TaskFlow: LLMs for Android Taint Specification

MARCO ALECCI, SnT, University of Luxembourg, Luxembourg
JORDAN SAMHI, SnT, University of Luxembourg, Luxembourg
MARC MILTENBERGER, Fraunhofer Institute for Secure Information Technology, Germany
STEVEN ARZT, Fraunhofer Institute for Secure Information Technology, Germany
TEGAWENDÉ F. BISSYANDÉ, SnT, University of Luxembourg, Luxembourg
JACQUES KLEIN, SnT, University of Luxembourg, Luxembourg

Taint analysis is a widely used technique to analyze Android apps, enabling the tracking of data flow within an app. A key step consists in identifying which methods act as `SOURCE`s (i.e., where a data of interest originates) and `SINK`s (where data might be exposed). Existing approaches typically fall into two categories: ① Handcrafted lists, which suffer from incompleteness and quickly become outdated; and ② Automated techniques, which, although scalable, over-approximate and produce many false positives, primarily due to the challenge of defining what qualifies as a `SOURCE`. While identifying `SINK`s is generally more straightforward (as they correspond to explicit exposure points), defining a universal criterion for what constitutes a `SOURCE` remains inherently challenging. For example, `isMicrophoneMute()` may not typically be considered a `SOURCE`, yet in specific contexts it could represent a significant privacy concern. This context dependence highlights the limitations of static, generic lists of `SOURCE` methods.

We present TaskFlow, a novel LLM-driven framework for generating *task-specific* lists of `SOURCE` (or `SINK`) methods aligned with specific analysis goals. By reasoning over API semantics and contextual usage, TaskFlow ① mitigates the noise commonly introduced by overly broad lists, leading to more precise taint analyses; and ② addresses the incompleteness of manual approaches.

CCS Concepts: • **Security and privacy → Software security engineering**; • **Software and its engineering**;

Additional Key Words and Phrases: Android Security, Static Analysis, Data Leaks

ACM Reference Format:
Marco Alecci, Jordan Samhi, Marc Miltenberger, Steven Arzt, Tegawendé F. Bissyandé, and Jacques Klein. 2026. TaskFlow: LLMs for Android Taint Specification. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (January 2026), 28 pages. https://doi.org/10.1145/3815184

1 Introduction

Taint analysis is a widely adopted technique in analyzing Android applications (apps) for detecting potential issues related to the flow of sensitive information (e.g., sensitive data leaks). It is commonly used to track how data propagates through an app's code, helping to identify potential privacy violations or unintended data exposures. A broad spectrum of research and tools leverages taint analysis for various purposes, including sensitive data leak detection [6, 10, 14, 18, 23, 41, 42], Android component leak detection [21], dynamic policy enforcement [39], and malware analysis [45, 53]. It has also

Authors' Contact Information: Marco Alecci, SnT, University of Luxembourg, Luxembourg, Luxembourg, marco.alecci@uni.lu; Jordan Samhi, SnT, University of Luxembourg, Luxembourg, Luxembourg, jordan.samhi@uni.lu; Marc Miltenberger, Fraunhofer Institute for Secure Information Technology, Darmstadt, Hessen, Germany, marc.miltenberger@sit.fraunhofer.de; Steven Arzt, Fraunhofer Institute for Secure Information Technology, Darmstadt, Hessen, Germany, steven.arzt@sit.fraunhofer.de; Tegawendé F. Bissyandé, SnT, University of Luxembourg, Luxembourg, Luxembourg, tegawende.bissyande@uni.lu; Jacques Klein, SnT, University of Luxembourg, Luxembourg, Luxembourg, jacques.klein@uni.lu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
Manuscript submitted to ACM

been applied to detect hidden behaviors [37, 51], analyze inter-app communication [8, 22], identify component hijacking vulnerabilities [27, 50], uncover runtime-sensitive values [40], and assess compliance with GDPR regulations [11, 12].

A core component of this technique is the identification of SOURCEs—methods where sensitive data enters the system—and SINKs—methods where such data could be exposed or misused. Any other methods are typically classified as NEITHER, as they are NEITHER SOURCEs nor SINKs. This process involves tagging (tainting) variables receiving values from SOURCE methods and monitoring how the tainted data propagates through the program, ultimately checking whether it reaches a SINK. Typically, in the Android ecosystem, a SOURCE is an API method that accesses privacy-sensitive data (e.g., location, contacts, clipboard), whereas a SINK is an API method that could potentially leak that data, such as writing to logs or transmitting it over the network.

