Navigating Developers' Quagmire: LLM-Enabled Privacy Compliance Analysis for SDK Integrations

Zhaojie Hu, Xueqiang Wang
University of Central Florida, Orlando, USA
{*zhaojie.hu, xueqiang.wang*}*@ucf.edu*

***Abstract*—The use of third-party SDKs has become a major source of privacy noncompliance in mobile apps, creating an urgent need for app developers to ensure privacy compliance during SDK integrations. However, existing methods for identifying privacy-noncompliant SDK integrations (PINs) largely rely on predefined noncompliance patterns based on externally observable app behaviors. These methods are limited in their ability to systematically detect PINs due to the lack of generic detection, and in providing concrete development guidance for app developers on SDK integrations due to limited visibility into the SDK integration context. To overcome these limitations, we introduce a new PIN detection paradigm using privacy-contextual consistency analysis, based on a key observation: PINs often manifest as inconsistencies between the privacy implications of SDK APIs and the context of their integrations, which allow for generic, PIN-independent checks.**

This study takes the first step in validating the feasibility of the new detection paradigm. We first establish the knowledge foundation for this paradigm by defining models that capture API privacy implications, privacy context, and generic consistency analysis rules that describe the proper use and implementation of SDK APIs. Based on the models, we develop an automated framework, PINFINDER, to detect PINs arising from SDK integrations in Android apps – software known for its extensive use of SDKs. The framework combines app analysis techniques for extracting SDK APIs and privacy context, along with large language models (LLMs) for understanding and analyzing privacy-contextual inconsistencies, and targeted enhancements to increase the feasibility of LLM-enabled consistency analysis. Our evaluation confirms the effectiveness and coverage of PINFINDER in detecting PINs. Running PINFINDER on 4,683 real-world apps further sheds light on the prevalence and magnitude of PINs, revealing lesser-known manifestations and their underlying causes. These causes highlight the need for greater standardization in SDK integration API design and improved transparency regarding their privacy implications.

1. Introduction

As mobile apps permeate daily life, they also introduce significant privacy risks. For example, in 2024 alone, Google Play blocked 1.3 million apps for privacy noncompliance [1]. A major source of such noncompliance is the widespread use of third-party SDKs. With the growing diversity of SDK functionalities and their associated privacy implications, how to integrate third-party SDKs while ensuring privacy compliance has become an increasingly pressing challenge for app developers. As reported by recent research [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], various privacy noncompliance issues may arise from SDK integrations, which we collectively refer to as **p**rivacy-noncompliant SDK **in**tegrations (PINs). The vast majority of PINs arise from improper use of SDK APIs during integration and flawed API implementations, such as missing calls to data collection opt-out APIs or ineffective implementations of these APIs.

Existing research prototypes [2], [3], [5], [11], [13], [15] and industry solutions [17], [18], [19] can report presence of some PINs but suffer from two key limitations. First, most existing methods detect PINs by observing app behaviors externally, such as analyzing network traffic to identify unauthorized SDK data collection, without examining the rich context of SDK integrations [3], [5]. As a result, they often fail to provide concrete SDK integration guidance to app developers, particularly on which SDK APIs trigger PINs and which ones could be used to address them. Second, existing methods have largely focused on specific types of PINs by predefining and tailoring noncompliance patterns for those types. For example, Du et al. [13] examined withdrawal options based on the pattern that effective withdrawal options have data or control dependencies with data collection. While effective at detecting specific PINs, these approaches offer limited and non-systematic coverage (as shown by the PIN taxonomy in § 3.1). In practice, however, app developers (as well as regulators) view the systematic detection of noncompliance issues and actionable mitigation as essential, as their absence hinders app privacy compliance [4], [20]. *Despite the critical need, new methods that provide more systematic detection of* PIN*s along with concrete SDK integration guidance have received limited research attention to date*.

Revealing PINs via Privacy-Contextual Consistency Analysis. Our unique observation addressing these limitations is that, although PIN types vary, they often manifest as inconsistencies between the privacy implications (*PI*) of SDK APIs and the privacy context (*PC*) of SDK integrations. Taking two PINs as examples: **PIN-1**: Similar

to those reported in [2], [8], [9], an app integrates an ad SDK and calls its `setDataOptIn` API with a `true` parameter, indicating that users have opted in to data collection (*PI*), despite users not having previously opted in through the app's UIs (*PC*). **PIN-2**: Similar to those reported in [13], [14], [15], even when the `setDataOptIn` API is called with a `false` parameter, indicating that future SDK behavior should refrain from collecting personal data (*PI*), the SDK continues to collect personal data afterward (*PC*). While prior work has proposed different noncompliance patterns tailored to each PIN [2], [8], [9], [13], [14], [15], both PINs ultimately fall into general inconsistencies between *PI* and *PC*, such as the absence of obligatory behaviors in *PC* (e.g., user opt-in via UIs in **PIN-1**) or the presence of prohibited behaviors (e.g., personal data collection in **PIN-2**). Reasoning about these inconsistencies has a greater potential to achieve systematic coverage, as they are not tied to specific PIN types and cover the actual use of SDK APIs (which is essential for generating actionable guidance). This observation raises an important research question: Instead of predefining privacy noncompliance patterns for different PINs, *can we combine the implications of SDK APIs with the privacy context of SDK integrations and perform a more general privacy-contextual consistency analysis as a new way to detect PINs*?

PINFINDER. Exploring the feasibility of this new PIN detection paradigm presents several challenges. First, there is a significant knowledge gap in connecting this concept of privacy-contextual consistency to practice. For example, for SDK APIs that affect privacy compliance (i.e., *PSAPI*s), how should we model them to capture their privacy implications? What does the privacy context look like for broad types of PINs, and what kinds of generic consistency rules could help identify these PINs? Second, even with the knowledge, building a PIN detection tool that realizes this paradigm remains non-trivial, as there has been little exploration of privacy-contextual consistency analysis, such as how to make SDK APIs and privacy context amenable to analysis, and how to effectively check the consistency rules.

This study addresses the knowledge gap by creating a taxonomy of PINs based on prior research, and by building several models that facilitate privacy-contextual consistency analysis based on this taxonomy. The first model is an *API Privacy Description* model, which captures the implications of SDK APIs by specifying when they should be used, how to use them correctly, and what behaviors are expected. The second model is a *Privacy-Context Trace* (PCTrace), which depicts the use of SDK APIs and various aspects of the privacy context relevant to compliance, such as user interactions, network traffic, and the app's privacy commitments. The PCTrace serves as a novel representation of SDK integrations, providing a basis for detecting PINs. The third model is a *Privacy-Contextual Consistency* model, which includes three general consistency rules for guiding the analysis and PIN detection over the PCTrace.

Leveraging these models, we validate the technical feasibility of detecting PINs under the new paradigm, by building an automated framework, PINFINDER, which combines common app analysis techniques with large language models (LLMs). We implement PINFINDER on Android – a platform well known for its extensive use of third-party SDKs [21]. Overall, PINFINDER builds an API modeler to extract privacy descriptions of SDK *PSAPI*s from their privacy guidance documents, constructs PCTraces for SDK integrations using privacy context collected from apps, and applies the consistency rules to the PCTraces to detect PINs. Realizing these steps, however, is challenging and necessitates new technical designs. For example, PCTraces extracted directly from apps include non-uniform modalities, such as network traffic and user interactions, which makes automated analysis difficult. Moreover, while LLMs offer new opportunities for consistency analysis due to their demonstrated semantic and contextual understanding capabilities [22], [23], applying them directly to PCTraces is less practical because (1) the privacy descriptions of SDK *PSAPI*s often contain regulatory, non-technical language that LLMs cannot easily interpret, and (2) analyzing PCTraces for all SDK integrations with LLMs can lead to cross-context contamination, where the privacy context of one SDK interferes with that of unrelated SDKs. To address these challenges, we leverage strategic combinations of LLMs to translate the PCTrace into easily comprehensible language and improve the practicality of consistency analysis by dividing traces into PCTrace slices (representing closely coupled SDKs) and decomposing the analysis into small, verifiable questions over these slices.

Measuring PINs in the Wild. Our evaluation confirms the effectiveness, with the identified PINs covering significantly more PIN types (based on our taxonomy) than prior research (§ 5.2). To investigate PINs in the wild, we constructed an SDK dataset (D_s) containing 64 popular SDKs and an app dataset (D_a) comprising 4,683 apps. Running PINFINDER on the datasets reveals that 882 (18.8%) apps have at least one PIN due to SDK integrations. Of these, 542 (11.6%) apps misuse SDK *PSAPI*s (e.g., failing to obtain user consent before setting consent status to true), 248 (5.3%) apps integrate SDKs that don't align with their claimed privacy behaviors (e.g., SDKs continuing data collection despite promises to stop), and 250 (5.3%) apps fail to invoke SDK *PSAPI*s when required by the context. Analyzing the PINs reveals several interesting and lesser-known insights. First, although current legislation holds app developers accountable for privacy violations, we found that most (88.1%) of the identified PINs originate not from direct SDK integrations by the app, but from SDKs deeper in the integration chain. This highlights the need for privacy analyses that offer deep visibility into the software supply chain while remaining developer-friendly, as demonstrated by our approach. Interestingly, we also found that child-directed apps tend to include fewer third-party SDKs than general apps, yet are more likely to contain PINs when they do. This suggests that while these developers may be more cautious in selecting SDKs, they are not necessarily more technically equipped (possibly even less so) to meet privacy

requirements. Third, we uncovered a range of lesser-known causes behind PINs, such as app developers using self-deceptive parameters in SDK *PSAPIs*, logic errors arising from interactions among SDKs, and mismatches between documented and actual SDK behaviors. These findings point to a need for greater standardization and transparency in SDK documentation, APIs, and their privacy implications. We have responsibly disclosed the PINs to affected stakeholders (app or SDK developers), and some have already been fixed or are in the process of being addressed. To support future research, we have released the taxonomy, datasets, API privacy descriptions, and PINFINDER's source code on an anonymized website [24].

Contributions. The contributions of this paper include:

• **A New PIN Detection Paradigm:** We propose privacy-contextual consistency analysis to enable more systematic detection of privacy-noncompliant SDK integrations and to provide concrete guidance for addressing them.

• **An Automated Framework Validating the Paradigm:** We take the first step toward realizing the detection paradigm by modeling SDK integrations and implementing an automated framework (PINFINDER), which combines app analysis and LLMs with a series of new enhancements.

• **Large-Scale Analysis of PINs in the Wild:** We conduct an extensive measurement study that highlights the widespread occurrence of PINs, revealing lesser-known manifestations and their underlying causes.

2. Motivating Example

```
1 boolean isARU = checkIsChild();
2 AppLovinPrivacySettings.setIsAgeRestrictedUser(isARU,
      context); // stop serving personalized ads and
      collecting data.
3 AppLovinSdk.initializeSdk(context, ...);
```

(a) `App code`: The app developer integrates AppLovin directly.

```
4 if (adapterParams.isARU) {
5     IronSource.setMetaData( is_child_directed , Boolean.
          toString(true)); // Only stop serving of
          personalized ads
6 }
7 IronSource.initISDemandOnly(...);
```

(b) `AppLovin code`: AppLovin integrates ironSource.

Figure 1: Code snippets of Mommy Baby Care Nursery app.

Motivating Example. To provide concrete context, we present a PIN in the Android app, `Mommy Baby Care Nursery`, which has over one million installs [25]. The app allows users, including children, to experience caring for a pregnant mother and her newborn. Our analysis reveals that the integration of a complex SDK chain causes the ironSource SDK in the app to consistently collect device identifiers from children, violating Google Play Families Policy [26] and COPPA [27], [28]. Figure 1a shows the simplified code illustrating the app's use of AppLovin, a major ad platform used by over 10,000 apps [29], for monetization. The app code first asks whether the user is a child (`isARU`, Line 1), and if so, calls AppLovin's `setIsAgeRestrictedUser` API (Line 2) to stop both the serving of personalized ads and the collection of device identifiers by AppLovin. Figure 1b reveals that AppLovin, in turn, integrates the ironSource SDK [30] through a mediation adapter. Based on whether the current user is a child, the adapter invokes the `setMetaData(''is_child_directed'', ...)` API of ironSource (Line 5) to instruct ironSource to treat the user as a child. The PIN with the `App->AppLovin->ironSource` integration chain is that the ironSource API (Line 5) only prevents ironSource from serving targeted ads. However, ironSource will still collect device identifiers (such as AAID) from children unless an additional `setMetaData(''is_deviceid_optout'', ...)` API is invoked for ironSource integration[1]. This PIN could have serious consequences, e.g., technically enabling ironSource to track, profile, and analyze children using these identifiers [11], [31], [32].

