PriAgent: A Collaborative Multi-Agent Framework for Auditing Android Privacy Compliance

Ziwei Zhang[1,2], Zhao Li[1,2*], Zhuojun Jiang[1,2], Jiangyi Yin[1,2], Xuebin Wang[1,2], Jiangchao Chen[1,2], Qingyun Liu[1,2]

[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
{zhangziwei, lizhao, jiangzhuojun, yinjiangyi, wangxuebin, chenjiangchao, liuqingyun}@iie.ac.cn

Abstract

Stringent regulations like General Data Protection Regulation (GDPR) mandate that an application's code-level data handling must align with its natural-language privacy policy, creating a critical auditing challenge. However, existing methods, predominantly reliant on static analysis, suffer from a critical limitation: in their pursuit of soundness via over-approximation, they exhibit "semantic blindness"—detecting what data flows exist but not why. This leads to an overwhelming volume of false positives, rendering automated auditing impractical. To bridge this gap, we introduce PriAgent, a novel framework that approaches compliance auditing as a multi-stage, AI-driven reasoning task. Instead of a monolithic model, PriAgent deploys a team of specialized agents that execute a divide-and-conquer strategy. They systematically prune the analysis space by abstracting data flows, pinpoint semantic loci critical for inspection, and perform on-demand summarization of large code blocks to ensure scalability. PriAgent leverages Retrieval-Augmented Generation (RAG) with a curated knowledge base of Android APIs, equipping agents to discern potentially non-compliant behavior from benign functionality. By correlating code-level evidence with the app's stated privacy policy, PriAgent delivers a holistic and explainable verdict for each potential violation. Our evaluations demonstrate that PriAgent significantly reduces false positives, enabling a more scalable and precise compliance audit.

Code — https://github.com/pharaoh1024/PriAgent.git

Introduction

The proliferation of mobile applications has led to unprecedented personal data collection (Gamba et al. 2023), prompting stringent regulations like the General Data Protection Regulation (European Union 2018) and California Consumer Privacy Act (State of California Department of Justice 2020). These laws mandate that an app's code-level data practices must align with its natural-language privacy policy. However, a profound semantic gap separates the natural-language assertions in a policy from the executable logic of an application, making automated compliance auditing remain a major challenge in software assurance.

*Corresponding author
Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

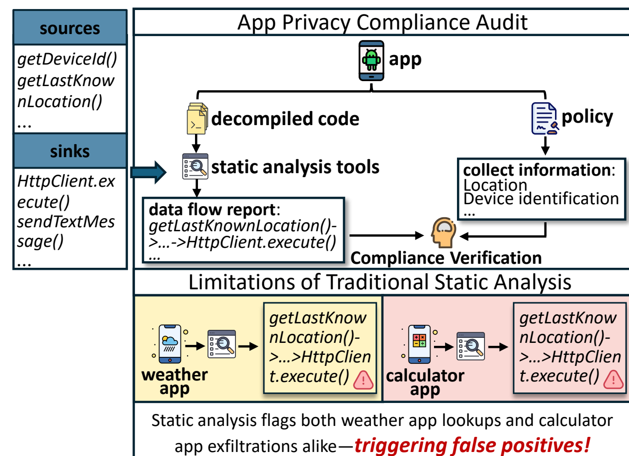

Figure 1: Workflow for App Privacy Compliance Auditing and the Semantic Blindness of Static Analysis. The figure illustrates how traditional static analysis struggles with semantically distinguishing data flows. Both a weather app and a calculator app perform a data flow from obtaining the last known location (`getLastKnownLocation()`) to sending data over the network (`HttpClient.execute()`). However, the static analyzer, lacking contextual understanding, flags both flows as potential violations, leading to false positives. The intended functionality of the weather app is not differentiated from the potentially unintended or undisclosed data transmission by the calculator app.

To address this, static analysis (Arzt et al. 2014; Zhang, Tian, and Duan 2019; Wei et al. 2018) has become a foundational technique. To ensure no true privacy violation is ever missed, these tools are designed for soundness, a goal they achieve through over-approximation. This conservative strategy identifies every possible path sensitive data could take from a source to a sink, regardless of its real-world feasibility. While effective at catching potential violations, this pursuit of soundness creates a critical flaw: semantic blindness. As illustrated in Figure 1, an analyzer cannot distinguish benign, intended functionality from a genuine privacy leak. This flaw inevitably buries true violations in a flood of false positives, rendering the entire auditing process unscalable.

The semantic reasoning that static analysis critically lacks is a core strength of Large Language Models (LLMs), presenting a natural path forward for bridging this gap. However, their naive application is infeasible. To be effective, LLM-powered auditing framework must overcome three fundamental challenges: **(1) High False Positives**: The framework must intelligently sift through the enormous volume of noisy alerts from static analysis. **(2) Cost and Scalability**: Directly analyzing entire codebases with powerful LLMs is prohibitively expensive and constrained by model context windows. **(3) High Explainability**: The final output must be a clear, human-readable justification, not the opaque alerts of traditional tools.

To systematically address these challenges, we introduce PriAgent, a novel framework that orchestrates a team of specialized AI agents to perform a collaborative reasoning task. First, to tackle noise and scalability, it performs a crucial triage. A FlowShaper agent abstracts enormous volume of raw alerts into high-level patterns, and a LocusFinder agent pinpoints the minimal set of "semantic loci"—the critical code locations within a long call chain that determine its true intent. This strategic pruning reduces the problem space by orders of magnitude. Second, to achieve deep semantic understanding, a SemanticVerifier agent conducts a focused analysis on these loci. It leverages Retrieval-Augmented Generation (RAG) with a curated knowledge base of Android APIs, empowering it to discern potentially non-compliant behavior from benign functionality. Finally, to deliver high explainability, a ComplianceArbiter agent synthesizes the code-level evidence with the app's privacy policy, rendering a holistic and interpretable verdict for each potential violation.