While State-of-the-art tools like FlowDroid [6] can be used to perform this analysis, users need to provide the SOURCEs and SINKs as an input. As such, users are typically faced with two options: either ❶ manually craft a list tailored to their specific analysis task [2, 7, 44, 48] or ❷ use one of the many predefined SOURCE and SINK lists available in the literature [5, 10, 14, 31, 43, 46]. The vast number of Android API methods available to developers makes the manual creation of lists impractical [43]. Moreover, these lists can rapidly become outdated, making manual classification for every new Android release very challenging. For this reason, several automated techniques for identifying SOURCEs and SINKs have been proposed in the literature [5, 10, 14, 31, 43, 46]. One of the most popular approaches is SUSI, a feature-based machine learning method for identifying SOURCEs and SINKs in the Android framework [5]. A more recent approach is DocFlow [46], which extracts taint specifications from software documentation and has been shown to outperform SUSI in terms of accuracy.

However, while predefined lists offer a convenient starting point, they suffer from several important limitations. Studies have demonstrated that SUSI often misclassifies NEITHER Android API methods as SOURCEs, resulting in a significant number of irrelevant data flows [17, 28, 43, 47]. Our analysis suggests that similar challenges may persist in more recent approaches. For instance, our analysis of the list generated by DocFlow[46] indicates that some methods labeled as SOURCE might be better classified as NEITHER, as they do not appear to expose sensitive or meaningful information (see Section5.3 for further discussion). Second, such lists may be too broad or overly inclusive for specific tasks. For instance, approaches automatically generating lists may label as NEITHER a method that should actually be considered a SOURCE for a specific use case. A compelling example of this challenge is found in a recent study on clipboard usage in Android apps [7]. In that work, the authors developed ClipboardScope, a static analysis tool that uses a task-specific list of SOURCEs focused solely on clipboard-related APIs, such as `getPrimaryClip()` (which, for example, is absent from the DocFlow list of SOURCEs [46]) and `getText()`. This custom list allowed the tool to uncover nuanced clipboard misuse patterns that a generic SOURCE list would likely overlook. However, constructing such tailored lists is time-consuming, error-prone, and requires significant expert domain knowledge [43].

The main reason behind these two issues is that defining a universal criterion for what constitutes a SOURCE (i.e., a method returning a sensitive data) remains challenging [19]. This stems from the inherently vague and context-dependent nature of sensitivity, which is difficult to translate from high-level regulatory definitions (e.g., the EU's GDPR) into low-level technical constructs (i.e., methods in this case). We postulate that the definition of a SOURCE method highly depends on the task,i.e., the purpose of the taint analysis,and the analyst's ultimate objective. For instance, although `getLastKnownLocation()` is a commonly accepted SOURCE, less obvious methods such as `isMicrophoneMute()` might only be relevant in particular threat models, e.g., targeted attacks, or, as in the case of ClipboardScope [7], for task-specific privacy analysis.

In this work, we propose TaskFlow, a novel approach that leverages Large Language Models (LLMs) to automatically generate task-specific SOURCE lists for Android taint analysis. TaskFlow operates on Android API method signatures and their official documentation, which are used as semantic input to an LLM-driven analysis. In contrast to static heuristics or ML models trained on code features, TaskFlow utilizes the reasoning capabilities of LLMs to infer semantic usage patterns of APIs and to determine their relevance to a specific analysis goal. For example, while a generic taint analysis engine might overlook `isMicrophoneMute()`, TaskFlow can correctly identify it as a sensitive SOURCE when analyzing potential surveillance vectors. By aligning SOURCE selection with the intended analysis task, our method reduces irrelevant taint flows and yields more actionable results. Although our primary focus is on SOURCE identification, the same task-driven strategy naturally extends to SINKs as well, enabling users to derive task-specific SINK lists in scenarios where targeted outbound channels might be of particular interest.