Challenges. Prior research [3], [11], [33], [34], [35], [36], [37], [38] and industry solutions [17], [18], [19] typically detect this specific PIN by identifying excessive personal data transmissions in app network traffic. Although this technique flags the PIN via app traffic analysis, it fails to offer actionable SDK integration guidance, e.g., the missing ironSource API (which is visible only in the full integration context). Moreover, extending this technique requires tailoring detection patterns to the full range of PINs, many of which (as shown in the taxonomy) are not directly observable in app traffic. Given the irreversible trend of reusing third-party SDKs, today's app developers expect approaches that address a range of PINs while providing actionable guidance to improve SDK integration [4], [20].

Observation. The app's Data Safety section [25] indicates that the app commits to complying with the Play Families Policy (①). For a child user, the user interactions with age-related UI presented by the `checkIsChild` method in Figure 1a, indicates that "*the current user has identified herself as a child through the age-related UI* (②)". Meanwhile, ironSource [39] states that when users are children and the app commits to complying with the Play Families Policy, the `setMetaData` API must be called with the parameters `(''is_deviceid_optout'', ''true'')` to prevent access to the AAID (③). By reasoning over these three pieces of information, one can determine that the conditions triggering the ironSource API (① and ②) are satisfied, making its invocation obligatory. The actual absence of the API invocation indicates a contextual inconsistency, resulting in a PIN. Performing such contextual consistency analysis across multiple SDK integrations could overcome the two main limitations of existing methods. First, these inconsistencies (e.g., unmet obligations) represent a broader class of issues that are not confined to specific PIN types. Second, analyzing the actual SDK APIs in the context of SDK integration, rather than focusing solely on external

1. AppLovin addressed this issue by adding the API in version 12.0.0.

app behaviors (e.g., network traffic), provides more app-internal, actionable guidance for app developers. This insight motivates the exploration of *a new paradigm for detecting* PINs *based on privacy-contextual consistency analysis*.

3. Understanding and Modeling PINs

3.1. A Taxonomy of PINs

To characterize the current landscape of PINs, we developed the taxonomy through a systematic review of the literature and collaborative coding of reported PINs. We identified papers reporting PINs from seven leading venues in privacy, security, and software engineering: IEEE S&P, USENIX Security, CCS, NDSS, PETS, ICSE, and FSE. From the proceedings published since 2020, we searched paper abstracts for the coexistence of two keyword sets, i.e., "`{privacy, personal data, complian\w*}`" (indicating privacy-related issues) and "`{librar\w*, sdk, third-?part\w*, advertis\w*}`" (indicating the involvement of third-party SDKs). This process yielded 188 candidate papers. Two researchers with expertise in privacy compliance manually reviewed them and agreed that 14 clearly reported at least one PIN. We note that, beyond these papers, many others have analyzed SDK data processing without proper disclosure in privacy policies or labels [3], [34], [35], [36], [37], [38], [40], [41], [42]. However, we excluded them from the analysis for two reasons: (1) their causes are not solely attributable to app developers' SDK integration but often involve other stakeholders such as privacy officers and legal counsel [43]; and (2) current practice treats data disclosure in privacy policies and labels as a post hoc activity rather than part of the development process, as reflected in prior work [44], [45], [46], [47], [48], [49].

Because the PINs were complex and independent coding would be difficult, the two researchers conducted collaborative coding, identifying potential PINs, resolving ambiguities, and iteratively refining categories of the PINs. This process resulted in three broad PIN categories and 16 subcategories at the second level of the taxonomy, reflecting the diverse manifestations of PINs. Table 1 presents the taxonomy of PINs along with the papers that report them. Overall, PINs stem from three key aspects of SDK APIs used during SDK integration, including (1) misuse of SDK APIs that violates various *correct-use conditions*, such as requiring API calls to follow a specific order (T1), or occur only after valid user consent (T2 and T3); (2) missing SDK APIs that violate *triggering conditions* requiring their use, for example, when users identify themselves as children, the APIs that limit data use for ad tracking should be invoked (T12); and (3) ineffective API implementations that fail to fulfill their claimed behavior, such as APIs that claim to limit data collection but do not do so (T15). These three broad aspects inform the design of the privacy-contextual consistency model (§ 3.3), which defines the scope of PINs that our tool reports.

3.2. Privacy Context for Detecting PINs

We analyzed techniques from prior research to detect PINs and understand the privacy context needed for detection. Table 1 summarizes the five major categories of privacy context for each PIN type, which are described below.

App-Level Privacy Commitments (C_{app}). Prior research, such as [2], [6], considers an app's commitments to privacy requirements or regulations important, as they apply to all SDKs integrated within the app. For example, an app's descriptions and related pages, such as the *Data Safety* section [50] on Google Play, may state that "*The developer has committed to follow the Play Families Policy*." Such privacy commitments serves as privacy context for SDK APIs, as SDK privacy guidance documents often specify how to use these APIs to fulfill the stated commitments [39].

App Running Environments (C_{env}). Besides C_{app}, app running environments such as IP address, geolocation, and user age are commonly used in prior studies [4], [10], [34] to determine the applicability of privacy regulations. For example, an app running with a California IP address may be a basis for the applicability of COPPA.

User Interactions (C_{user}). User interactions with the app, such as engaging with opt-out options [13], [14] and consent requests [5], [51], reflect users' privacy preferences and define the expected use of SDK APIs.

Data Processing Events (C_{data}). As demonstrated in prior studies [3], [11], [33], [34], [35], [36], [37], [38], data processing events such as the access, collection, and sharing of user personal data by SDKs have been instrumental in detecting a range of PINs. Typically, these studies identify such events using several complementary approaches, such as the invocation of system-level APIs that access personal data [35], the analysis of network traffic [3], [4], [6], [10], [11], and data flows as shown in app or SDK code [2], [13].

Use and Non-Use of SDK APIs (C_{api}). The use and non-use of an SDK API can provide context for the correct use of other APIs due to cross-API constraints. For instance, including the ironSource initialization API can help detect the PIN related to the invalid invocation order (**T1**), as the `setMetaData(''is_child_directed'', ...)` API must be "*set before initializing the SDK*" [39] to be effective.

3.3. Privacy-Contextual Consistency Model

API Privacy Description. Since PINs often stem from the use and implementation of SDK APIs, we introduce an API privacy description model that captures the aspects of SDK APIs relevant to privacy compliance. Aligned with the aspects violated by the three major categories of PINs (in Table 1), SDK privacy guidance documents typically specify when an SDK's *PSAPI*s must be used, the expected privacy behaviors, and the conditions for correct usage. For example, the AppLovin documentation [52] describes calling the `setIsAgeRestrictedUser` API with a `true`

TABLE 1: Summary of the PIN taxonomy

	PINs and Related Research	Types of Privacy Context				
		C_{app}	C_{env}	C_{user}	C_{data}	C_{api}
Misuse of SDK APIs	T1: Invalid API invocation order [2]					✓
	T2: SDK usage triggering data collection without user consent [3], [4], [5], [6]	✓	✓	✓	✓	
	T3: SDK usage triggering data collection despite user rejection [3], [5]		✓	✓	✓	
	T4: Passing un-anonymized PII to SDK API [7]	✓				✓
	T5: Misuse of APIs that limit data collection [2], [8], [9]		✓	✓	✓	✓
	T6: Misuse of APIs that opt out of data sharing/selling [2], [10]		✓	✓	✓	✓
	T7: Misuse of APIs that limit data use for ad tracking [2]			✓		✓
	T8: Misuse of APIs that flag child users [2]	✓		✓		✓
	T9: Misuse of APIs that make privacy regulations applicable [4]	✓			✓	
Missing SDK APIs	T10: Missing APIs for limiting data collection [2], [4], [6], [12], [13]	✓		✓	✓	✓
	T11: Missing APIs for opting out of data sharing/selling [2], [4], [10]	✓	✓	✓	✓	✓
	T12: Missing APIs for limiting data use for ads tracking [4], [12]	✓			✓	
	T13: Missing APIs for flagging child users [2], [4], [12]	✓		✓	✓	✓
	T14: Missing APIs for making privacy regulations applicable [4]	✓			✓	
Ineffective API Implementations	T15: SDK APIs failing to limit data collection [13], [14], [15]	✓		✓	✓	✓
	T16: SDK APIs failing to opt out of data sharing or selling [10], [16]		✓	✓	✓	✓

parameter as: "*If the user qualifies as a child under applicable laws* [`triggering condition`], *set the age-restricted user flag this way before initializing or using the AppLovin SDK* [`correct-use condition`]," and "*does not collect personal information from children or serve ads to children* [`expected behavior`]." Building on this, we define an API privacy description as a four-tuple, specified as follows.

$$\mathcal{A} = \{(\theta_i, P_i, M_i, B_i) \mid \theta_i \in \Theta\}$$

Where:

- $\Theta = \{\theta_1, \theta_2, \ldots, \theta_t\}$ represents the set of possible parameter combinations, where each combination $\theta_i = (v_{i1}, \ldots, v_{in})$ is a tuple of parameter values for the API. Each $\theta_i \in \Theta$ is associated to three elements: P_i, M_i, and B_i. all expressed in plain languages.
- $P_i = \{p_{i1}, \ldots, p_{im}\}$ represents the set of correct-use conditions[2] (e.g., presupposed conditions, temporal constraints between API invocations, and validity constraints on parameters) that needs to be true for the correct invocation of the API.
- $M_i = \{m_{i1}, \ldots, m_{ij}\}$ represents the triggering conditions that, if met by the context, make the invocation of the API obligatory.
- $B_i = \{b_{i1}, \ldots, b_{ik}\}$ represents the set of expected behaviors that describe the privacy changes to the SDK state when the API is invoked. Examples are prohibitions enforced by the API (e.g., disable data collection).

In Figure 6 in the Appendix A, we show the privacy description for the `setIsAgeRestrictedUser` API.

Privacy Context Trace. We consider a privacy context trace (PCTrace) as a privacy-focused representation that captures all major types of privacy context in SDK integrations (as identified in § 3.2). A privacy context trace (PCTrace), denoted as $\mathcal{T}$, is constructed for each app execution and consists of a time-ordered sequence that unifies all privacy context necessary for detecting PINs, as follows:

$$\mathcal{T} = C_{app} \parallel C_{env} \parallel \text{sort}_{\text{time}}(C_{user} \cup C_{data} \cup C_{api})$$

2. We avoid using preconditions or postconditions, as they pertain to parameters and return values but correct-use conditions extend beyond that.

Where: $\parallel$ concatenates two lists, and $sort_{\text{time}}$ combines several time-ordered lists based on time.

Privacy-Contextual Consistency Model. Considering the three categories of PINs and the elements in SDK *PSAPI*s' privacy descriptions, we expect a privacy-compliant SDK integration to satisfy at least three privacy-contextual consistency rules.

• Rule 1 (**Correct-Use Rule**). For an SDK *PSAPI* A already used in SDK integrations with parameters θ_i, with a privacy description denoted as $\mathcal{A}$, all correct-use conditions of the *PSAPI* (P_i) needs to be met by the PCTrace (denoted as $\mathcal{T}$). An example is that calling the API `setHasUserConsent(true)` should be preceded by a user interaction in $\mathcal{T}$ that actually obtains consent from the user.

• Rule 2 (**Triggering Rule**). For an SDK *PSAPI* A, with the API privacy description denoted as $\mathcal{A}$, if the PCTrace (denoted as $\mathcal{T}$) satisfies all triggering conditions of the *PSAPI* with parameters θ_i, then the *PSAPI* A should have been used during the integration of the SDK.

• Rule 3 (**Behavior Rule**). For an SDK *PSAPI* A already used in SDK integration with parameters θ_i, with the API privacy description denoted as $\mathcal{A}$, the privacy changes caused by the API's behaviors (B_i) need to be met by the PCTrace (denoted as $\mathcal{T}$). For example, once the API `setIsAgeRestrictedUser` is called for child users, its behavior (stopping the collection of device identifiers) must be followed by the data processing events in the PCTrace.