Our contributions are threefold:

- **A Novel Multi-Agent Auditing Framework:** To the best of our knowledge, PriAgent is the first framework to orchestrate a collaborative team of specialized AI agents that systematically apply semantic reasoning to the complex task of Android privacy compliance auditing.
- **Scalable and Precise Semantic Analysis:** We introduce a multi-stage pipeline that strategically minimizes the analytical scope, making LLM-based auditing practical at scale while dramatically reducing false positives.
- **Holistic and Explainable Compliance Auditing:** PriAgent provides a holistic assessment by correlating code-level behavior with privacy policy statements. It generates detailed, narrative reports that explain why a data flow is judged to be benign or malicious, dramatically improving usability for security auditors.

Related Work

Android Privacy Compliance Auditing

Research in automated privacy compliance auditing primarily follows two methodologies: static and dynamic analysis.

Static analysis. It is a dominant approach for verifying consistency between code and policy. This area was pioneered by frameworks such as PPChecker (Yu et al. 2018), HPDroid (Fan et al. 2020), and the system proposed by Zimmeck et al. (Zimmeck et al. 2017). These systems typically rely on underlying taint analysis techniques to map data flows. The foundational taint tracker, FlowDroid (Arzt et al. 2014), provides robust context and flow-sensitive analysis but has limitations in handling Inter-Component Communication. To overcome this, specialized tools like Epicc (Octeau et al. 2013) and IccTA (Li et al. 2015) were developed. The landscape also includes other significant contributions, such as the universal analysis framework Amandroid (Wei et al. 2018), DroidSafe (Gordon et al. 2015) for resolving dynamically constructed values, and FastDroid (Zhang, Tian, and Duan 2019), which utilizes pollution variable relationship graphs. More recently, to address the low efficiency of these tools, GNChecker (Fan et al. 2024) was introduced to improve static analysis performance through a segmentation strategy.

Dynamic analysis. It executes the application to gather concrete evidence of its behavior. Systems like PoliCheck (Andow et al. 2020) and PurPliance (Bui et al. 2021) employ this method, typically by monitoring network traffic for data transmissions that may violate policy statements. The effectiveness of this approach hinges on the code coverage achieved by automated testing tools. These tools vary in strategy, from the search-based testing of Sapienz (Alshahwan et al. 2018; Mao, Harman, and Jia 2016) to the guided random testing in WCTester (Zeng et al. 2016) and the model-based exploration used by Fastbot (Cai, Zhang, and Yang 2020; Lv et al. 2022).

Furthermore, the scope of compliance consistency analysis extends beyond mobile applications to other platforms, such as Mini Apps (Wang et al. 2024b), iOS (Liu et al. 2024), Smart Contracts (Yang et al. 2024). Ultimately, Static analysis, in its pursuit of soundness through over-approximation, inevitably suffers from the semantic blindness we described, drowning auditors in a sea of false positives. Conversely, dynamic analysis is constrained by incomplete code coverage, risking missed true violations.

Large Language Models for Program Analysis

Large Language Models (LLMs) are significantly reshaping software engineering, with diverse applications spanning code completion (Lozhkov et al. 2024), comprehension (Wang et al. 2023), synthesis (Zheng et al. 2024), and repair (Bouzenia, Devanbu, and Pradel 2024). In the specific domain, pioneering studies have employed LLMs to infer or rank program invariants (Wu et al. 2024) or to retrieve function specifications to augment traditional bug detectors (Li et al. 2024; Wang et al. 2024a). However, these efforts focus on localized code segments or isolated properties. They are not designed to interpret the end-to-end semantic intent of the long, inter-procedural data flows central to compliance auditing.

Method

PriAgent is a novel framework that automates Android privacy compliance auditing by linking an app's privacy policy to its code-level behavior. It processes raw data flow reports from static analysis into explainable verdicts.