Contributions. To address these challenges, this paper makes the following contributions:

- We discuss the challenges of identifying SOURCE methods in Android taint analysis, highlighting their inherently task-dependent nature.
- We introduce TaskFlow, an LLM-driven framework that generates task-specific SOURCE lists from natural language task descriptions. The approach relies on a scalable two-phase pipeline and can be naturally extended to generate SINK lists.
- We evaluate TaskFlow across diverse Android analysis tasks, demonstrating its effectiveness in generating precise and focused taint specifications compared to existing generic approaches.
- We publicly release TaskFlow and all associated resources to facilitate further research and adoption:

https://github.com/Trustworthy-Software/TaskFlow

2 Background

In this section, we briefly explain Taint Analysis, and the notion of SOURCE and SINK methods.

Taint Analysis. Taint analysis is an instantiation of dataflow analysis that tracks how data propagates through a program's control flow graph. When a variable is assigned the return value of a SOURCE method, it becomes tainted. This taint propagates to other variables through assignments or operations involving the tainted data. A report is generated if a tainted variable is later passed to a SINK method. In Android, for example, `getImei()` is a typical SOURCE, while `sendTextMessage()` represents a common SINK.

SOURCE method. In general, a SOURCE method refers to any method that returns a value, regardless of its sensitivity. Indeed, taint analysis, in general, can be applied to track any data. However, in the majority of cases, the literature amalgamates SOURCE methods with *sensitive* SOURCE methods. Since this amalgam is prevalent, in line with this convention, in this paper, we use the term SOURCE method to refer to a *sensitive* SOURCE method, i.e., any method that returns a piece of *sensitive* data.

SINK method. In contrast to SOURCE methods, which produce data, SINK methods represent endpoints where data may be exposed. These include, but are not limited to, operations such as logging, file I/O, network transmission, or inter-component communication. SINK methods are generally easier to identify, both for humans and automated systems, because they involve explicit data output operations. This makes them less prone to ambiguity in classification, as also reflected in our preliminary study (see Section 3) where LLMs achieved high accuracy in labeling SINK methods. As a result, our work mainly focuses on the more challenging task of identifying context-sensitive SOURCE methods.

3 Preliminary Study

Before developing our approach, we first conducted a series of preliminary experiments that provided us with valuable insights into the capabilities of LLMs in the context of taint specification. These experiments are described in this section. Given the increasing popularity of LLMs for software engineering tasks, we explored the idea of using them for labeling Android methods as SOURCE or SINK. In other words, our goal was to use LLMs to determine whether a given method should be classified as a SOURCE or a SINK when running tools such as FlowDroid to analyze Android apps.

We based our preliminary study on an existing manually labeled ground truth dataset of Android SOURCE and SINK methods [43]. The dataset contains 231 SOURCE methods, 130 SINK methods, and 654 NEITHER methods—that is, methods that are NEITHER SOURCEs nor SINKs. This yields a total of 1015 API methods used in our experiments. For our evaluation, we constructed two simple prompt templates, a zero-shot and a one-shot variant, focused exclusively on the 231 SOURCE and 130 SINK methods, temporarily excluding the NEITHER category. The one-shot prompt (the zero-shot version is identical except for the omission of examples) is as follows:

> **Prompt:** You are an expert in Taint Analysis for Android Applications.
> Please classify the following Android method into one of these two labels: SOURCE, SINK.
> Examples:
> Method : "android.telephony.SmsManager.sendTextMessage()"
> Category: SINK
> Method : "android.location.LocationManager.getLastKnownLocation()"
> Category: SOURCE
> Answer only with the Label. Now, classify the following method.
> Method : **[METHOD_SIGNATURE]**

Note that only the method signature is provided in these prompts (In the examples, we removed parameter types and return values for space reasons, but the full signature was provided to the model). Importantly, we do not define what constitutes a SOURCE or a SINK; instead, we rely entirely on the LLM's internal knowledge and capabilities. We issued each prompt five times per method using OpenAI's `gpt-4o-mini` model [33] via their API. We then selected the majority label as the final prediction and computed the accuracy with respect to the ground truth. Results are reported in Table 1.

Table 1. SOURCE/SINK Classification Accuracy.