4. Design of PINFINDER

To validate the feasibility of the new PIN detection paradigm, we propose PINFINDER, an automated framework for detecting PINs in Android apps (which are known for their extensive use of SDKs [21]). Figure 2 shows an overview of PINFINDER. The first component is an API privacy modeler that takes SDK privacy guidance documents and SDK code as input and extracts privacy descriptions for SDK *PSAPI*s based on the definition in § 3.3. The second component is a PCTrace builder that takes app data and SDK *PSAPI* privacy descriptions to generate a PCTrace for each app, by joining and synthesizing privacy context like user

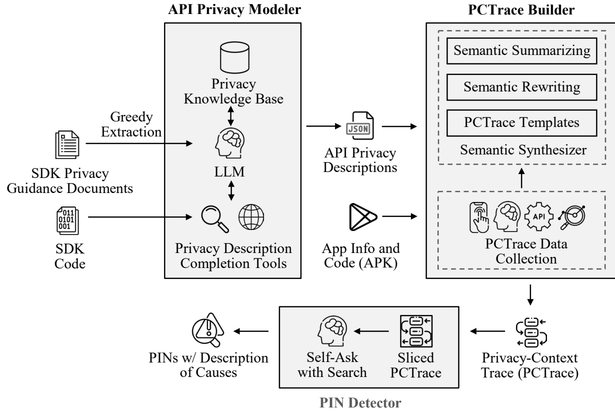

Figure 2: Overview of PINFINDER

interactions, data processing events, and SDK API usage (as described in § 3.3). The third component is a PIN detector that uses LLMs to detect violations of the privacy-contextual consistency rules based on the PCTrace (see § 3.3). Below, we introduce the design of these components, highlighting the real-world challenges and our ways to address them.

4.1. API Privacy Modeler

Prior research has explored extracting API models from documents using NLP techniques [53], [54], [55], [56], [57], [58]. Recent advancements in LLMs have significantly simplified this extraction task [59], [60]. Inspired by these, we apply LLMs for extracting API privacy descriptions from SDK privacy guidance documents. Specifically, we first convert the documents published on SDK websites (in HTML format) into Markdown using html2text [61], which removes web-specific code noise while preserving structural elements such as hierarchical sections. The modeler loads the Markdown documents as input and prompts pre-trained LLMs to (1) identify *PSAPI*s from the documents, and (2) for each *PSAPI*, build a privacy description specifying its parameters, correct-use and triggering conditions, and expected behaviors. The modeler further provides examples for few-shot prompting [62], and directs LLMs to follow a JSON format based on the API privacy description in § 3.3.

Although the modeler yields some correct results, directly applying LLMs faces challenges that reduce extraction accuracy and completeness. First, pre-trained LLMs, with their broad knowledge, often flag non-privacy-sensitive APIs as privacy-related, like the `setMuted(boolean)` API [29] in AppLovin, which simply mutes audio but is incorrectly identified as privacy-sensitive. Second, the unique characteristic of overlapping contexts in SDK privacy guidance documents can lead LLMs to make incorrect cross-API inferences. For example, the `setGppConsentString` API of MobileFuse [63] was mistakenly labeled with the "*under 13 years of age*" condition due to shared context with `setSubjectToCoppa`. Third, incomplete SDK documents, such as missing API parameter values, can create gaps in privacy descriptions. For example, the `setUSPrivacyString` API [64] specifies only `1YNN` as a valid parameter but omits many others defined by the relevant privacy standard [65].

An Agentic API Privacy Modeler. To address these challenges, we introduce three enhancements to the modeler: **(1)** The first is a *privacy domain-specific knowledge base* that helps filter out non-privacy-sensitive SDK APIs. We build a FAISS vector knowledge base [66] by indexing texts from three major privacy laws (GDPR [67], CCPA [68], COPPA [28]) and the Google Play Families Policies (GPFP) [26], and integrate it into an agentic modeler using LangChain [69], a framework for developing language model-powered applications. For each SDK API flagged by the LLMs as a potential *PSAPI*, the agent queries the knowledge base to verify that the API is related to privacy regulations before extracting its privacy description. **(2)** The second enhancement is a *greedy strategy*. Instead of processing entire SDK documents with overlapping context, the modeler iteratively presents blocks of text surrounding the references to *PSAPI*s to the LLMs, progressively building up the text block hierarchy. This continues until all elements of the API privacy description (as defined in § 3.3) are identified, minimizing irrelevant or noisy content. **(3)** The third enhancement introduces two *privacy description completion tools*. The first, `code-search`, analyzes SDK code archives (e.g., JAR or AAR) using Soot [70] to identify fully qualified class names, method signatures, and values associated with API parameters by scanning assignments and conditional checks on the parameters. For parameter values not explicitly noted in the privacy guidance documents, the modeler infers their meanings based on those that are noted. The second tool, `DuckDuckGoSearch`, is an online search tool integrated into the LangChain framework. For API privacy descriptions extracted from SDK documents, the modeler uses additional prompts to guide the completion of the description based on online privacy standards, by automatically calling the `DuckDuckGoSearch` tool, e.g., "*For API parameters with fewer than two values, use DuckDuckGoSearch to check whether a parameter is defined by a privacy standard. If so, retrieve its possible values and meanings online.*" As evaluated in § 5.1, these enhancements notably improve the modeler, with a 94.7% F1-Score in identifying *PSAPI*s and an 89.0% F1-Score in the extraction of API privacy descriptions, representing a 10.7% and 8.2% improvement compared to the modeler without the enhancements.

4.2. PCTrace Builder

PCTrace Data Collection from Apps. The PCTrace builder first collects the five types of privacy context (§ 3.2) with a combination app dynamic analysis techniques and LLMs. Specifically, the builder collects app privacy commitments (C_{app}) by first crawling its *Data Safety* section [50] published on Google Play, and detects commitments by matching keywords typically hardcoded in the section, such as `committed` and `doesn't`. Examples of the C_{app} are *The developer has committed to follow the Play Families*

Policy" and *The developer says this app doesn't [share or collect] user data*". Moreover, the builder collects two representative app running environments (C_{env}) across testing runs: user age and IP address, following prior practices in [4], [10], [16], [34], [71]. To account for age, the builder signs into a test device using Google Accounts configured for each eligible age group and runs the app. The device's IP address is adjusted using NordVPN [72] to simulate different locations. Note that each testing run of the app results in a separate PCTrace.

For user interactions (C_{user}), the builder simulates and captures UI events using FastBot [73], a widely used UI testing tool known for its extensive UI coverage [74], [75], [76]. We log various user interactions triggered on clickable UI elements, such as button clicks and checkbox toggles, along with associated text data and timestamps. To enrich the semantics of these events, especially when information is limited (e.g., a button labeled "`OK`"), we capture additional context, including nearby UI element text, navigation path, UI title, and managing classes (i.e., the activity stack). A complexity arises from the fact that C_{user} may interfere with other types of privacy context, e.g., user interactions with an age gate or geolocation selection UI interfering with the setup in C_{env}. To ensure consistency between these types of privacy context, we detect the presence of age- and location-related UIs, and determine user interactions on such UIs aligned with C_{env}, through LLM-guided UI exploration inspired by prior research [77].

During the app testing runs, the PCTrace builder utilizes a dynamic instrumentation framework, Frida [78], to monitor SDK *PSAPI* invocations (C_{api}). Specifically, the builder injects hooks into the *PSAPI*s using the API signatures collected in § 4.1 and records each invocation, including parameters, return values, call stacks, and invocation time. Furthermore, the PCTrace builder collects events related to data access and transmission (C_{data}) through API and network traffic analysis. Specifically, we reviewed several open-source privacy tools, such as Camille [79] and PrivacySentry [80], to compile a list of data-accessing APIs that retrieve personal data from the system, including 73 APIs across 12 categories, such as accessing location data, device identifiers, and network information. Similar to monitoring SDK *PSAPI*s, the builder uses Frida to log the invocation of these data-accessing APIs. To assess whether personal data is transmitted, the builder uses Burp Suite [81] to intercept and analyze network traffic during app testing. It matches the return values of the data-accessing APIs to determine whether specific personal data is transmitted and identifies the domains to which the data is sent – a common practice for analyzing personal data transmission from network traffic [82], [83].

Semantic Synthesis. Connecting the above data in the way as defined in § 3.3 forms a preliminary PCTrace. However, such a PCTrace is not of uniform modality, with some parts being in plain language (e.g., C_{app}) and other significant portions (like C_{api} and C_{data}) in different modalities, which hinders the comprehensibility of the overall PCTrace (and further consistency analysis built on it). We address this problem using a semantic synthesizer that losslessly transforms the preliminary trace into plain language.

• **PCTrace Templates**. Central to the synthesizer is a set of descriptive templates that transform PCTrace data into plain-language sentences. Table 2 presents several example templates. For instance, by inserting the user's age into the template "*The app is currently used by a user aged [age],*" C_{env} can be easily expressed in plain language. Similar conversions apply to other data types, such as describing the use and unuse of SDK *PSAPI*s by explicitly noting their triggering and correct-use conditions, and expected behaviors. For instance, if an SDK is initialized in an app but its *PSAPI*s are not used in the integration, the template specifies when the *PSAPI*s must be invoked, e.g., "*In the case that [triggering-conditions], the [api] of [sdk] must be called with [parameters].*" These *PSAPI*-related sentences often specify clear requirements regarding the privacy context, facilitating consistency analysis over the PCTrace.

TABLE 2: Templates for Different Types of PCTrace Data

Type of Context	Template
C_{app}	[app privacy commitments].
C_{env}	The app is currently used by a user of age [age]. The app is currently used by a user in [location].
C_{user}	The user [action] the [UI element] on a UI for [UI purpose].
C_{api}	The invocation of [api] of [sdk] must meet the following conditions: [correct-use conditions]. If [triggering conditions], the [api] of [sdk] must be called with [parameters]. The [app | sdk] invokes the [api] of [sdk] with [parameters] such that [sdk] will [expected behaviors].
C_{data}	The [sdk] accesses [personal data]. The [sdk] collects [personal data] and transmits it to [domain].

• **Semantic Rewriting**. One complication arises from the implicit nature of some API privacy descriptions, particularly when describing prohibitions and obligations. For example, the behaviors of some *PSAPI*s are expressed using soft, indirect phrasing, e.g., the documents of Digital Turbine describe the behavior of the `setGdprConsent` API as "*only contextual ads can be displayed*" rather than "*targeted ads will be disabled.*" We expect the latter cases because, by directly describing the prohibited behaviors, they make potential inconsistencies between the implications of SDK *PSAPI*s and the privacy context more explicit. Addressing the indirect phrasing requires domain-specific knowledge of the content of SDK privacy guidance documents. To address this, we manually collected common phrase patterns used in the documentation of 20 SDKs (randomly sampled from the Google Play SDK Index, with no overlap with the evaluation dataset in § 5) to express prohibitions or obligations. Table 5 in the Appendix A lists these patterns found in SDK documentation. The semantic synthesizer identifies these patterns in API privacy descriptions and leverages LLMs to rewrite them using explicit terms like "`forbid`" or "`must`" before applying the template.

• **Semantic Summarizing**. Another challenge arises from fragmented and noisy UI data. For example, the click

event on the `Switch to private` button in the *Instagram* app (shown in Figure 5 in the Appendix A) alone does not provide enough information. Including surrounding UI elements, however, leads to 12 pieces of information totaling over 150 words, which results in lengthy, cluttered sentences with excessive and unclear semantics. To address this, the semantic synthesizer performs LLM-based semantic summarization to extract the purpose of the UI from a variety of UI data, such as the text of nearby UI elements, the navigation path and title of the UI, and the classes managing the UI. The identified UI purposes are then incorporated into the template for user interactions.

4.3. PIN Detector

Motivated by the semantic and contextual understanding capabilities of LLMs [22], [23], we build an LLM-enabled PIN detector that identifies violations of the three privacy-contextual consistency rules (§ 3.3) in the PCTrace. However, two major challenges arise. First, the privacy descriptions of some *PSAPI*s may include conditions or behaviors expressed in non-technical, regulatory language that cannot be directly verified from the PCTrace. For example, calling `setAgeRestrictedUser(true)` in HyprMX is intended to enable child-directed treatment, a regulatory concept that implies a range of technical requirements, such as disabling personalized advertising and prohibiting data collection [84]. When LLMs are used to check rules involving such conditions or behaviors, they often struggle to identify the appropriate evidence to validate within the PCTrace. Second, when using LLMs to check a full app PCTrace (which includes many SDKs), they often yield logically correct results, but many do not correspond to actual PINs due to cross-context pollution. For example, the *PaperColor* app uses the `clearGdprConsentData` API from the `Digital Turbine` SDK, which "clears any existing GDPR consent flags and data" for both this SDK and other upstream consumers. However, this contextual information was mistakenly applied to assess the *AdMob* SDK that is unrelated to *Digital Turbine*. As a result, even though *AdMob* obtained user consent, its API invocations that request ads based on personal data were erroneously flagged as violations of the correct-use rule (**T2**) due to cross-context pollution.