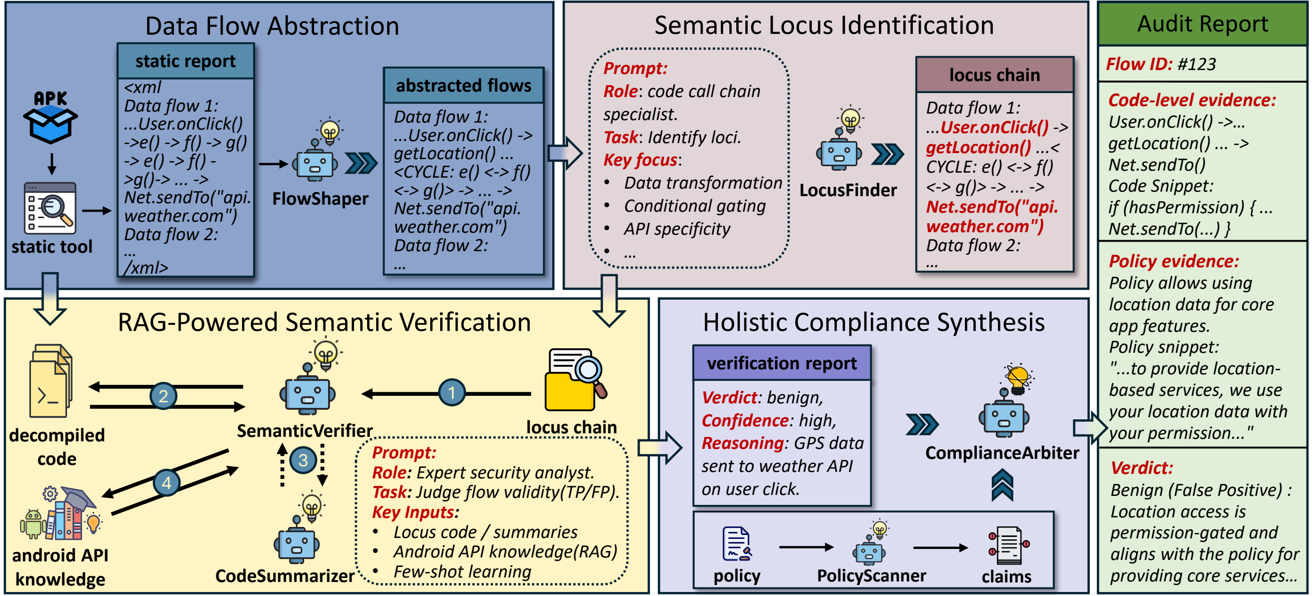

Figure 2: Overview of the PriAgent Framework.

The overall architecture, depicted in Figure 2, orchestrates this sequence of specialized agents across four principal stages: (1) Data Flow abstraction, (2) Semantic Locus Identification, (3) RAG-Powered Semantic Verification, and (4) Holistic Compliance Synthesis. Each agent in the framework is guided by a sophisticated, role-specific prompt.

Data Flow Abstraction

The initial set of data flows, D_{raw}, produced by static analyzers is often too voluminous for direct LLM-based analysis. This stage aims to abstract and reduce this set into a manageable and representative collection, D_{abs}. This crucial pre-processing, performed by the **FlowShaper** agent, is an innovative first step that makes subsequent, expensive semantic analysis computationally feasible by focusing on architectural data flow patterns rather than thousands of redundant instances.

We formally define a data flow $d \in D_{raw}$ as a tuple $d = (src, snk, \pi)$, where src and snk are the source and sink method signatures, and $\pi = \langle m_1, m_2, \ldots, m_n \rangle$ is the call chain. The FlowShaper applies two key abstraction techniques, both aimed at significantly reducing the number of flows requiring costly LLM analysis:

- **Intra-package flow aggregation:** We define an equivalence relation $\sim_{pkg}$ where two flows, d_i and d_j, are considered equivalent ($d_i \sim_{pkg} d_j$) if they satisfy the following conditions:

$$\begin{aligned} package(d_i.src) &= package(d_j.src) \quad \text{and} \\ package(d_i.snk) &= package(d_j.snk). \end{aligned} \tag{1}$$

  The FlowShaper then partitions D_{raw} by this relation. This allows the framework to assess a package's behavior as a single entity, drastically reducing the token consumption that would result from redundantly analyzing numerous similar internal flows.

- **Iterative path abstraction:** To handle flows within iterative constructs, the FlowShaper identifies cyclical or recursive call patterns. It abstracts these paths using special notations such as `<CYCLE: methodA() <-> methodB()>`. This powerful abstraction ensures that a single underlying vulnerability within a loop is investigated only once, preventing a potential token explosion from analyzing countless near-identical paths.

The output of the FlowShaper is a set of abstracted data flows, $D_{abs} = \{d'_1, d'_2, \ldots, d'_m\}$, where $m \ll |D_{raw}|$. Each abstracted flow $d' = (src, snk, \pi_{abs})$ contains an abstracted call chain π_{abs} that is structurally and semantically richer than the raw input, primed for the next stage of intelligent triage.

Semantic Locus Identification

The previous stage outputs a set of abstracted data flows, D_{abs}. However, each flow's call chain, π_{abs}, can still traverse dozens of methods, making a full analysis with an LLM computationally prohibitive. The critical challenge, therefore, is to pinpoint the specific methods within π_{abs} whose logic truly governs the data flow's validity. To solve this, the **LocusFinder** agent performs a strategic triage to identify a minimal set of functions, which we term the Semantic Loci. Formally, for each abstracted flow $d' \in D_{abs}$, the LocusFinder agent's goal is to produce a small subset of methods, denoted as $L(d')$. This subset is selected based on semantic relevance, but to ensure predictable performance and cost, we also enforce a hard cap, τ_{loci}. The final set of loci must satisfy the following conditions:

$$L(d') \subseteq \pi_{abs}, \quad |L(d')| \leq \tau_{loci} \tag{2}$$

We set $\tau_{loci} = 5$ as a balance between analytical depth and resource consumption. The LocusFinder agent is itself

an LLM call, guided by a sophisticated Chain-of-Thought (CoT) prompt that instructs it to act as a security expert and weigh each method in the call chain against these five critical dimensions: **(1) Data transformation potential**, assessing a method's capacity to alter or sanitize data and thus reduce its sensitivity; **(2) Conditional gating**, identifying "gatekeeper" methods that use logic like permission or consent checks to block a flow, a common source of false positives; **(3) Proximity to source/sink**, which assigns higher importance to methods at the data's entry and exit points; **(4) API & class specificity**, distinguishing generic utilities from purpose-built, suspicious classes; and **(5) Data colocation**, which scrutinizes the aggregation of sensitive data with other user identifiers, a strong heuristic for user profiling and privacy leaks.

RAG-Powered Semantic Verification

This stage is PriAgent's analytical core, where the **SemanticVerifier** agent ascertains the validity of a potential data flow by synthesizing diverse evidence through a structured reasoning process that emulates an expert security analyst.