Prompt	SOURCE Accuracy	SINK Accuracy	Overall Accuracy
Zero-Shot	68.40 %	100.00 %	79.78 %
One-Shot	98.27 %	99.23 %	98.61 %

As seen in Table 1, the one-shot prompt yields impressive results, with an overall accuracy of 98.6%. However, taint specification is inherently a three-class classification task, as the NEITHER label is the most common across the Android API framework [43]. We, therefore, extended our experiments to include the NEITHER category and re-ran the evaluations using only the better-performing one-shot prompt. We also evaluated a variant of the one-shot prompt in which the method signature is augmented with the official Android API documentation describing the method's functionality. The results are reported in Table 2.

Table 2. SOURCE/SINK/NEITHER classification accuracy.

Prompt	SOURCE Accuracy	SINK Accuracy	NEITHER Accuracy	Overall Accuracy
One-Shot	11.00%	88.00%	84.00%	68.00%
One-Shot + Doc	22.00%	89.00%	86.00%	72.00%

When using exactly the same one-shot prompt as before, the accuracy for SINK and NEITHER methods remains relatively high, although SINK accuracy is not as exceptional as in the two-class setting. However, the most striking observation concerns the SOURCE methods, whose accuracy drops sharply to 11%. Even when augmenting the prompt with API documentation, SOURCE accuracy remains low (22%), while the accuracy for SINK and NEITHER remains largely stable. Despite providing richer semantic context, the model continues to misclassify a substantial number of SOURCEs. These results highlight two important findings: ❶ SINK methods are generally easier to define (even for LLMs), as they are defined as Android API functions that send out one or more types of data [43], and sending data out can be factually determined. ❷ In contrast, as discussed in the introduction and confirmed by previous studies [17, 28, 43, 47], identifying what qualifies as a SOURCE method is substantially more difficult. The LLM often mislabels many NEITHER methods as SOURCEs.

These preliminary findings led us to reconsider the challenge of defining a unique, general-purpose notion of SOURCE for labeling Android methods. The LLM also appears to lack a clear definition of what should be considered a SOURCE method. Indeed, as previously discussed in Section 1, whether a method should be treated as a SOURCE in practice often depends on the specific analysis task and the end user's goals. Take, for instance (once again), the method `isMicrophoneMute()`. At first glance, it might appear to belong to the NEITHER category, as one could argue that it provides no information. However, in certain scenarios—or tasks—this method could indeed function as a SOURCE in a taint analysis. Imagine, for example, an attacker using the return value of `isMicrophoneMute()` to decide whether to begin recording the user and sending that information to an external server. In such a case, this method must be treated as a SOURCE to correctly capture the data flow within an Android APK. This task-specific nature of SOURCEs motivates the need for generating tailored lists of SOURCE methods that align with the requirements of a particular analysis.

Importantly, these findings do not imply that SINKs are unimportant; rather, they suggest that existing SINK specifications (e.g., those used by FlowDroid, SUSI, or CoDoC) or even an LLM-generated SINK list, obtained simply via the same prompting strategy we used in this preliminary experiment, may already be sufficiently reliable for many analyses. In practice, users may combine a task-specific SOURCE list with any of the generic SINK lists to perform taint analysis. At the same time, certain scenarios may benefit from task-specific SINK identification. For instance, an analyst investigating SMS-based exfiltration or proximity-based wireless channels may prefer highly focused SINK sets rather than broad, task-agnostic lists.

Overall, the preliminary findings motivate a task-driven approach in which TaskFlow focuses on the more challenging problem of SOURCE identification, while also enabling the generation of task-specific SINK lists when needed.

4 Approach

To address the need for task-specific taint analysis, we introduce TaskFlow, an approach that generates tailored taint specification for Android methods for use as input to static taint analyzer tools such as FlowDroid. Since our main