Self-Ask with Search on PCTrace Slices. As shown in Figure 3, the PIN detector first identifies three aspects of SDK *PSAPI*s (i.e., expected behaviors, correct usage, and triggering conditions) and then verifies whether the corresponding consistency rules (§ 3.3) are followed in the PCTrace. To address the challenge of non-technical requirements, we prompt LLMs to determine whether the rules are directly verifiable from the PCTrace. For those that are not, we apply Self-Ask with Search (SAwS) [85], leveraging its ability to decompose rules into smaller, verifiable sub-rules, supported by external knowledge retrieved via search. For example, as shown in the analysis of the HyprMX SDK integration in Figure 3, the task of verifying the enablement of child-directed treatment is decomposed into more specific sub-rules, such as disabling the collection of device identifiers and disabling personalized ads. These sub-rules are effectively verified on the PCTrace, which reveals the access and collection of the AAID, leading to the detection of a PIN with a detailed description.

To reduce cross-context pollution, we introduce the concept of PCTrace slices, which confines consistency analysis to closely related SDKs. Specifically, PINFINDER analyzes an app's call graph to identify its SDK integration chains, e.g., the package list [`com.app`, `com.applovin`, `com.ironSource`] represents the ironSource integration chain in the motivating example. A PCTrace slice is created for each SDK integration chain by including all app-level privacy commitments, along with other PCTrace data, but only if they fall within the scope of the identified packages. This includes cases where stack traces of data-accessing APIs overlap with the packages, and the classes managing user interactions belong to the packages. Besides eliminating cross-context pollution, consistency analysis on PCTrace slices offers other advantages, such as preventing LLM failures caused by the full PCTrace exceeding the token limit. Our evaluation (§ 5.2) demonstrates that this approach, which combines SAwS and PCTrace slices, is effective, achieving 89.6% precision in detecting PINs, representing a 68.2% improvement over the detection without SAwS and PCTrace slices.

5. Evaluation

SDK Dataset (D_s). The SDK dataset contains 64 popular Android SDKs. To build it, we began by selecting the 154 popular SDKs indexed by AppBrain library statistics [86]. Next, we expanded the set through snowball sampling, iteratively reviewing SDK documentation to identify other SDKs integrated by the collected ones. This process added 45 SDKs, resulting in an initial set of 199. We then examined the documents of these SDKs to identify and exclude those that fail to explicitly address `privacy`, `regulation`, or `compliance`. This filtering narrowed the scope to 64 SDKs, including 43 ads SDKs, 24 analytics SDKs, and 2 others (e.g., push notifications), with some belonging to multiple categories. For these 64 SDKs, we also collected their code archives (JAR or AAR) and privacy guidance documents, yielding 64 archives and 119 documents in total.

App Dataset (D_a). We ran *google-play-scraper* [87] in November 2024 to obtain a list of the top 100 free apps in each of the 54 app categories (e.g., education, social). After deduplicating, we compiled a list of 5,044 unique app packages Due to the lack of API access to Google Play, we downloaded apps from APKPure [88], an alternative app store commonly used in prior research [89], [90], to obtain the app's code (i.e., APK, XAPK). In total, we collected 4,683 apps available on APKPure, which almost evenly distribute across the 54 app categories, and added them to the app dataset for further analysis.

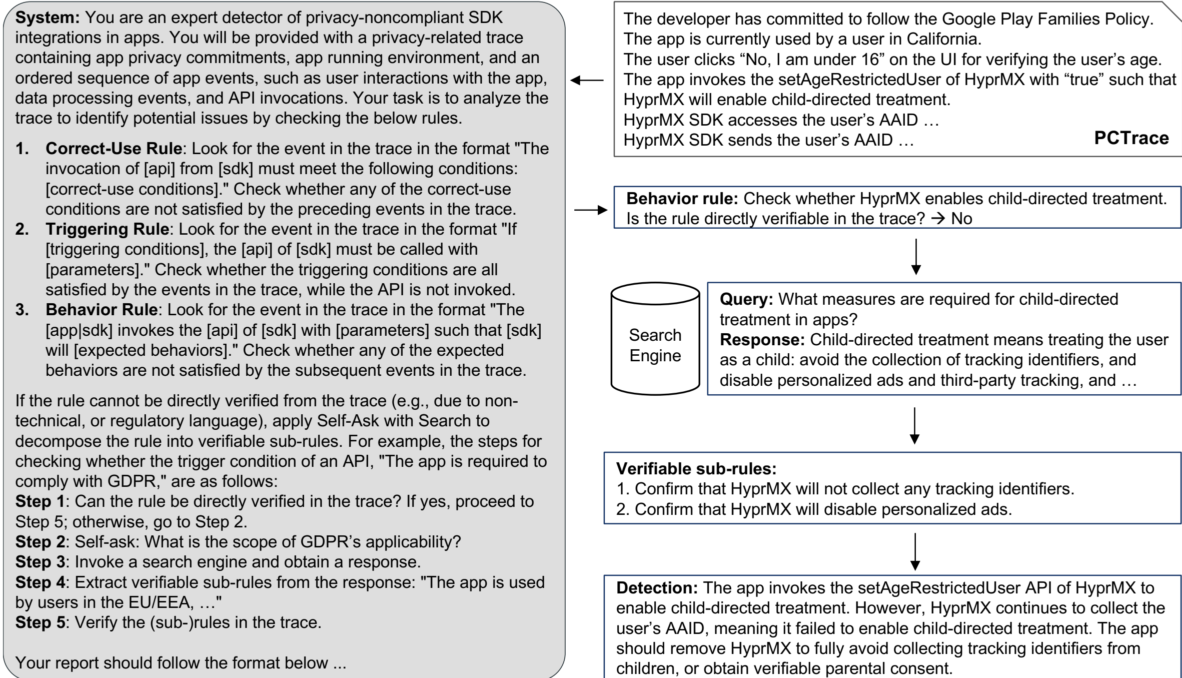

Figure 3: System prompt used by the PIN detector

5.1. Evaluation of API Privacy Modeler

Groundtruth of API Privacy Descriptions. We randomly sampled 10 SDKs from D_s to build a groundtruth of api privacy descriptions. Specifically, two researchers (both with 2.5-3 years of app development experience) independently reviewed 12 privacy guidance documents of the SDKs to identify their *PSAPI*s. They also annotated the conditions and behaviors of these APIs, based on those defined in the API privacy description model in § 3.3. Overall, the researchers achieved a Cohen's Kappa score of 0.94 for identifying *PSAPI*s and 0.91 for annotating the conditions and behaviors of these *PSAPI*s. Following individual annotation, they discussed and resolved their disagreements, which results in 45 *PSAPI*s and 153 API conditions and behaviors.

Effectiveness of *PSAPI* Identification. We built and ran an API privacy modeler using GPT-4o, the flagship model of ChatGPT [91], on the sampled SDKs. The modeler generated privacy descriptions for 48 potential *PSAPI*s, comprising 166 condition or behavior elements. The modeler produced 4 FPs and 1 FN in identifying *PSAPI*s, resulting in a precision of 91.7%, a recall of 97.8%, and an F1-score of 94.7%. The FPs are mainly caused by: (1) The SDK documents include APIs for platforms other than Android, such as iOS, which we classify as FPs, as they fall outside the scope of this study; (2) The modeler sometimes struggles to differentiate between *PSAPI*s (those with real privacy implications) and those that provide privacy functionalities. For example, the `openPrivacyDashboard` API, which only opens a privacy dashboard, is incorrectly flagged as an *PSAPI*. The FNs stem from the crawling of SDK documents that failed to capture *PSAPI*s in dynamic web content.

Effectiveness in Extracting API Privacy Descriptions. The modeler yields 24 FPs and 11 FNs in extracting conditions and behaviors for API privacy descriptions, with a precision of 85.5%, recall of 92.8%, and F1-score of 89.0%. A key source of FPs is LLMs' misinterpretation of privacy implications. For instance, from the sentence *"you must send your users' consent statuses"* in Unity Ads' documentation [92], the modeler incorrectly infers that "*user must express consent*" is required before calling `MetaData.set(''privacy.consent'', ''false'')` - an API that indicates the user has declined consent. FNs are mainly caused by the modeler's greedy extraction strategy, which stops once it fills all fields in an API's description, leading to missed conditions or behaviors when they spans multiple text blocks or hierarchy levels.

Comparison with Direct LLM-Based Extraction. We run a direct LLM-based extraction on the sampled SDKs without the enhancements described in § 4.1, resulting in F1-scores of 84.0% for extracting *PSAPI*s and 80.8% for identifying condition and behavior fields in the API privacy descriptions. These results show a 10.7% and 8.2% decrease in F1-scores for both aspects, compared to the modeler with the enhancements, confirming that the enhancements notably improve the effective extraction of API privacy descriptions.

5.2. Evaluation of PIN Detection

App Execution Setups. The API privacy modeler extracted privacy descriptions of 224 potential *PSAPI*s for the SDKs

in D_s. These potential *PSAPI*s serve mainly three purposes: 92 (41.1%) are designed to support GDPR compliance, 64 (28.6%) for CCPA, and 42 (18.8%) for COPPA. Other less common purposes include APIs for controlling location tracking (15) and for privacy-sensitive initialization of SDKs (4). To align with these major purposes, we evaluated and measured the PINs using three setups: (1) **S-EU**: To check GDPR-related PINs, we verified the availability of the D_a apps in EU countries by setting the country parameter to "`fr`" (France) in *google-play-scraper* [87]. For the 4,641 apps available, we evaluated each app using a OnePlus Ace3V device connected through a French IP address. (2) **S-CA**: To check CCPA-related PINs, we specified the country parameter as "`us`" to validate app availability in the US, and then ran 4,683 available apps from an IP address located in California. (3) **S-Child**: To examine COPPA-related PINs, we identified 201 apps that have committed to comply with Google Play Families Policies. We tested these apps on a device configured with a child's Google Account and an IP address located in a US state. In total, the three setups result in 9,625 testing runs of the D_a apps, with PIN detection costing $246.87 ($0.026 per app, a rate that is substantially lower than that of commercial developer compliance tools [19], [93]).

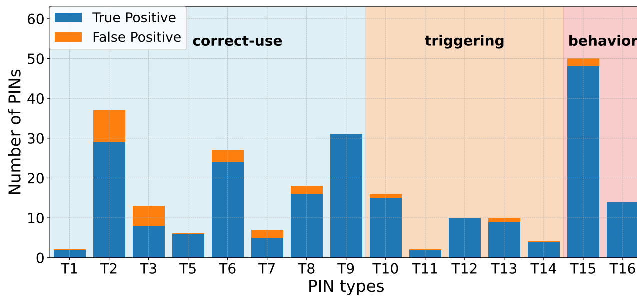

Figure 4: Distribution of PINs across different types

Overall Effectiveness. Applying PINFINDER with GPT-4o across the 9,625 testing runs identified 1,881 potential PINs across 882 apps. To assess PINFINDER's effectiveness, we randomly sampled 1,000 testing runs (517 from `S-CA`, 469 from `S-EU`, and 14 from `S-Child`) and manually reviewed the reported PINs. In total, 55, 55, and 4 apps showed PINs for the three setups, respectively, amounting to 249 potential PINs (142 from the correct-use rule, 43 from the triggering rule, and 64 from the behavior rule). Among these, 223 PINs were confirmed, with 26 being FPs, yielding a precision of 89.6%. Specifically, 22 FPs were from the correct-use rule, 2 from the triggering rule, and 2 from the behavior rule. Figure 4 illustrates the distribution of these FPs across different PIN types. We found that 4 FPs were due to inaccuracies in API privacy descriptions. For example, the `setGDPRConsent` API in Pangle is found to accept only 0 or 1 [94], but the SDK internally calls it with -1 to indicate unknown consent, triggering a false alarm regarding the correct-use rule. The remaining 22 FPs were due to consistency analysis errors by the LLM. The majority (14 out of 22) reflected an overly sensitive interpretation of privacy risks, particularly in **T2** and **T3**. For instance, while Applovin's documentation *"requires you to correctly set certain consent values"*, the LLM sometimes incorrectly flags `setHasUserConsent(false)` as a violation unless user interactions explicitly decline consent. However, this call does not constitute a privacy risk, as the processing of user personal data is constrained by it. Note that we are unable to report false negatives due to the prohibitive manual effort required to build a reasonable ground truth.