Formally, the input for this stage is a set of tuples, $\{(d'_i, L_i)\}$, where $d'_i \in D_{abs}$ is an abstracted data flow and $L_i = L(d'_i)$ is its corresponding set of semantic loci identified in the previous stage. The goal of the SemanticVerifier is to produce a set of structured verdicts, $V = \{v_1, v_2, \dots, v_m\}$, where each verdict v_i provides a definitive and explainable judgment on the corresponding flow d'_i. We define each verdict v_i as a tuple:

$$v_i = (\text{flow_id}_i, \text{judgment}_i, \text{explanation}_i, \text{confidence}_i) \quad (3)$$

where $\text{judgment}_i \in \{\text{True Positive, False Positive}\}$.

Evidence curation and summarization. The verification process begins with evidence gathering for each locus in L_i. The SemanticVerifier utilizes a custom tool to retrieve the full source code for each method $m \in L_i$ from the decompiled application. A critical challenge here is the inherent verbosity of code, which can exceed an LLM's context window and incur prohibitive computational costs.

To mitigate this, PriAgent employs an on-demand **codeSummarizer** agent. For any retrieved code block $C(m)$ whose size exceeds a predefined threshold τ_{size}, set to 3000 tokens to balance detail with context limits, the summarizer generates a high-fidelity abstraction, $S(m)$. This summary is not a naive truncation; it is a structured representation that preserves the core data-handling logic, I/O operations, and control flow predicates. Thus, the curated evidence for each locus, E_m, is defined as:

$$E_m = \begin{cases} S(m) & \text{if size}(C(m)) > \tau_{size} \\ C(m) & \text{otherwise} \end{cases} \quad (4)$$

This ensures that the subsequent analysis is both scalable and information-rich.

RAG-powered prompt engineering. The cornerstone of our verification is the construction of a comprehensive reasoning context for the LLM. This is achieved through a sophisticated prompt engineering strategy that integrates multiple facets of evidence. For each flow d'_i, the prompt $\mathcal{P}_i$ is composed of:

- **Flow specification:** The full abstracted data flow path π_{abs} from d'_i.
- **Curated code evidence:** The set of all relevant code snippets or summaries, $\{E_m | m \in L_i\}$.
- **Augmented API knowledge (RAG):** The prompt is dynamically enriched with authoritative context from our knowledge base, $\mathcal{KB}_{API}$. We constructed this knowledge base by systematically parsing the official android developer documentation (Android Developers 2025) to extract key information for each sensitive API, including its official purpose, required permissions, and typical usage patterns. This data was then structured into a vector database optimized for semantic retrieval. When a source or sink method is encountered, the SemanticVerifier queries $\mathcal{KB}_{API}$ to retrieve this precise context. This RAG approach is a key advantage over fine-tuning, as the knowledge base can be continuously updated independently of the LLM, ensuring long-term adaptability.
- **In-Context exemplars (Few-shot learning):** To ground the LLM's reasoning, the prompt includes three handcrafted exemplars that teach it to recognize abstract patterns of common false positives which require holistic semantic reasoning: **(1) Evaluating the execution context**: This involves identifying infeasible paths and recognizing flows that are permissible because they are explicitly user-initiated. **(2) Assessing data state and purpose**: This teaches the model to recognize when data is no longer sensitive after transformation and to distinguish flows essential for an app's core functionality from those for secondary purposes. **(3) Interpreting code semantics**: This teaches the model to understand the true behavior of code. It covers the misinterpretation of framework APIs, the resolution of simple reflective calls, and the assessment of risks associated with obsolete APIs in modern Android versions.

Chain-of-Thought Verdict Generation. Using the constructed prompt $\mathcal{P}_i$, the SemanticVerifier instructs the LLM to perform Chain-of-Thought (CoT) reasoning. The model analyzes the code, correlates it with the RAG-augmented API knowledge and in-context exemplars, and synthesizes these findings to produce a judgment and a detailed rationale. This step-by-step process makes the LLM's reasoning transparent and reliable.

Holistic Compliance Synthesis

In its final stage, PriAgent synthesizes verified code-level findings into actionable compliance intelligence, translating technical evidence into clear verdicts on an application's adherence to its stated privacy policy.

This stage is orchestrated by the **ComplianceArbiter** agent, which takes the structured verdicts V from the *SemanticVerifier* and the app's privacy policy, $\mathcal{P}$, as input. To understand the policy's content, a **PolicyScanner** agent, also powered by a specialized LLM prompt, parses the unstructured privacy policy text. It is tasked with identifying and extracting explicit claims regarding data collection, usage, and sharing for specific data types, thereby converting le-

gal prose into a structured representation $C_{\mathcal{P}}$ for subsequent comparison.

The **ComplianceArbiter** then cross-examines each `True Positive` verdict by comparing the LLM's explanation with the extracted policy claims ($C_{\mathcal{P}}$). This synthesis produces the final audit report, $\mathcal{R}_i$, a structured object designed for an analyst and defined as:

$$\mathcal{R}_i = (\text{flow_id}_i, \text{risk_category}_i, \text{analyst_briefing}_i) \quad (5)$$

The risk_category_i is assigned from a multi-tiered standard: **(1) High-risk violation** for flows that are undisclosed or contradicted by the policy; **(2) Undeclared behavior** for flows whose purpose is ambiguous or not clearly declared; and **(3) Declared behavior** for flows accurately described in the policy.