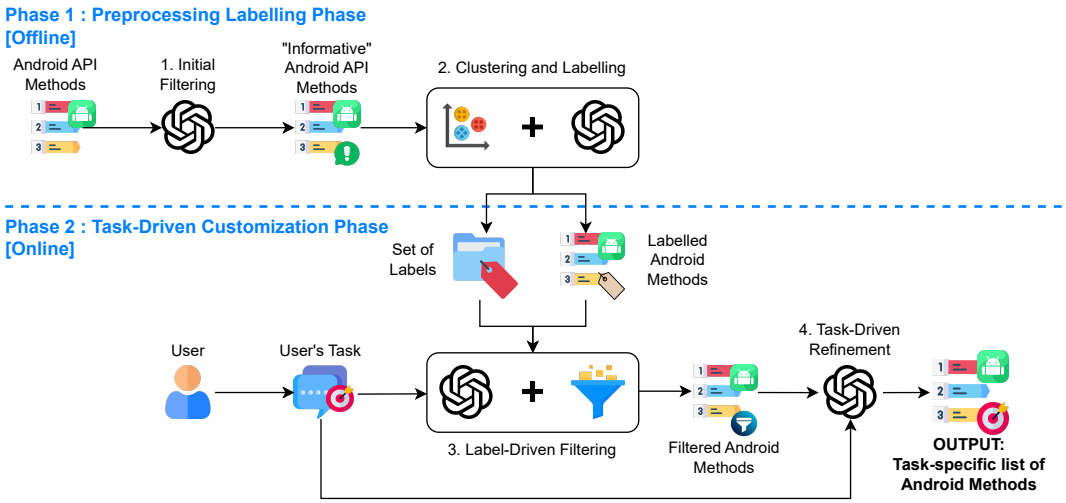

Fig. 1. Overview of the TaskFlow approach, illustrating the two main phases.

focus is on SOURCEs, we describe here in detail the TaskFlow pipeline for SOURCEs. However, the same pipeline can be used (with minor modifications) for SINKs, as we will show later in Section 5.7 (RQ6).

An overview of TaskFlow is illustrated in Figure 1. Users can describe their analysis task, problem, or goal in *natural language*, and TaskFlow will produce a customized list of relevant SOURCEs, removing the need to rely on overly broad pre-generated lists or to create one manually. TaskFlow comprises two main phases: ❶ a one-time **Preprocessing Labelling Phase**; and ❷ a **Task-Driven Customization Phase**. The *Preprocessing Labelling Phase* is executed only once by the authors to prepare the system by processing the entire Android API surface. Phase 1 needs to be re-run on the updated API surface only when a new Android version is released. After that, end users can directly apply the Task-Driven Customization Phase without additional preprocessing. Subsequently, the *Task-Driven Customization Phase* is executed each time a new user-defined task is provided. This second phase is designed for efficiency; for example, future work could integrate it into a web interface, allowing users to submit task descriptions and download the resulting task-specific SOURCE list file, aiming for practicality and ease of use.

This two-phase architecture addresses the computational challenges posed by the vast Android API surface. Processing the full API—which includes a massive number of methods [43]—at query time would be computationally intensive and costly, potentially taking over 30 hours and incurring significant expense, particularly when using proprietary models such as `gpt-4o-mini`(a discussion on the choice of the underlying LLM is provided later in Section 4.3). To overcome these limitations, the *Preprocessing Labelling Phase* analyzes the entire set of Android methods once and assigns preliminary labels. During the subsequent *Task-Driven Customization Phase*, only a small, relevant subset of methods—identified using the initial labels—needs to be re-evaluated in the context of the specific user task. This design enables a fast, lightweight, and cost-effective user experience, improving the real-world applicability of TaskFlow. An ablation study, presented later in Section 5.5, demonstrates the advantages of employing this two-phase strategy. The two phases and their implementation details are described in the following sections.

4.1 Phase 1: Preprocessing Labelling Phase

The objective of this initial phase is to preprocess the comprehensive set of Android API methods to facilitate the efficient generation of task-specific SOURCE lists in the second phase. This involves filtering irrelevant methods and assigning informative labels to potentially relevant ones. The steps are as follows:

❶ **Initial Filtering.** Our starting point is the list of public, documented, and concrete Android API methods identified by Samhi et al. [43], comprising 46 034 methods. However, many of these methods (e.g., utility functions) do not return potentially sensitive information and are thus unsuitable as SOURCEs for typical security or privacy analyses. To filter these out, we utilize an LLM to analyze the signature and documentation (i.e., the Javadoc) of each method, focusing specifically on its return value. Specifically, we prompted the LLM to ask whether the information returned by a method can represent a gain of information for an attacker (i.e., it could potentially be leveraged by an attacker), or not. Methods are assigned one of these labels: ADVANTAGE_GAIN if the return value could be exploited, or NO_INFORMATION otherwise. Following this analysis, 13,105 methods (28.51%) were categorized as ADVANTAGE_GAIN, while the remaining 32,857 methods (71.49%) were classified as NO_INFORMATION and excluded from subsequent steps.