To further evaluate the PIN detector (§ 4.3), we run PINFINDER on the 1,000 app testing runs without applying enhancements (SAwS and PCTrace slices). We found that 814 potential PINs were reported, with a much lower precision of 21.4% (compared to 89.6% with enhancements). Notably, among the 223 PINs confirmed earlier, 174 were reported by PINFINDER without applying enhancements. This confirms those enhancements significantly improves the accuracy of PIN detection, with negligible impact on the coverage of PINs.

PINFINDER with GPT-4o vs. Deepseek-V3. To evaluate PINFINDER's reproduciblity across differtent LLMs, besides running PINFINDER with GPT-4o, we also ran PINFINDER with DeepSeek-V3 [95] on the 1,000 sampled testing runs. Specifically, DeepSeek-V3 detected 265 potential PINs, with 228 being actual PINs, yielding a precision of 86.0%, slightly lower than GPT-4o's 89.6%. Of the PINs, 215 were detected by both LLMs, 8 only by GPT-4o, and 13 only by DeepSeek-V3. Using the two different LLMs, we did not observe a significant difference in the number of PINs across different PIN types. Therefore, PINFINDER running with different LLMs (such as GPT-4o and DeepSeek-V3) demonstrates varying trade-offs in precision and coverage when detecting PINs. However, the overall effectiveness remains similar, indicating the robustness of the privacy-contextual consistency analysis paradigm and its minimal reliance on a specific LLM.

Intra-LLM Variability Test. To evaluate the stability of PINFINDER under repeated executions of an LLM, we ran the entire PINFINDER pipeline with GPT-4o on the 1,000 sampled testing runs five times. Across these five runs, 208 of the 223 total detected PINs were consistently reported in every run, yielding a stability score of 93.3%. This result suggests that PINFINDER is stable and robust, with only limited variability in its detection results.

PIN Coverage Compared to Prior Studies. The 223 confirmed PINs cover 15 types in the taxonomy (§ 3.1). Figure 4 shows the distribution across PIN types. The most common PIN type is T15 (SDK APIs failing to limit data collection) with 48 identified issues, followed by T9 (Misuse of APIs that make privacy regulations applicable) with 31 issues. The least identified PIN types are T1 (Invalid API invocation order) and T11 (Missing APIs for opting out of data sharing/selling), with only 2 PINs identified each. T4 (Passing un-anonymized PII to SDK API) is the only one type not covered by PINFINDER, which we believe is due to the less common use of such APIs across the 1,000 test

runs. Table 6 in Appendix A provides additional examples of detected PINs.

By detecting 15 types of PINs, PINFINDER achieves a type-level coverage of 93.8% (15 out of 16), demonstrating the potential of PINFINDER in systematizing PIN detection. We were unable to conduct a head-to-head comparison with most prior tools due to their limited availability. However, our earlier efforts in building the taxonomy of PINs (§ 3.1) through empirical tool analysis enable us to approach a tentative comparison of the types of PINs covered by different tools. For instance, we observed that Zhang et al. [2], Du et al. [13], and Reyes et al. [11] may cover 8 (50.0%), 2 (12.5%), and 4 (25.0%) types of PINs, respectively. These findings seem to suggest that, relative to prior studies that rely on prescribed patterns, PINFINDER may have the potential to cover a broader range of PINs, possibly due to its detection approach based on the privacy-contextual consistency model. Beyond PIN types, we obtained 40 apps from [2], 13 of which were confirmed to have noncompliance issues. PINFINDER was able to successfully flag PINs in 11 of the 13 apps, with no false alarms. PINFINDER did not report PINs in two of the apps: one could not run on our test device due to compatibility issues, and the relevant SDKs in the other were not initialized during execution. However, PINFINDER identified an additional app with PINs, i.e., an app that allows users to listen to the Holy Qur'an, which invokes `setTagForUnderAgeOfConsent(-1)` despite being aware that the user is a child (**T13**). While this does not constitute a comprehensive comparison, we tentatively conclude that, for the types of PINs identifiable by some prior tools, PINFINDER offers a comparable level of detection.

6. PINs in the Wild

6.1. Landscape

Use of SDKs and *PSAPI*s. Out of the 4,683 D_a apps, 4,343 (92.7%) integrate at least one D_s SDK, with a total of 59 unique SDKs integrated. The most commonly integrated SDKs are Firebase (3,921 apps), Google Ad Manager (2,957 apps), and Facebook Audience Network (2,680 apps), while the least integrated are AppBrain (1 app), Adbrix (1 app) and Splunk MINT (2 apps). On average, each app integrates 8.1 SDKs. Running the 4,343 apps reveals that 1,599 (36.8%) apps invoke at least one *PSAPI*, resulting in a total of 9,714 invocations covering 125 of the 224 unique *PSAPI*s. Table 3 provides statistics on the use of SDKs and *PSAPI*s.

Upon checking the stack traces of the invoked *PSAPI*s, we found that a significant portion (9,625, 90.5%) of the invocations were initiated by other SDKs in the integration chain, while only 1,015 (9.5%) were directly initiated by the app code. These *PSAPI* invocations from other SDKs – largely opaque to app developers – pose significant challenges for maintaining app privacy compliance (as seen in the motivating example), highlighting the need for a detection tool similar to ours that enhances developer utility by not only identifying the presence of PINs but also explaining their causes. Moreover, we observed that child-directed apps integrate significantly fewer D_s SDKs. Specifically, only 69.2% of apps in the `S-Child` setup integrate at least one D_s SDK, compared to over 90% of apps in the `S-EU` and `S-CA` setups, potentially suggesting that developers of child-directed apps are more privacy-cautious for adopting third-party SDKs.

TABLE 3: Overall app and *PSAPI* statistics

Setup	Apps	Apps That		*PSAPI* Invokes		
		Integrate SDKs	Invoke *PSAPIs*	Total	Invoked by App Code	Invoked by Other SDKs
`S-EU`	4,641	4,302	1,276	5,005	516	4,489
`S-CA`	4,683	4,343	1,278	5,351	472	4,879
`S-Child`	201	139	46	284	27	257
Subtotal	4,683	4,343	1,599	10,640	1,015	9,625

Overview of PINs. Table 4 shows a breakdown of the PINs identified in the three setups. In total, 882 (18.8%) apps were reported to contain PINs that violate one or more of the three rules. Specifically, 542 (11.6%), 248 (5.3%), and 250 (5.3%) apps have PINs violating the correct-use rule, behavior rule, and triggering rule, respectively. In terms of *PSAPI* invocations, 1,020 (9.6%) and 515 (4.8%) result in PINs violating the correct-use and behavior rules, respectively, while 346 invocations are missing, violating the triggering rule. Also, in line with the *PSAPI* usage data, most (88.1%) of the PINs stem from SDK integrations performed by other SDKs rather than the app code itself, which highlights the urgency of analyzing app privacy risks from a comprehensive, supply chain perspective. Furthermore, the integration of 28 (65.1%) ad SDKs was found to be affected by PINs, compared to 14 (58.3%) for analytics SDKs. PINs also vary slightly across app categories, with game apps being more likely to be affected than others, such as finance and shopping apps.

> Finding 1: 18.8% of apps contain at least one PIN, with the majority (88.1%) originating from SDK integrations by third-party SDKs, rather than from the app's own SDK integrations.

Notably, although many developers of child-directed apps avoid integrating third-party SDKs (as previously noted), those that do are more likely to have PINs compared to other apps, i.e., 29.4% compared to 11.6% and 10.3% for apps in the `S-EU` and `S-CA` setups, respectively. We believe this difference is mainly driven by the stricter privacy requirements of GPFP and COPPA for handling children's data, such as the prohibition of data collection from children, which many SDKs struggle to meet even when their *PSAPI*s are used to configure the SDKs.

> Finding 2: On average, child-directed apps integrated fewer SDKs; however, those that did integrate SDKs were found to contain more PINs, particularly related to children's data collection.

TABLE 4: Overall statistics of PINs

Setup	PINs Violating Correct-Use Rule				PINs Violating Behavior Rule				PINs Violating Triggering Rule[§]				SubTotal
	Apps	*PSAPIs*	Invokes by App Code	Invokes by Other SDKs	Apps	*PSAPIs*	Invokes by App Code	Invokes by Other SDKs	Apps	*PSAPIs*	Invokes by App Code	Invokes by Other SDKs	Apps
`S-EU`	256 (5.5%)	405 (8.1%)	28 (5.4%)	377 (8.4%)	175 (3.8%)	233 (4.7%)	34 (6.6%)	199 (4.4%)	186 (4.0%)	224	73	151	539 (11.6%)
`S-CA`	373 (8.0%)	549 (10.3%)	28 (5.9%)	521 (10.7%)	144 (3.0%)	214 (4.0%)	22 (4.7%)	192 (3.9%)	76(1.6%)	97	16	81	483 (10.3%)
`S-Child`	30 (14.9%)	66 (23.2%)	10 (37.0%)	56 (21.8%)	32 (16.0%)	68 (23.9%)	10 (37.0%)	58 (22.6%)	21 (10.4%)	25	0	25	59 (29.4%)
Subtotal	542 (11.6%)	1020 (9.6%)	69 (6.8%)	951 (9.9%)	248 (5.3%)	515 (4.8%)	66 (6.5%)	449 (4.7%)	250 (5.3%)	346	89	257	882 (18.8%)

[§] The percentage for *PSAPI* invocations could not be reported, as these PINs indicate missing *PSAPI* invocations.

6.2. PINs Violating Correct-Use Rule

256 (5.5%), 373 (6.3%), and 30 (14.9%) apps in the `S-EU`, `S-CA`, and `S-Child` setups, respectively, are reported to contain PINs violating the correct-use rule. The high percentage of PINs in child-directed apps is primarily due to the absence of parental consent before configuring the consent (or opt-in) status for an SDK (**T2**). Overall, the PINs stem from the integration of 25 unique D_s SDKs, involving 49 *PSAPI*s. In particular, the invocation of certain *PSAPI*s is more likely to cause PINs, with 20 *PSAPI*s triggering PINs at least 20% of the time. Analyzing these 20 *PSAPI*s, we found that 7 are related to configuring US privacy strings for CCPA compliance, 5 to managing user consent, and 5 to child-specific treatment, while the remaining *PSAPI*s control the use of identifiers, event tracking, logging, and etc.

> Finding 3: Child-directed apps often configure SDK consent status before obtaining consent. Various *PSAPI*s are used incorrectly, with US privacy string configuration APIs being the most common.

The integration of certain D_s SDKs is highly likely to be affected by PINs, notably TradPlus (60.0%), Startapp (40.0%), and MobileFuse (35.3%). We reviewed the documents of these SDKs and the apps integrating them to identify potential causes. This revealed two factors that may lead to these PINs: *poor documentation quality* and *confusing PSAPI design*. For instance, the MobileFuse document [96] provides much shorter descriptions of its *PSAPI*s compared to other SDKs and also includes demo code with hard-coded parameters that lower down the privacy protection, such as "`setSubjectToCoppa(false)`," which disables child-directed treatment without verifying the user's age. TradPlus [97], on the other hand, defines *PSAPI* parameters in a counterintuitive and confusing way. For example, passing `true` to its `setCCPADoNotSell(boolean)` method results in "*accept reported data*," whereas passing `false` means "*California users do not report data*." Moreover, passing `1` to the `setLGPDConsent(int)` API indicates "*user decline*," while `0` indicates "*user consent*." These parameters convey semantically opposite meanings compared to similar *PSAPI*s in other D_s SDKs, which results in two-thirds of the apps invoking these *PSAPI*s being reported for PINs.

> Finding 4: Potential causes of incorrect *PSAPI* use include poor privacy documentation (with privacy-irresponsible demo code) and counterintuitive, confusing *PSAPI* design.

Case Study: App Developers' Self-Deceiving *PSAPI* Invocations. *Anonymized VPN*, with over 1 million installs, was reported to contain 8 PINs violating the correct-use rule. We found that the app developer either passes hard-coded parameters with unmet presuppositions or uses self-deceiving values not defined in the SDK documents. For example, when integrating Digital Turbine, the app invokes the `setGdprConsent` API with a `true` parameter without obtaining user consent (**T2**). Similarly, for the `setUSPrivacyString` API, which accepts strings like `1YNN`, the app uses an invalid parameter (**T6**), "`myUSPrivacyString`." It also invokes the `setGdprConsentString` API with a parameter "`myGdprConsentString`" (**T2**). Calling these *PSAPI*s with these parameters fails to meaningful privacy assurance to users. Hence, we reported the issues to the developer, and they are working on a fix. Similar PINs were found in five other apps, including *Anonymous Nails*, which uses a version number (i.e., "`8.1`") as a consent string (**T2**).