Crucially, the analyst_briefing is a concise LLM-generated narrative that justifies the assigned risk category by synthesizing code evidence with the relevant policy clauses (or their absence). Ultimately, PriAgent transforms a large volume of noisy alerts into a focused queue of high-risk, explainable issues, dramatically enhancing auditor efficiency and accuracy.

Experiments

This section presents a series of experiments designed to rigorously evaluate the PriAgent framework. Our evaluation is guided by four research questions (RQs) that systematically assess PriAgent's effectiveness, performance, and usability.

- **RQ1:** How effective is PriAgent at mitigating static analysis false positives while preserving true positive detection?
- **RQ2:** How effective is PriAgent at semantically analyzing data flows to reduce false positives in real-world applications?
- **RQ3:** What is the impact of PriAgent's architectural components on its effectiveness, and what is the performance-cost trade-off across different LLM backends?
- **RQ4:** How effectively does PriAgent perform end-to-end privacy compliance auditing by correlating code analysis with policy claims?

Experimental Setup

Baselines. To ensure a comprehensive evaluation, we benchmark PriAgent against three distinct static analysis tools. **FlowDroid** (Arzt et al. 2014), a widely recognized academic framework known for its high-accuracy, context-sensitive analysis; **Amandroid** (Wei et al. 2018), noted for its efficiency (we extended it with a lightweight path-reconstruction module to recover full data-flow chains for fair comparison); and **AppShark** (ByteDance 2022), an open-source industrial tool from ByteDance.

Large Language Models (LLMs). Our experiments utilize a curated set of state-of-the-art LLMs, including top-tier proprietary models (**Gemini 2.5 Pro**, **Claude 3.5 Sonnet**, and **GPT-4o**) to measure maximum capability, and leading open-source models (**LLaMA 3-70B**, **Qwen-2.5-Coder 32B**, and **DeepSeek-R1**) to evaluate the effectiveness of powerful alternatives.

Datasets. Our evaluation employs four distinct datasets to rigorously test PriAgent's capabilities from different angles. **TaintBench** (Luo et al. 2022): A standard benchmark consisting of 39 real-world malicious apps with expert-verified taint flows. **UBCBench** (Zhang et al. 2021): A complementary benchmark of 24 applications that covers a broader range of Android features and usage scenarios. **Real-World Benign Apps (RWBA)**: To assess PriAgent's performance in a high-noise, real-world setting, we constructed this dataset of 20 popular, non-malicious applications. The apps were curated from the Google Play Store across diverse categories to ensure functional complexity and code quality. **Large-Scale Audit Set (LSAS)**: We built this dataset of 300 applications. These were selected by randomly sampling applications from the Google Play Store that provided a public-facing privacy policy link in their store listing. This random sampling methodology ensures a representative cross-section of typical app behaviors and policy disclosure practices in the wild.

Specification and Environment. We used tailored source and sink lists for each dataset. For TaintBench, the list was generated directly from the benchmark's ground-truth labels to ensure maximum relevance. For the other three datasets (UBCBench, RWBA, and LSAS), we used a common configuration to simulate real-world analysis: FlowDroid's default source/sink list, which was pruned by removing known inappropriate definitions as documented in prior work (Luo, Bodden, and Späth 2019). All experiments were run on a server equipped with an Intel Xeon Gold 6330 CPU and 256GB of RAM.

RQ1: False Positive Mitigation

Objective: RQ1 evaluates PriAgent's core capability: how effectively it can reduce false positives from static analysis tools while preserving their ability to detect true violations.

Results: Prior work has demonstrated that baseline static analysis tools perform poorly on benchmarks like TaintBench and UBCBench, often suffering from severe false positive and false negative rates. PriAgent is designed to specifically mitigate the false positive problem, thereby improving the usability and trustworthiness of static analysis results. As shown in Table 1, PriAgent delivers a dramatic improvement. On the challenging TaintBench dataset, it boosts FlowDroid's precision from 73.7% to 93.2%. This is achieved by semantically identifying four major categories of false positives as shown in Figure 4(a): Data Transformation, recognizing that internal data handling does not constitute a leak; API Semantic Misinterpretation, which corrects flawed assumptions about API behavior or assesses risks in the context of modern platform versions(e.g., `getDeviceId()` to retrieve the IMEI is no longer functional for regular apps on Android 10+); Condition-Dependent Infeasible Paths, which identifies unreachable flows; and Simple Reflection Misresolution, which resolves benign, internal method invocations. This capability was further confirmed on the UBCBench dataset, where PriAgent again increased the F1-Score from 0.809 to 0.829 by eliminating the remaining false positives as shown in Figure 3.

Baseline Tool	Analysis Layer	LLM Backend	TP	FP	FN	Precision	Recall	F1-Score
FlowDroid	Tool Only	–	70	25	116	0.737	0.376	0.498
	w/PriAgent	Gemini 2.5 Pro	68	5	118	**0.932**	0.366	**0.525**
		Claude 3.5 Sonnet	68	6	118	0.919	0.366	0.523
		GPT-4o	67	7	119	0.905	0.360	0.515
		LLaMA 3-70B	65	10	121	0.867	0.349	0.498
		Qwen-2.5-Coder 32B	64	12	122	0.842	0.344	0.489
		DeepSeek-R1	64	13	122	0.831	0.344	0.487
Amandroid	Tool Only	–	20	15	166	0.571	0.108	0.182
	w/PriAgent	Gemini 2.5 Pro	20	3	166	**0.870**	0.108	**0.192**
		Claude 3.5 Sonnet	20	4	166	0.833	0.108	0.191
		GPT-4o	19	4	167	0.826	0.102	0.182
		LLaMA 3-70B	18	6	168	0.750	0.097	0.172
		Qwen-2.5-Coder 32B	17	7	169	0.708	0.091	0.162
		DeepSeek-R1	17	8	169	0.680	0.091	0.161
AppShark	Tool Only	–	85	18	101	0.825	0.457	0.588
	w/PriAgent	Gemini 2.5 Pro	83	4	103	**0.954**	0.446	**0.608**
		Claude 3.5 Sonnet	83	5	103	0.943	0.446	0.606
		GPT-4o	82	6	104	0.932	0.441	0.599
		LLaMA 3-70B	80	8	106	0.909	0.430	0.585
		Qwen-2.5-Coder 32B	79	10	107	0.888	0.425	0.576
		DeepSeek-R1	79	11	107	0.878	0.425	0.573

