSS`.
   - First, infer the types of data being collected based on the `functionalityUsage` object. For example,
`UserIdentification` with an argument of "Email Address" implies the collection of "Personal info".
`CustomEventTracking` of purchase events implies collection of "Financial info".
   - Next, check if these inferred data types are declared in the `DSS` under the `"Analytics"` key.
   - If a collected data type is **not** listed in the DSS, report this as a violation under `OmittedDataPractices`.
2. **Analyze Behavioral Violations:** Identify any inherently risky implementation patterns based on the
`functionalityUsage` summary alone.
   - For `MaskingMisconfiguration`, look for evidence of insufficient masking (e.g., unmasking password fields) or
overly broad unmasking (e.g., global unmasking).
   - For `LoggingOfSensitiveInfo` or `PII_LeakageInIdentification`, check the `arguments` and `description`fields for
evidence of PII or other sensitive data being passed to tracking or identification APIs.
   - For `HardCodedConsent`, look for evidence in the `ExplicitUserConsent` or `UserOptIn_OptOut` sections where
consent is programmatically forced (e.g., a hard-coded `true` value).

## 3. Output Requirement & Schema

Your entire response **MUST be a single, valid JSON object.** Do not include any introductory text or explanations. The
JSON object must conform precisely to the `privacyAudit` schema below.

```
{
  "privacyAudit": {
    "Disclosure_Transparency_Violations": {
      "OmittedDataPractices": {
      "isPresent": false,
      "description": "",
      "evidence": [],
      "arguments": []
      },
      ...
    },
    ...
  }
}
```

## 4. Rules and Constraints

1. **Confidence Threshold:** Only set `isPresent` to `true` if there is clear, direct evidence from the
`functionalityUsage`input.
2. **Evidence Requirement:** For every violation marked `isPresent: true`, the `evidence` array MUST be populated using
the corresponding evidence from the `functionalityUsage` input.
3. **Argument Extraction**: Populate the `arguments` array with the specific arguments from the
`functionalityUsage`input that led to the violation finding.
````

Figure 15: Prompt template for Phase 3 Step 3.2.