Case Study: Logical Flaws in SDK Mediation. *Anonymized Office Viewer*, an app with over 100 million installs, integrates AD(X) for monetization. Similar to the motivating example, AD(X) further integrates Unity Ads, forming an integration chain of *App-AD(X)-Unity*. A PIN was reported in the app for invoking the Unity Ads *PSAPI* `MetaData.set("gdpr.consent", Boolean.TRUE)` without first obtaining user consent (**T2**). Our analysis reveals that this PIN stems not from the app code but from a logical flaw in AD(X)'s implementation. Specifically, when consent requests are not sent or users have not expressed consent, AD(X) sets the user consent status to `ADXConsentStateUnknown`. However, the unknown consent status in AD(X) is not properly communicated to Unity Ads. As long as consent has not been declined, AD(X) signals to Unity Ads that consent has been obtained. Consequently, Unity Ads displays ads to users who have not provided consent. We reported this issue but have not received a response.

6.3. PINs Violating Behavior Rule

We identified 248 apps containing PINs that violate the behavior rule across 23 D_s SDKs. These PINs typically occur when the expected behaviors of the *PSAPI*s define specific obligations or restrictions (e.g., on data collection), but the SDKs' subsequent actions fail to adhere to these requirements, mostly due to the implementation flaws of the SDKs. Furthermore, the use of seven *PSAPI*s in apps has been found to have over a 20% likelihood of causing a PIN. These *PSAPI*s cover a range of functionalities, such as setting up US privacy strings, configuring child-directed treatment, and tracking in-app events. For example, the API with the highest likelihood is the `setIabUsPri-`

vacyString API of the Digital Turbine SDK. This API is expected to opt users out of data sharing or sale when the parameter "*Y" is passed, which, however, is violated because data sharing still occurs in the SDK (**T16**).

> Finding 5: *PSAPI*s often fail to restrict data processing as documented, with analytics SDKs affecting the most apps.

Case Study: Misalignment Between SDK Documents and Implementations. *Anonymized Kids Learning* uses the Adjust SDK for analytics. To comply with GPFP requirements, the app calls Adjust's setPlayStoreKidsAppEnabled(true) API, which the SDK documentation claims will "*prevent the SDK from accessing device and advertising IDs*" [98]. In practice, however, the API only prevents the collection of the AppSetId [99], and the SDK still obtains advertising IDs via the Settings$Secure system class and sends them to *https://app.adjust.com* as the fire_adid parameter (**T15**). This occurs in at least 11 types of in-app analytics events, including attribution, measurement, ad revenue, and subscription tracking. Due to app developers' limited visibility into the SDK implementation, the misalignment between SDK documentation and actual implementations can easily go unnoticed, potentially misleading developers about their app's compliance status. Similar PINs caused by Adjust were found in another 3 apps, for which we reported the details to the app developers and the Adjust SDK team.

6.4. PINs Violating Triggering Rule

PINs violating triggering rule occur when *PSAPI*s of SDKs should have been invoked but were not. Specifically, 186 (4.0%), 76 (1.6%), and 21 (10.4%) apps tested in the S-EU, S-CA, and S-Child setups, respectively, were found to contain such PINs, leading to missing *PSAPI* invocations in a total of 250 (5.3%) apps. These missing *PSAPI* invocations reflect the app's failure to meet various privacy requirements, such as not calling setConsentData in AppsFlyer to record an EU user's consent status (**T10**) or failing to configure Facebook Ads' audiences using the setMixedAudience API (**T13**). In total, 89 (25.7%) missing *PSAPI* invocations should have been added to improve the app's SDK integration, while 257 (74.3%) should have been added to enhance the integration of SDKs by other SDKs. As with other types of PINs, child-directed apps are more likely to be affected due to stricter privacy requirements imposed on these apps.

> Finding 6: More *PSAPI*s should be added to improve SDK integrations, with the majority (74.3%) of the effort expected from SDKs that integrate other SDKs.

Case Study: Literal Meanings of *PSAPI* vs. Privacy Implications. PINFINDER reported that the *Anonymized Cartoons* app should have invoked the MetaData.set(...) API of the Unity Ads SDK with the parameter combination ["user.nonbehavioral", "true"] but failed to do so (**T13**). An analysis of the app reveals that it uses an *App-Appodeal-Unity Ads* integration chain, similar to the motivating example. When tested in the S-Child setup, we found that Appodeal is properly configured for child-directed treatment, such as stopping the serving of personalized ads. However, Appodeal passes the child status to Unity Ads by calling its MetaData.set(...) API with the ["privacy.useroveragelimit", "false"] parameter combination, assuming this would trigger child-directed treatment, as this parameter combination is explicitly linked to user age. Unfortunately, this parameter only controls the display of an age gate, while the user.nonbehavioral parameter is the one that triggers child-directed treatment in Unity Ads, a distinction that the Appodeal developer failed to recognize. As a result, even though the user indicates they are a child, Unity Ads continues to serve personalized ads to them. This issue highlights the need to *standardize PSAPIs with clear privacy implications* to ensure that the use of multiple SDKs within an app leads to consistent privacy assurances. We found that five other child-directed apps in D_a are affected by the same PIN. We reported this issue to Appodeal and the app developers (which is currently pending confirmation).

> Finding 7: Apps using multiple SDKs may encounter some inconsistencies in privacy assurance across these SDKs, which calls for greater standardization of *PSAPI*s with clear privacy implications.

7. Discussion

Limitations. The significance of the new paradigm lies in its potential to enhance the utility for developers in detecting privacy non-compliance in modern software. However, the prototype implementation, PINFINDER, represents only the first step in validating the paradigm, with both its scope and technical design being limited.

- **Platform Limitation**. The implementation and evaluation were conducted exclusively on Android platforms. The combination of app analysis and LLMs was only designed to address challenges encountered specifically within the Android domain and the SDKs present in the dataset. We leave the extension to other platforms as a future direction (as will be discussed later).
- **SDK Version Mismatch between Documentation and Code**. PINFINDER does not explicitly check the version match between SDK documentation and the SDKs actually integrated into apps, which could potentially threaten the validity of the identified PINs. We relied on two implementation choices to reduce version variance: (1) we collected the most recent apps and corresponding SDK documentation, which helps narrow SDK version gaps; and (2) we evaluated SDK APIs in code only when they exactly match the documentation, which serves as another indicator that the SDK versions in code are close to those in the documentation. However, we acknowledge that these choices do not guarantee an exact SDK version match, which is a

limitation that requires more careful analysis and treatment. To further reduce potential version mismatches, we believe that PINFINDER could be integrated with widely available SDK version detection tools [100], [101] to ensure alignment between documentation and code.

• **Disadvantages of Dynamic Analysis**. PINFINDER extracts app execution traces using dynamic analysis, which suffers from coverage issues (e.g., in terms of UI events and network triggers). Thus, our results should be interpreted as a lower bound of PINs observable during the analysis. Our rationale for choosing dynamic analysis is that, although static analysis may offer broader coverage, dynamic analysis is essential for confirming PINs by providing concrete runtime evidence. We expect that these coverage issues may become less salient in some important scenarios or with advances in testing tools. For example, in settings where developers self-assess their apps, they are often able to trigger a broad range of app functionalities and benefit from the reports generated by PINFINDER; with recent advances in app analysis, such as LLM-guided execution and UI exploration [102], [103], it is becoming more feasible to exercise more app functionalities and reduce coverage gaps.

Future Directions and Deployment. Future research should investigate how the paradigm can be extended to other platforms and identify principled designs that address challenges across these platforms. In particular, while the detection rules may generalize across platforms, applying PINFINDER to different platforms such as iOS, Web, or agentic systems would require platform-specific analysis and understanding, such as analyzing SDK documentation for each platform and selecting appropriate runtime monitoring tools to collect API invocations and privacy-context traces. Moreover, in the measurement section, we run PINFINDER on apps already published in the marketplace. We envision the detection paradigm being applicable beyond this, particularly in enabling privacy-compliant software development from the perspectives of both app and SDK developers. Our next step will involve facilitating the integration of this detection paradigm into developers' toolchains.

8. Related Work

Detecting Privacy Risks with SDKs. Prior studies have identified various types of privacy risks in software, such as missing consent notices [5], [51], [104], [105], [106], [107], invalid consent (e.g., not freely given, ambiguous) [5], [8], [51], [106], [108], [109], ineffective opt-out mechanisms [2], [13], [14], and violations involving the collection of children's data [2], [11]. Among these, multiple studies highlight third-party SDKs as a major source of privacy risks, i.e., PINs, and propose detection methods [2], [11], [13], [15], [105]. For instance, Zhang et al. [2] used expert-derived "privacy violation patterns" detectable via combined static and dynamic analysis. Du et al. [13] assessed opt-out effectiveness using data-flow analysis, based on the assumption that a valid opt-out must influence data collection through control or data-flow. Reyes et al. [11] targeted COPPA violations by checking for child-specific flags or data. While effective, these methods are constrained by their reliance on prescribed patterns, and fail to generalize to broad PIN types (as demonstrated in § 5.2). As SDKs proliferate and integration methods diversify, new, previously undocumented PINs may arise, further limiting existing detection methods. To support developers in improving SDK integrations [4] and aid regulators in identifying violations [20], there is a growing need for a generalizable detection method - one that we introduce in this work.

9. Conclusion

In this study, we propose a novel paradigm based on privacy-contextual consistency analysis as a more systematic and developer-friendly solution for detecting privacy-noncompliant SDK integrations (PINs) within apps. To validate this paradigm, we take the first step by constructing a taxonomy of PINs, developing API privacy descriptions and a set of contextual consistency rules, and prototyping an automated framework, called PINFINDER, to realize the paradigm in Android apps. Our evaluation and measurement on real-world apps demonstrate that PINFINDER effectively identifies PINs caused by SDK integrations, achieving significantly broader coverage than prior studies and uncovering lesser-known issues along with their root causes.

Acknowledgments

The authors would like to thank the anonymous reviewers and shepherd for their insightful and constructive feedback. The authors are supported in part by the UCF Seed Funding Program and NSF Awards No. 2520321 and 2320974.

Ethics considerations

All data used in this study, including SDK privacy guidance documents, SDK code archives, app metadata, and app code, is publicly accessible and released by SDK and app developers through their websites, public code repositories, and app marketplaces. We have taken special care to ensure that our research practices adhere to ethical standards. Specifically, we assessed the SDKs and apps in controlled experimental environments without involving any personal data from real users or the authors. Additionally, we recognize the importance of timely reporting and communication of detected privacy risks to relevant stakeholders. Hence, we have reported all identified PINs to the relevant stakeholders, such as app and SDK developers, via email, providing both detailed descriptions and technical explanations of the risks. While most of the stakeholders remained silent, some – particularly those with the most users – have acknowledged the PINs and are actively working to resolve (or have already resolved) them, including the developers of `iFunny` (resolved in March 2025), *Anonymized VPN*, and *Anonymized Kids Learning*. To prevent any unintended

harm, such as reputational damage to stakeholders during their investigation and resolution, we have anonymized most of the app names, except for the motivating example where the issue has already been addressed.

LLM usage considerations

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality. The authors did not use LLMs to generate any ideas used in the study. The automated framework, PinFinder, leverages LLMs to identify privacy-noncompliant SDK integrations, which makes it difficult to precisely reproduce the results. To address this limitation, we implemented the prototypes of PinFinder using different LLMs, such as GPT-4o and DeepSeek-V3, and evaluated their effectiveness. The results suggest that the methodology of PinFinder is not tied to a specific LLM and maintains similar effectiveness across different models. More details are provided in § 5.

References

[1] A. Vakulov, "Google play's security: 2.36 million apps blocked for violations in 2024," *Forbes*, 2025. [Online]. Available: https://www.forbes.com/sites/alexvakulov/2025/02/02/google-plays-security-236m-apps-blocked-for-violations-in-2024/

[2] Y. Zhang, Z. Hu, X. Wang, Y. Hong, Y. Nan, X. Wang, J. Cheng, and L. Xing, "Navigating the privacy compliance maze: Understanding risks with {Privacy-Configurable} mobile {SDKs}," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 6543–6560.

[3] I. Arkalakis, M. Diamantaris, S. Moustakas, S. Ioannidis, J. Polakis, and P. Ilia, "Abandon all hope ye who enter here: A dynamic, longitudinal investigation of android's data safety section," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5645–5662.

[4] N. Alomar and S. Egelman, "Developers say the darnedest things: Privacy compliance processes followed by developers of child-directed apps," *Proceedings on Privacy Enhancing Technologies*, 2022.