Table 1: Comprehensive performance comparison on TaintBench.

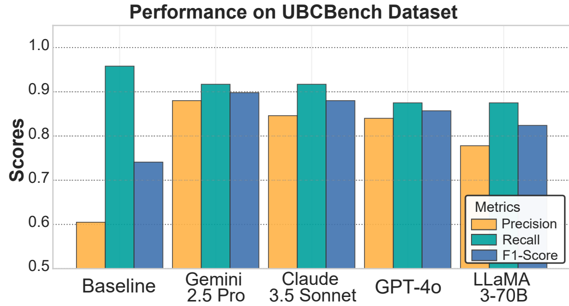

Figure 3: Performance on the UBCBench dataset using FlowDroid as the baseline.

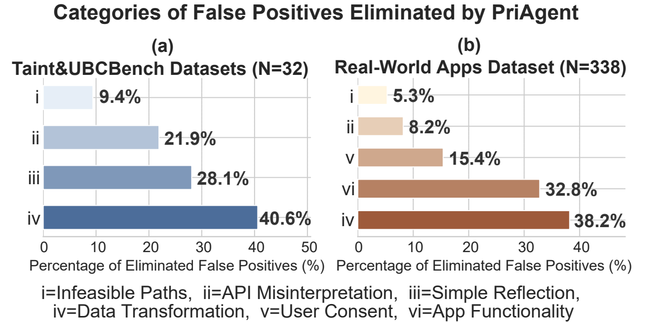

Figure 4: Distribution of false positive types eliminated by PriAgent on (a) taintBench&UBCBench and (b) RWBA dataset.

RQ2: Real-World Semantic Interpretation

Objective: RQ2 assesses PriAgent's ability to interpret complex data flows in benign, real-world applications.

Results: On the RWBA dataset, PriAgent reduced 583 raw alerts from FlowDroid to just 245 actionable reports, a 58% reduction. A breakdown of the 338 eliminated false positives is presented in Figure 4(b). The results show that PriAgent's effectiveness stems from its ability to understand code context. In addition to the categories identified in RQ1, two new types of false positives are prevalent in this real-world setting: Core Application Functionality, where data flows are essential to the app's primary purpose, and flows triggered by User Consent, such as data collection that occurs only after a user explicitly clicks an"Agree" button. The 245 remaining alerts underscore the fundamental limits of static analysis. They are predominantly caused by dynamic code features, such as dynamic code loading and complex Inter-component communication, which are difficult to resolve without runtime information and remain a key challenge for future work.

RQ3: Performance Analysis and Architectural Validation

Objective: RQ3 analyzes PriAgent's performance-cost trade-off to demonstrate its practical acceptability for real-world auditing tasks. We also present an ablation study to validate the necessity of its core architectural components.

Results: PriAgent is designed to be both time- and cost-efficient. To provide concrete metrics, we analyzed its performance on our two main datasets using the Gemini 2.5 Pro backend. In our setup, the static analyzer generated an average of 2˜3 data flows per app on TaintBench and a much higher 29 flows per app on the noisier RWBA dataset. Our framework processes each flow efficiently, with an average time of 25 seconds and a cost of $0.16 per flow.

This translates to a highly acceptable average wall-clock time of 75 seconds per app for TaintBench and 2.7 minutes for the more complex RWBA apps. The total time is significantly less than the sum of individual flow processing times due to parallelization. The cost-effectiveness trade-off across different LLM backends is illustrated in Figure 5, confirming that options exist for various budget and accuracy requirements.

Ablation Study: To validate our architectural design, we conducted an ablation study on a randomly selected subset of 15 apps from TaintBench and RWBA. This smaller subset was necessary because, as the results show, removing efficiency-focused components dramatically increases token consumption, making a full-scale run prohibitively expensive. The results in Table 2 confirm that each component is indispensable. The removal of any component leads to a significant degradation in either efficiency or accuracy. FlowShaper, LocusFinder and codeSummarizer are the cornerstones of efficiency; respectively, with the removal of LocusFinder also causing a systemic collapse in accuracy. Lacking the API knowledge from RAG, the F1-score drops to 0.501, proving that external knowledge is essential.

Configuration	F1-Score	Time (s)	Cost ($)
PriAgent	**0.524**	**120**	**$3.04**
w/o FlowShaper	0.522	600	$15.20
w/o LocusFinder	0.460	1800	$45.60
w/o codeSummarizer	0.512	300	$7.60
w/o RAG	0.501	115	$2.90

Table 2: Ablation study of PriAgent's core components on a mixed 15-app subset.

RQ4: End-to-End Compliance Auditing

Objective: RQ4 evaluates PriAgent's capability to correlate code analysis with privacy policies and generate high-precision compliance verdicts on a large, real-world dataset.