To mitigate the risks of relying on the inherently probabilistic output of LLMs, we queried each method five times and used the majority label. To further increase confidence in our results, we manually inspected a statistically significant random sample of 96 methods, drawn uniformly from the full set of 46,034 methods (i.e., including both ADVANTAGE_GAIN and NO_INFORMATION), with a confidence level of 95% and a 10% margin of error. Upon inspection, we identified one method, *android.hardware.Camera$Parameters: java.lang.String getSceneMode()*, which was labeled as ADVANTAGE_GAIN but should likely be labeled as NO_INFORMATION, as the method simply retrieves the current scene mode setting of the camera (e.g., night mode or portrait mode). All remaining inspected methods were correctly labeled (i.e., we did not find methods mislabeled as NO_INFORMATION). However, it is important to note that while methods incorrectly retained as ADVANTAGE_GAIN can still be discarded during the task-driven refinement in Phase 2, methods incorrectly filtered out as NO_INFORMATION cannot be recovered. This limitation is an inherent consequence of the two-phase design and is discussed further in Section 4.3.

❷ **Clustering and Labelling.** At the end of the first sub-phase, the methods retained (i.e., those labeled ADVANTAGE_GAIN) undergo a labeling process to categorize the type of information potentially exposed by their return values. The idea is to associate each method with a label such as the following:

> { **"CONTACT"**: "By accessing contact data, an attacker could learn about the user's social connections and relationships, potentially enabling phishing or impersonation attacks." }

To achieve this categorization, our methodology starts with generating a numerical vector representation (embedding) for each method labeled ADVANTAGE_GAIN. These embeddings, designed to capture the semantic essence of each method, were derived from its signature and official documentation using OpenAI's `text-embedding-3-small` model [35]. Specifically, the method signature and the documentation were concatenated in that order, and the resulting text was used to generate the embeddings. Subsequently, with semantic embeddings generated for all relevant methods, we applied a clustering algorithm to group methods exhibiting similar semantic characteristics. We employed the K-Means algorithm [26] to partition the methods into distinct clusters based on their embedding vectors, with the intention of automatically grouping APIs likely to expose similar types of information or pose related categories of risks. Once these clusters of semantically similar methods were formed, the final step involved assigning a meaningful and descriptive label to each. For this, we leveraged an LLM. For each identified cluster, the LLM was prompted to generate a concise

LabelID (e.g., CONTACT_DATA) and a more detailed LabelDescription. This description was intended to explain the type of information generally exposed by methods within that cluster and the potential implications of such exposure. An illustrative example, similar to the "CONTACT_DATA" example presented above, was provided to guide the LLM's output format and the expected level of detail.

To determine the optimal number of clusters for the K-Means algorithm, we experimented with a range of cluster counts, from as few as 50 up to 1000, given the substantial dataset size of approximately 13 000 methods. We observed that higher cluster counts tended to produce privacy labels that were overly fine-grained and often repetitive, with several clusters containing only a small number of methods. For instance, clusters such as CAMERA_FACING_DIRECTION, CAMERA_RESOLUTION, and CAMERA_FLASH_MODE appeared as separate labels (and extremely specific), despite referring to closely related concepts. Conversely, lower cluster counts resulted in labels that were too coarse and generalized, reducing the granularity necessary for meaningful categorization. For example, a generalized label like PRIVATE_DATA might be assigned to diverse methods with varying implications, described as: "Return values may expose sensitive data used in various system operations, which could be leveraged for unauthorized access or manipulation of user data." Such labels, while broadly accurate, fail to convey the specific nature of the privacy risks involved. Through this empirical process, we found that 150 clusters provided an effective balance between specificity and generality, generating coherent and informative labels. This conclusion was further supported by the results of TaskFlow, as presented later in the Evaluation Section (see Section 5). Consequently, we adopted 150 as the final number of clusters for our analysis.