[5] T. T. Nguyen, M. Backes, and B. Stock, "Freely given consent? studying consent notice of third-party tracking and its violations of gdpr in android apps," in *CCS 2022*, 2022, pp. 2369–2383.

[6] K. Kollnig, A. Shuba, R. Binns, M. Van Kleek, and N. Shadbolt, "Are iphones really better for privacy? comparative study of ios and android apps," *arXiv preprint arXiv:2109.13722*, 2021.

[7] X. Zhang, X. Wang, R. Slavin, T. Breaux, and J. Niu, "How does misconfiguration of analytic services compromise mobile privacy?" in *ICSE 2020*, 2020, pp. 1572–1583.

[8] C. Matte, N. Bielova, and C. Santos, "Do cookie banners respect my choice?: Measuring legal compliance of banners from iab europe's transparency and consent framework," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 791–809.

[9] M. Zhang, W. Meng, Y. Zhou, and K. Ren, "Cschecker: revisiting gdpr and ccpa compliance of cookie banners on the web," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[10] M. A. B. Aziz and C. Wilson, "Johnny still can't opt-out: Assessing the iab ccpa compliance framework," *Proceedings on Privacy Enhancing Technologies*, 2024.

[11] I. Reyes, P. Wijesekera, J. Reardon, A. Elazari Bar On, A. Razaghpanah, N. Vallina-Rodriguez, S. Egelman *et al.*, """won't somebody think of the children?" examining coppa compliance at scale," in *PETS 2018*, 2018.

[12] M. Tahaei, M. Ramokapane, T. Li, J. I. Hong, and A. Rashid, "Charting app developers' journey through privacy regulation features in ad networks," in *The 22nd Privacy Enhancing Technologies Symposium*. De Gruyter Open Ltd., 2022, pp. 33–56.

[13] X. Du, Z. Yang, J. Lin, Y. Cao, and M. Yang, "Withdrawing is believing? detecting inconsistencies between withdrawal choices and third-party data collections in mobile apps," in *IEEE S&P 2024*. IEEE Computer Society, 2023, pp. 14–14.

[14] D. Bui, B. Tang, and K. G. Shin, "Do opt-outs really opt me out?" in *CCS 2022*, 2022, pp. 425–439.

[15] Y. Xiao, C. Zhang, Y. Qin, F. F. S. Alharbi, L. Xing, and X. Liao, "Measuring compliance implications of third-party libraries' privacy label disclosure guidelines," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1641–1655.

[16] Z. Liu, U. Iqbal, and N. Saxena, "Opted out, yet tracked: Are regulations enough to protect your privacy?" *arXiv preprint arXiv:2202.00885*, 2022.

[17] AppCensus, "Appcensus: Privacy audits for mobile apps," https://appcensus.io/, 2025.

[18] "Data Theorem App Store Privacy Protect," https://www.datatheorem.com/solutions/app-store-privacy-protect, 2024.

[19] "NowSecure Platform," https://www.nowsecure.com/products/nowsecure-platform/, 2024.

[20] E. D. P. Supervisor, "Annual report 2020," Accessed: 2025-04-09, 2021. [Online]. Available: https://www.edps.europa.eu/system/files/2021-04/2021-04-19-annual-report-2020_EN.pdf

[21] ProAndroidDev, "Google play sdk index breakdown," https://proandroiddev.com/google-play-sdk-index-breakdown-6203000d9018, 2022.

[22] S. L. Sravanthi, M. Doshi, T. P. Kalyan, R. Murthy, P. Bhattacharyya, and R. Dabre, "Pub: A pragmatics understanding benchmark for assessing llms' pragmatics capabilities," *arXiv preprint arXiv:2401.07078*, 2024.

[23] Y. Zhu, J. R. A. Moniz, S. Bhargava, J. Lu, D. Piraviperumal, Y. Zhang, H. Yu, and B.-H. Tseng, "Can large language models understand context?" *arXiv preprint arXiv:2402.00858*, 2024.

[24] "Pinfinder supplementary materials website," https://sites.google.com/view/pinfinder/home, 2025.

[25] Sweet Maker Shop, "Mommy baby care nursery," https://play.google.com/store/apps/details?id=com.sms.caregame, 2024.

[26] "Google play families policies," https://support.google.com/googleplay/android-developer/answer/9893335?hl=en, 2024.

[27] "Children's online privacy protection rule (coppa)," https://www.ftc.gov/legal-library/browse/rules/childrens-online-privacy-protection-rule-coppa.

[28] "Children's online privacy protection rule ("coppa")," https://www.ftc.gov/legal-library/browse/rules/childrens-online-privacy-protection-rule-coppa, 2025.

[29] AppLovin, "Max android advanced settings," https://developers.applovin.com/en/max/android/overview/advanced-settings/, 2025.

[30] "ironsource — turning apps into scalable businesses," https://www.is.com/, 2024.

[31] I. Reyes, P. Wijesekera, A. Razaghpanah, J. Reardon, N. Vallina-Rodriguez, S. Egelman, C. Kreibich *et al.*, "" is our children's apps learning?" automatically detecting coppa violations," in *Workshop on Technology and Consumer Protection (ConPro 2017), in conjunction with the 38th IEEE Symposium on Security and Privacy (IEEE S&P 2017)*, 2017.

[32] SpyBoy, "Is your advertising id a privacy risk? the hidden dangers of mobile ad tracking," https://spyboy.blog/2024/10/25/is-your-advertising-id-a-privacy-risk-the-hidden-dangers-of-mobile-ad-tracking/, 2024.

[33] M. Gruber, C. Höfig, M. Golla, T. Urban, and M. Große-Kampmann, ""we may share the number of diaper changes": A privacy and security analysis of mobile child care applications," *Proceedings on Privacy Enhancing Technologies*, 2022.

[34] Á. Feal, P. Calciati, N. Vallina-Rodriguez, C. Troncoso, and A. Gorla, "Angel or devil? a privacy study of mobile parental control apps," *Proceedings on Privacy Enhancing Technologies*, 2020.

[35] S. Li, Z. Yang, Y. Nan, S. Yu, Q. Zhu, and M. Yang, "Are we getting well-informed? an in-depth study of runtime privacy notice practice in mobile apps," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1581–1595.

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

[37] D. Rodriguez, J. A. Calandrino, J. M. Del Alamo, and N. Sadeh, "Privacy settings of third-party libraries in android apps: A study of facebook sdks," *Proceedings on Privacy Enhancing Technologies*, 2025.

[38] N. Samarin, A. Sanchez, T. Chung, A. D. B. Juleemun, C. Gilsenan, N. Merrill, J. Reardon, and S. Egelman, "The medium is the message: How secure messaging apps leak sensitive data to push notification services," *arXiv preprint arXiv:2407.10589*, 2024.

[39] "Ironsource's privacy api document," https://developers.is.com/ironsource-mobile/android/regulation-advanced-settings/#step-1, 2024.

[40] K. Zhao, X. Zhan, L. Yu, S. Zhou, H. Zhou, X. Luo, H. Wang, and Y. Liu, "Demystifying privacy policy of third-party libraries in mobile apps," in *ICSE 2023*. IEEE, 2023, pp. 1583–1595.

[41] D. Liu, Y. Xiao, C. Zhang, K. Xie, X. Bai, S. Zhang, and L. Xing, "{iHunter}: Hunting privacy violations at scale in the software supply chain on {iOS}," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 5663–5680.

[42] Z. Tan and W. Song, "Ptpdroid: Detecting violated user privacy disclosures to third-parties of android apps," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 473–485.

[43] S. A. Horstmann, S. Domiks, M. Gutfleisch, M. Tran, Y. Acar, V. Moonsamy, and A. Naiakshina, ""those things are written by lawyers, and programmers are reading that." mapping the communication gap between software developers and privacy experts," *Proceedings on Privacy Enhancing Technologies*, 2024.

[44] L. Yu, T. Zhang, X. Luo, and L. Xue, "Autoppg: Towards automatic generation of privacy policy for android applications," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2015, pp. 39–50.

[45] S. Zimmeck, R. Goldstein, and D. Baraka, "Privacyflash pro: Automating privacy policy generation for mobile apps." in *NDSS*, vol. 2, 2021, p. 4.

[46] L. Yu, T. Zhang, X. Luo, L. Xue, and H. Chang, "Toward automatically generating privacy policy for android apps," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 4, pp. 865–880, 2016.

[47] T. Li, L. F. Cranor, Y. Agarwal, and J. I. Hong, "Matcha: An ide plugin for creating accurate privacy nutrition labels," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 8, no. 1, pp. 1–38, 2024.

[48] J. Gardner, Y. Feng, K. Reiman, Z. Lin, A. Jain, and N. Sadeh, "Helping mobile application developers create accurate privacy labels," in *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2022, pp. 212–230.

[49] S. Pan, T. Hoang, D. Zhang, Z. Xing, X. Xu, Q. Lu, and M. Staples, "Toward the cure of privacy policy reading phobia: Automated generation of privacy nutrition labels from privacy policies," *arXiv preprint arXiv:2306.10923*, 2023.

[50] Google, "Provide information for google play's data safety section," 2023. [Online]. Available: https://support.google.com/googleplay/android-developer/answer/10787469?hl=en

[51] S. Koch, B. Altpeter, and M. Johns, "The {OK} is not enough: A large scale study of consent dialogs in smartphone applications," in *USENIX Security 2023*, 2023, pp. 5467–5484.

[52] "Applovin max android privacy overview," https://developers.applovin.com/en/max/android/overview/privacy/, 2024.

[53] T. Lv, R. Li, Y. Yang, K. Chen, X. Liao, X. Wang, P. Hu, and L. Xing, "Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 1837–1852.

[54] P. Hu, R. Liang, Y. Cao, K. Chen, and R. Zhang, "{AURC}: Detecting errors in program code and documentation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1415–1432.

[55] X. Ren, X. Ye, Z. Xing, X. Xia, X. Xu, L. Zhu, and J. Sun, "Api-misuse detection driven by fine-grained api-constraint knowledge graph," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 461–472.

[56] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 27–37.

[57] H. Zhong, N. Meng, Z. Li, and L. Jia, "An empirical study on api parameter rules," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 899–911.

[58] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, "Inferring method specifications from natural language api descriptions," in *2012 34th international conference on software engineering (ICSE)*. IEEE, 2012, pp. 815–825.

[59] Y. Ma, W. Tian, X. Gao, H. Sun, and L. Li, "Api misuse detection via probabilistic graphical model," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 88–99.

[60] Y. Zhang, "Detecting code comment inconsistencies using llm and program analysis," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 683–685.

[61] Alireza Savand, "html2text," https://github.com/Alir3z4/html2text/.

[62] OpenAI, "Prompt engineering best practices for chatgpt," https://help.openai.com/en/articles/10032626-prompt-engineering-best-practices-for-chatgpt, 2025.

[63] MobileFuse, "Data Privacy," https://docs.mobilefuse.com/docs/android-sdk-data-privacy.

[64] M. Learn, "Sdk privacy for android," https://learn.microsoft.com/en-us/xandr/mobile-sdk/sdk-privacy-for-android, 2025.

[65] I. A. Bureau, "Us privacy string - ccpa," https://github.com/InteractiveAdvertisingBureau/USPrivacy/blob/master/CCPA/US%20Privacy%20String.md, 2025.

[66] LangChain, "Langchain faiss integration," https://python.langchain.com/docs/integrations/vectorstores/faiss/, 2025.

[67] "Complete guide to gdpr compliance," https://gdpr.eu, 2025.

[68] State of California - Department of Justice - Office of the Attorney General, "California consumer privacy act (ccpa)," https://oag.ca.gov/privacy/ccpa, 2025.

[69] "Langchain," https://www.langchain.com/, 2025.

[70] "Soot - a framework for analyzing and transforming java and android applications," https://soot-oss.github.io/soot/, 2023.

[71] C. Tagliaro, F. Hahn, R. Sepe, A. Aceti, and M. Lindorfer, "I still know what you watched last sunday: Privacy of the hbbtv protocol in the european smart tv landscape," in *30th Annual Network and Distributed System Security, NDSS 2023*, 2023.

[72] "Nordvpn," https://nordvpn.com/, 2025.

[73] Z. Lv, C. Peng, Z. Zhang, T. Su, K. Liu, and P. Yang, "Fastbot2: Reusable automated model-based gui testing for android enhanced by reinforcement learning," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*, 2022.

[74] H. Hu, R. Dong, J. Grundy, T. M. Nguyen, H. Liu, and C. Chen, "Automated mapping of adaptive app guis from phones to tvs," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–31, 2023.