Results: We deployed PriAgent on LSAS dataset of 300 apps. The initial static analysis yielded 9,384 raw alerts, which wbbmere distilled by PriAgent into 2,304 verified data flows. The ComplianceArbiter then flagged 127 as potential violations. To validate these findings, two authors with expertise in mobile security performed a manual audit on all 127 flagged reports. Using a cross-validation process to ensure accuracy, they confirmed 119 genuine violations across 54 of the 300 apps, achieving a final verdict precision of 93.7%.

The majority of these violations, 79 instances, were categorized as Undeclared Behavior, where the data transmission was not mentioned in the policy. The remaining 40

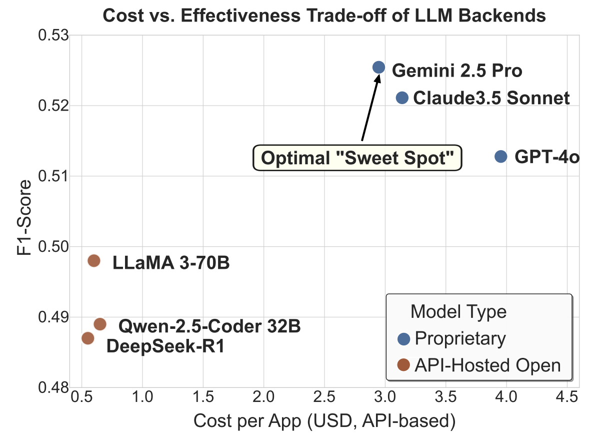

Figure 5: Cost vs. Effectiveness plot for different LLM backends. Gemini 2.5 Pro is shown to be in the "sweet spot".

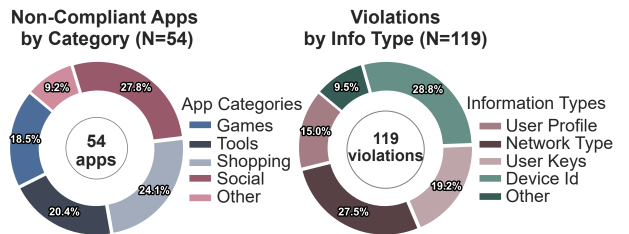

Figure 6: Quantitative analysis of the 119 confirmed violations found in 54 non-compliant apps.

instances were High-Risk Violations that contradicted the policy's claims. To further analyze these findings, Figure 6 illustrates the distribution of the 54 non-compliant apps across store categories and the types of information involved in the 119 confirmed violations. As the figure shows, non-compliant apps are most concentrated in Social & Comm. and Shopping & Lifestyle categories. Our analysis reveals that the most frequently leaked data types are sensitive Device Identifiers and NetWork Type. For each violation, PriAgent generates a detailed audit report synthesizing code and policy evidence as shown in the appendix of supplementary material, fulfilling the promise of a truly usable compliance auditing tool.

Conclusion

This paper tackles the semantic blindness of static analysis, a primary cause of false positives in Android privacy auditing. We introduce PriAgent, a multi-agent framework that employs AI-driven semantic reasoning to make large-scale auditing practical. By systematically reducing the analysis scope and correlating code with privacy policies, PriAgent achieves a 58% reduction in false alerts on real apps and 93.7% precision in end-to-end audits. By shifting from syntactic tracking to semantic reasoning, PriAgent offers a practical path to scalable, trustworthy automated compliance.

Acknowledgments

This work is supported by the Scaling Program of Institute of Information Engineering,CAS (Grant No. E5V00911D2).

References

Alshahwan, N.; Gao, X.; Harman, M.; Jia, Y.; Mao, K.; Mols, A.; Tei, T.; and Zorin, I. 2018. Deploying search based software engineering with sapienz at facebook. In *International Symposium on Search Based Software Engineering*, 3–45. Springer.

Andow, B.; Mahmud, S. Y.; Whitaker, J.; Enck, W.; Reaves, B.; Singh, K.; and Egelman, S. 2020. Actions speak louder than words:{Entity-Sensitive} privacy policy and data flow analysis with {PoliCheck}. In *29th USENIX Security Symposium (USENIX Security 20)*, 985–1002.

Android Developers. 2025. Android API Reference Documentation. https://developer.android.com/reference. Accessed on 2025-07-25.

Arzt, S.; Rasthofer, S.; Fritz, C.; Bodden, E.; Bartel, A.; Klein, J.; Le Traon, Y.; Octeau, D.; and McDaniel, P. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices*, 49(6): 259–269.

Bouzenia, I.; Devanbu, P.; and Pradel, M. 2024. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*.

Bui, D.; Yao, Y.; Shin, K. G.; Choi, J.-M.; and Shin, J. 2021. Consistency analysis of data-usage purposes in mobile apps. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2824–2843.

ByteDance. 2022. AppShark. https://github.com/bytedance/appshark. Accessed on 2025-07-25.

Cai, T.; Zhang, Z.; and Yang, P. 2020. Fastbot: A multi-agent model-based test generation system Beijing Bytedance Network Technology Co., Ltd. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, 93–96.

European Union. 2018. The EU General Data Protection Regulation (GDPR). https://gdpr-info.eu/. Accessed on 2025-07-25.

Fan, M.; Shi, J.; Wang, Y.; Yu, L.; Zhang, X.; Wang, H.; Jin, W.; and Liu, T. 2024. Giving without Notifying: Assessing Compliance of Data Transmission in Android Apps. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 1595–1606.