Similarly to what we have done in the first sub-phase, to increase confidence in the quality of the results, we manually inspected a statistically significant random sample of 96 methods. The sample was selected to achieve a 95% confidence level with a 10% margin of error. The goal is to determine whether the assigned cluster labels were coherent and semantically meaningful with respect to the inspected methods. The vast majority of labels were found to be appropriate. However, we identified two clear mislabelings: `<android.hardware.SensorManager: float[] getOrientation(float[], float[])>` (it computes the device's orientation angles using sensor data) and `<java.security.spec.ECPoint java.math.BigInteger getAffineX()>` (returns the X coordinate of an elliptic curve point). Both were labeled LOCATION, which is only weakly justified for the first case and clearly incorrect for the second, which is cryptographic in nature. Despite these two errors, the high accuracy observed in the manual inspection suggests that the automated labeling process is generally robust.

The output of this first phase is a refined set of Android API methods (only ADVANTAGE_GAIN ones), where each method is annotated with one label characterizing its return value. In our setup, the complete execution of the Preprocessing Labeling Phase over the Android API surface took a total of 16 hours and 43 minutes. However, as explained, this phase needs to be run only once each time a new Android version is released, making the long execution time acceptable for practical use of the tool. This labeled dataset serves as the foundation for the Task-Driven Customization Phase.

4.2 Phase 2: Task-Driven Customization Phase

In the second phase, the pre-labeled dataset (output from phase 1) and the user's natural language description of their specific task, problem, or analysis goal serve as input. As further discussed later in Section 4.3, this dataset represents a task-agnostic upper bound of potentially relevant methods, which Phase 2 refines by instantiating a concrete, task-specific notion of what constitutes a SOURCE. This phase returns a task-specific list of Android API SOURCEs aligned with their particular analysis objectives. The process unfolds through the following key steps:

❸ **Label-Driven Filtering.** An LLM processes the task description to identify the privacy labels from the curated set established in Phase 1 that are most pertinent to the described task. Leveraging this set of task-relevant labels, the system then filters the labeled Android API methods generated during the Preprocessing Labelling Phase, retaining only those methods associated with the selected labels. This significantly reduces the number of methods that need to be considered further for the final refinement.

❹ **Task-Driven Refinement** In the final step, the LLM performs a targeted analysis of the filtered set of methods. Given the user's original task description and the associated relevant labels, the LLM refines the list to ensure that the included methods are highly specific and directly actionable for the user's intended analysis. This step aims to eliminate any remaining methods that might be tangentially related but not directly crucial for the task at hand, resulting in a precise and effective list of `SOURCE`s.

4.3 Design Choices

In this subsection, we discuss our design choices and evaluate their consequences.

Consequences of the Two-Phase Pipeline. TaskFlow adopts a two-phase design that reflects a deliberate trade-off between conceptual generality, scalability, and practical applicability. While our core argument is that the notion of a `SOURCE` is inherently task-dependent, Phase 1 introduces a task-agnostic filtering step to remove methods that are unlikely to contribute to any security- or privacy-oriented taint analysis. Indeed, given the large size of the Android framework, an effective pre-filtering mechanism is necessary to eliminate clearly non-relevant methods, such as utility methods that would not contribute meaningfully to any task. Conceptually, Phase 1 defines a coarse, task-agnostic upper bound of potentially relevant methods, while Phase 2 refines this set by instantiating concrete task-specific notions of what constitutes a `SOURCE`. To perform this filtering, we draw on prior literature and the typical application of taint analysis, which targets methods that handle sensitive or security-relevant information. We therefore introduce the notion of `ADVANTAGE_GAIN` to retain all methods that could plausibly provide an attacker with some information or benefit under a broad range of reasonable tasks, that is, methods that are potentially sources. We acknowledge that some specific tasks, such as "does this app leak a random number?", do not fit the "attacker gains something" paradigm and are therefore outside the scope of the current filtering approach. However, we do not consider this a fundamental limitation of our approach; rather, it reflects the interests and assumptions of the taint-analysis community, where tools such as FlowDroid are typically used to reason about information flows that provide an attacker with meaningful information or advantage. This design choice explicitly prioritizes scalability and broad applicability over completeness for tasks outside this paradigm. As we will show later in an ablation study in RQ4 (see Sec 5.5), omitting the preprocessing phase leads to prohibitive runtimes and reduced precision when operating over the full Android API surface. The initial filtering step enables TaskFlow to remain computationally feasible while still preserving a large set of potentially relevant methods. Another consequence of the two-phase pipeline is that our approach is more robust to false positives than to false negatives in Phase 1. Methods incorrectly retained as `ADVANTAGE_GAIN` can still be discarded during task-driven refinement in Phase 2, whereas methods incorrectly filtered out as `NO_INFORMATION` cannot be recovered. This asymmetry—once again a consequence of the two-phase pipeline and a trade-off for scalability—represents a limitation of the approach and is mentioned later in Section 6 as a threat to validity.