[75] E. Ma, S. Huang, W. He, T. Su, J. Wang, H. Liu, G. Pu, and Z. Su, "Automata-based trace analysis for aiding diagnosing gui testing tools for android," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 592–604.

[76] Y. Guo, D. Wang, L. Wang, Y. Fang, C. Wang, M. Yang, T. Liu, and H. Wang, "Beyond app markets: Demystifying underground mobile app distribution via telegram," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 8, no. 3, pp. 1–25, 2024.

[77] H. Wang, Y. Fang, Y. Liu, Z. Jin, E. Delph, X. Du, Q. Liu, and L. Xing, "Hidden and lost control: on security design risks in iot user-facing matter controller," 2025.

[78] "Frida - dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers." https://frida.re/, 2024.

[79] Zhengjim, "Camille," https://github.com/zhengjim/camille/tree/83c5b23912db83d00ac0a615c89d905daa9772b5.

[80] Alleny, "Privacysentry," https://github.com/allenymt/PrivacySentry.

[81] PortSwigger, "Burp Suite," https://portswigger.net/burp.

[82] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 ways to leak your data: An exploration of apps' circumvention of the android permissions system," in *28th USENIX security symposium (USENIX security 19)*, 2019, pp. 603–620.

[83] K. A. Bamberger, S. Egelman, C. Han, A. E. Bar On, and I. Reyes, "Can you pay for privacy? consumer expectations and the behavior of free and paid apps," *Berkeley Tech. LJ*, vol. 35, p. 327, 2020.

[84] "Hyprmx sdk integration guide (android / amazon) - privacy," https://documentation.hyprmx.com/sdk-integration-guides/android-amazon/privacy, 2025.

[85] O. Press, M. Zhang, S. Min, L. Schmidt, N. A. Smith, and M. Lewis, "Measuring and narrowing the compositionality gap in language models," *arXiv preprint arXiv:2210.03350*, 2022.

[86] "Android library statistics," https://www.appbrain.com/stats/libraries, 2024.

[87] facundoolano, "google-play-scraper," https://github.com/facundoolano/google-play-scraper, 2022.

[88] "Apkpure," https://apkpure.com/ur/, 2025.

[89] P. Calciati, K. Kuznetsov, A. Gorla, and A. Zeller, "Automatically granted permissions in android apps: An empirical study on their prevalence and on the potential threats for privacy," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 114–124.

[90] A. Razagallah, R. Khoury, and J.-B. Poulet, "Twindroid: A dataset of android app system call traces and trace generation pipeline," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 591–595.

[91] OpenAI, "Openai platform – models documentation," https://platform.openai.com/docs/models, 2024.

[92] U. Technologies, "Gdpr legal documentation for unity ads," https://docs.unity3d.com/Packages/com.unity.ads@3.3/manual/LegalGdpr.html, 2023.

[93] "Regulatory Compliance - Data Theorem," https://www.datatheorem.com/solutions/compliance/, 2024.

[94] Pangle, "Initialize pangle sdk," https://www.pangleglobal.com/zh/integration/android-initialize-pangle-sdk.

[95] DeepSeek, "Deepseek-v3," https://github.com/deepseek-ai/DeepSeek-V3, 2024.

[96] MobileFuse, "Android sdk data privacy documentation," https://docs.mobilefuse.com/docs/android-sdk-data-privacy, 2025.

[97] TradPlus, "Privacy regulations," https://docs.tradplusad.com/en/docs/tradplussdk_android_doc_v6/privacy_policy/os_privacy_policy/, 2025.

[98] Adjust, "Android sdk privacy features documentation," https://dev.adjust.com/en/sdk/android/v4/features/privacy, 2025.

[99] Android Developers, "Appsetid reference documentation," 2025. [Online]. Available: https://developer.android.com/design-for-safety/privacy-sandbox/reference/adservices/appsetid/AppSetId

[100] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1695–1707.

[101] J. Zhang, A. R. Beresford, and S. A. Kollmann, "Libid: reliable identification of obfuscated third-party android libraries," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 55–65.

[102] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang, "Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[103] C. Wang, T. Liu, Y. Zhao, M. Yang, and H. Wang, "Llmdroid: Enhancing automated mobile app gui testing coverage with large language model guidance," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 1001–1022, 2025.

[104] K. Kollnig, P. Dewitte, M. Van Kleek, G. Wang, D. Omeiza, H. Webb, and N. Shadbolt, "A fait accompli? an empirical study into the absence of consent to {Third-Party} tracking in android apps," in *SOUPS 2021*, 2021, pp. 181–196.

[105] T. T. Nguyen, M. Backes, N. Marnau, and B. Stock, "Share first, ask later (or never?) studying violations of {GDPR's} explicit consent in android apps," in *USENIX Security 2021*, 2021, pp. 3667–3684.

[106] X. Hu and N. Sastry, "Characterising third party cookie usage in the eu after gdpr," in *WebSci 2019*, 2019, pp. 137–141.

[107] M. Degeling, C. Utz, C. Lentzsch, H. Hosseini, F. Schaub, and T. Holz, "We value your privacy... now take some cookies: Measuring the gdpr's impact on web privacy," *arXiv preprint arXiv:1808.05096*, 2018.

[108] C. Utz, M. Degeling, S. Fahl, F. Schaub, and T. Holz, "(un) informed consent: Studying gdpr consent notices in the field," in *CCS 2019*, 2019, pp. 973–990.

[109] D. Bollinger, K. Kubicek, C. Cotrini, and D. Basin, "Automating cookie consent and {GDPR} violation detection," in *USENIX Security 2022*, 2022, pp. 2893–2910.

Appendix A.
Figures and Tables

Figure 5 presents the “Switch to privacy account” UI of the Instagram app. Table 5 lists common phrase patterns used in SDK privacy guidance documents to express prohibitions or obligations, along with examples of how these patterns are used. Figure 6 provides the privacy description for the `setIsAgeRestrictedUser` API of the AppLovin SDK. Table 6 presents additional examples of PIN detected by PINFINDER.

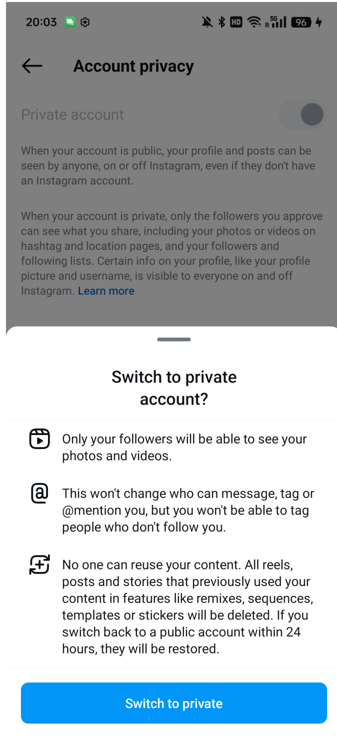

Figure 5: An example of Instagram UI

```
{
   API_name :  setIsAgeRestrictedUser(boolean) ,
   class_name :  com.applovin.sdk.AppLovinPrivacySettings
       ,
   configurations : [
    {
       parameters : [true],
       correct_use_conditions : [
         Before initializing or using the AppLovin SDK
      ],
       triggering_conditions : [
         The app needs to comply with COPPA, or other age
             -related requirements ,
         The user is under the age of 16, or defined by
             applicable laws as age-restricted
      ],
       behaviors : [
         Forbid the collection of personal information
             from children ,
         Forbid the serving of targeted ads to children
      ]
    },
    {
       parameters : [false],
       correct_use_conditions : [
         The user is confirmed to be over 16 and not to
             be in an age-restricted category .
      ],
       triggering_conditions : [],
       behaviors : []
    }
  ]
}
```

Figure 6: Privacy description for AppLovin’s `setIsAgeRestrictedUser` API

TABLE 5: Common phrase patterns to express obligations and prohibitions

	Phase Pattern	Example	Example after Semantic Synthesis
Prohibitions	[be | will | can | do] not	User's personal information will not be used for advertising	Forbid user's personal information for advertising purposes.
	no...will, no longer	Will no longer receive data from this user	Forbid the SDK from receiving data from this user
	disable, disallow, prevent, prohibit, reject, deny, avoid, opt-out, restrict	Disable third-party sharing	Forbid third-party sharing
	stop, cease, pause, turn off	Stop Adjust from sharing user data with third parties	Forbid Adjust from sharing user data with third parties
	only...will	Only contextual ads will be shown to the user	Forbid personalized ads from being shown to the user
	all...[not | non-]	All ads served will be non-personalized	Forbid ads served being personalized
Obligations	required	Consent is required to track data and send requests	Must obtain consent to track data and send requests
	must/have to	Must obtain user consent before requesting any ads from the Exchange server	Must obtain user consent before requesting any ads from the Exchange server

TABLE 6: Examples of PINs detected by PINFINDER

Consistency Rule	PIN Type	Example of Detected PIN
Correct-use Rule	**T1:** Invalid API invocation order	AppLovin SDK requires user consent to be set before initialization, but `setHasUserConsent` was invoked after the SDK was initialized.
	T2: SDK usage triggering data collection without user consent	`setHasUserConsent(true)` was called before the appearance of any consent-related UIs.
	T3: SDK usage triggering data collection despite user rejection	The user clicked "Decline All" for data usage options, but `setConsent(true)` was still invoked.
	T5: Misuse of APIs that limit data collection	`setAutoLogAppEventsEnabled(true)` was called despite the user opting out of data collection.
	T6: Misuse of APIs that opt out of data sharing/selling	`setDoNotSell(false)` was invoked even though the user had opted out of data selling.
	T7: Misuse of APIs that limit data use for ad tracking	The user disallowed the use of the advertising identifier for tracking, but `setAppLimitAdTracking(false)` was still called.
	T8: Misuse of APIs that flag child users	`setTagForChildDirectedTreatment(0)` was invoked even though the user is a child.
	T9: Misuse of APIs that make privacy regulations applicable	`setSubjectToGDPR(false)` was called despite the user being subject to GDPR as a user of the European Union.
Triggering Rule	**T10:** Missing APIs for limiting data collection	Although the EU user declined data collection, `setGdprConsent` was not called to reflect this preference.
	T11: Missing APIs for opting out of data sharing/selling	Although the user opted out of the sale of their personal information by selecting "*Do Not Sell My Data*," the `setDoNotSell` method was not called accordingly.
	T12: Missing APIs for limiting data use for ads tracking	`setAdvertiserTrackingEnabled` was not invoked for a user who opted out of interest-based advertising tracking.
	T13: Missing AP for flagging child users	`setTagForChildDirectedTreatment` was not invoked, even though the user is a child and the app targets children.
	T14: Missing APIs for making privacy regulations applicable	`setSubjectToGDPR` was not invoked for a user who falls under the scope of the GDPR in the EU.
Behavior Rule	**T15:** SDK APIs failing to limit data collection	Although `setIsAgeRestrictedUser(true)` was invoked to indicate that the user is a child, personal data such as the Advertising ID (AAID) continues to be collected.
	T16: SDK APIs failing to opt out of data sharing or selling	`setUSPrivacyString("1YY-")` was set to reflect the user's opt-out from data sharing, but `setDoNotSell(false)` was later invoked indicating that data sharing activities continued.

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper introduces Privacy-Noncompliant SDK Integrations (PINs) to capture privacy violations caused by third-party SDKs in mobile apps and argues that existing approaches relying on network traffic analysis and predefined rules are insufficient. To address this, the research work proposes PINFINDER, an automated framework that performs privacy-contextual consistency analysis to detect violations and identify the specific SDK APIs responsible.

B.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research.
- Creates a New Tool to Enable Future Science.
- Provides a Valuable Step Forward in an Established Field.

B.3. Reasons for Acceptance

1) While many existing approaches rely solely on LLMs to extract privacy descriptions for non-compliance analysis, this work introduces additional strategies that strengthen the modeling process and improve reliability.
2) Instead of relying on pattern-based detection, the paper proposes a context-aware analysis framework, which is a significant contribution since context plays a crucial role in SDK-related privacy violations. For example, the paper shows that when the UI identifies a user as a child, it may trigger an SDK API that collects data resulting in a policy violation. In contrast, the same SDK API behaves benignly under different contexts.
3) The study analyzes 64 SDK APIs used across 4,683 real-world mobile apps, many of which involve deep SDK dependency chains. The results reveal that 18.8% of the apps contain at least one PIN, and notably 88.1% of these violations originate from third-party SDKs rather than the app's own code, highlighting the systemic risks introduced by SDK dependencies.