Fan, M.; Yu, L.; Chen, S.; Zhou, H.; Luo, X.; Li, S.; Liu, Y.; Liu, J.; and Liu, T. 2020. An empirical evaluation of GDPR compliance violations in Android mHealth apps. In *2020 IEEE 31st international symposium on software reliability engineering (ISSRE)*, 253–264. IEEE.

Gamba, J.; Feal, Á.; Blazquez, E.; Bandara, V.; Razaghpanah, A.; Tapiador, J.; and Vallina-Rodriguez, N. 2023. Mules and permission laundering in android: Dissecting custom permissions in the wild. *IEEE Transactions on Dependable and Secure Computing*, 21(4): 1801–1816.

Gordon, M. I.; Kim, D.; Perkins, J. H.; Gilham, L.; Nguyen, N.; and Rinard, M. C. 2015. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, 110.

Li, H.; Hao, Y.; Zhai, Y.; and Qian, Z. 2024. Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1): 474–499.

Li, L.; Bartel, A.; Bissyandé, T. F.; Klein, J.; Le Traon, Y.; Arzt, S.; Rasthofer, S.; Bodden, E.; Octeau, D.; and McDaniel, P. 2015. Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, 280–291. IEEE.

Liu, D.; Xiao, Y.; Zhang, C.; Xie, K.; Bai, X.; Zhang, S.; and Xing, L. 2024. {iHunter}: Hunting Privacy Violations at Scale in the Software Supply Chain on {iOS}. In *33rd USENIX Security Symposium (USENIX Security 24)*, 5663–5680.

Lozhkov, A.; Li, R.; Allal, L. B.; Cassano, F.; Lamy-Poirier, J.; Tazi, N.; Tang, A.; Pykhtar, D.; Liu, J.; Wei, Y.; et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Luo, L.; Bodden, E.; and Späth, J. 2019. A qualitative analysis of android taint-analysis results. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 102–114. IEEE.

Luo, L.; Pauck, F.; Piskachev, G.; Benz, M.; Pashchenko, I.; Mory, M.; Bodden, E.; Hermann, B.; and Massacci, F. 2022. TaintBench: Automatic real-world malware benchmarking of Android taint analyses. *Empirical Software Engineering*, 27(1): 16.

Lv, Z.; Peng, C.; Zhang, Z.; Su, T.; Liu, K.; and Yang, P. 2022. 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*, 1–5.

Mao, K.; Harman, M.; and Jia, Y. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th international symposium on software testing and analysis*, 94–105.

Octeau, D.; McDaniel, P.; Jha, S.; Bartel, A.; Bodden, E.; Klein, J.; and Le Traon, Y. 2013. Effective {Inter-Component} communication mapping in android: An essential step towards holistic security analysis. In *22nd USENIX Security Symposium (USENIX Security 13)*, 543–558.

State of California Department of Justice. 2020. California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa. Accessed on 2025-07-25.

Wang, C.; Lou, Y.; Liu, J.; and Peng, X. 2023. Generating variable explanations via zero-shot prompt learning. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 748–760. IEEE.

Wang, C.; Zhang, W.; Su, Z.; Xu, X.; Xie, X.; and Zhang, X. 2024a. LLMDFA: analyzing dataflow in code with large language models. *Advances in Neural Information Processing Systems*, 37: 131545–131574.

Wang, Y.; Fan, M.; Zhou, H.; Wang, H.; Jin, W.; Li, J.; Chen, W.; Li, S.; Zhang, Y.; Han, D.; et al. 2024b. MiniChecker: Detecting Data Privacy Risk of Abusive Permission Request Behavior in Mini-Programs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 1667–1679.

Wei, F.; Roy, S.; Ou, X.; and Robby. 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)*, 21(3): 1–32.

Wu, G.; Cao, W.; Yao, Y.; Wei, H.; Chen, T.; and Ma, X. 2024. Llm meets bounded model checking: Neuro-symbolic loop invariant inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 406–417.

Yang, S.; Lin, X.; Chen, J.; Zhong, Q.; Xiao, L.; Huang, R.; Wang, Y.; and Zheng, Z. 2024. Hyperion: Unveiling dapp inconsistencies using llm and dataflow-guided symbolic execution. *arXiv preprint arXiv:2408.06037*.

Yu, L.; Luo, X.; Chen, J.; Zhou, H.; Zhang, T.; Chang, H.; and Leung, H. K. 2018. Ppchecker: Towards accessing the trustworthiness of android apps' privacy policies. *IEEE Transactions on Software Engineering*, 47(2): 221–242.

Zeng, X.; Li, D.; Zheng, W.; Xia, F.; Deng, Y.; Lam, W.; Yang, W.; and Xie, T. 2016. Automated test input generation for android: Are we really there yet in an industrial case? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 987–992.

Zhang, J.; Tian, C.; and Duan, Z. 2019. Fastdroid: efficient taint analysis for android applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 236–237. IEEE.

Zhang, J.; Wang, Y.; Qiu, L.; and Rubin, J. 2021. Analyzing android taint analysis tools: FlowDroid, Amandroid, and DroidSafe. *IEEE Transactions on Software Engineering*, 48(10): 4014–4040.

Zheng, T.; Zhang, G.; Shen, T.; Liu, X.; Lin, B. Y.; Fu, J.; Chen, W.; and Yue, X. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*.

Zimmeck, S.; Wang, Z.; Zou, L.; Iyengar, R.; Liu, B.; Schaub, F.; Wilson, S.; Sadeh, N. M.; Bellovin, S. M.; and Reidenberg, J. R. 2017. Automated Analysis of Privacy Requirements for Mobile Apps. In *NDSS*, volume 2, 1–4.