LLM Choice. For the LLM components of our pipeline, we primarily employ `gpt-4o-mini` [34] by OpenAI, chosen for its favorable balance of accuracy, inference speed, and cost efficiency. The impact of model selection is analyzed later in RQ5 (Section 5.6), where we compare the proprietary `gpt-4o-mini` with six different open-source LLMs: `gpt-oss-20b` [36], `deepseek-r1-32b` [9], `qwen3-30b` [49], `gemma3-27b` [13], `phi3-14b` [1], and `llama3.1-8b` [29].

All open-source models were executed locally on an NVIDIA RTX5000 ADA GPU via the `ollama` [32] deployment framework. For each model family, we selected the largest variant (in terms of parameter count) that could be run within our available hardware resources.

5 Evaluation

In this section, we evaluate our approach TaskFlow addressing the following research questions (RQs):

- **RQ1:** How effective is TaskFlow at generating task-specific lists of `SOURCE` methods for Android taint analysis?
- **RQ2:** How does TaskFlow compare against existing predetermined and generic `SOURCE` lists from the literature?
- **RQ3:** How does the granularity of the user-defined task description affect the precision and completeness of the `SOURCE` lists generated by TaskFlow?
- **RQ4:** How does removing the Preprocessing Labelling Phase affect the performance and computational cost of TaskFlow?
- **RQ5:** How does the choice of the underlying LLM impact the effectiveness of TaskFlow?
- **RQ6:** How effective is TaskFlow at generating task-specific lists of `SINK` methods for Android taint analysis?
- **RQ7:** How does FlowDroid perform on real-world apps when using TaskFlow-generated taint specifications compared to existing ones (e.g., SuSi)?

5.1 Evaluation Setup

Our evaluation follows a task-driven approach: for each RQ, we define specific analysis scenarios and assess whether TaskFlow produces relevant and precise `SOURCE` and `SINK` lists aligned with the intended task. Detailed task definitions and experimental procedures are provided for each corresponding RQ.

Metrics. We evaluate the quality of the generated lists using standard classification metrics, namely Precision, Recall, and F1-score, computed against task-specific ground-truth datasets. These metrics allow us to assess both the correctness and completeness of the generated specifications.

Manual Analysis. In several parts of our evaluation, we complement automated metrics with manual inspection of the results (e.g., to assess method relevance). Given the large size of the Android API and the number of generated outputs, exhaustively analyzing all instances is impractical. Therefore, following common practice in the literature, we rely on statistically significant random sampling to perform manual analysis while maintaining a high level of confidence in our observations. Specifically, we compute the required sample size using standard statistical estimation techniques, based on a 95% confidence level and a 10% margin of error. Such configurations are commonly adopted in prior Android related studies [2, 3, 20, 43] and these parameter values ensure that our manual analysis is both feasible (in a reasonable time) and statistically representative of the entire dataset. The sample size n is determined using the following formula:

$$n = \frac{Z^2 \cdot p \cdot (1 - p)}{e^2}$$

where Z is the z-score corresponding to the desired confidence level (1.96 for 95%), p is the estimated proportion (set to 0.5 to maximize variance and ensure a conservative estimate), and e is the margin of error (0.1 in our case). The resulting sample sizes used in our experiments are consistent with standard practice and commonly used sample size calculators.

Baselines. In RQ2 and RQ7, we compare TaskFlow against existing state-of-the-art approaches for taint specification, including SuSi [5], CoDoc [43], and DocFlow [46], which provide generic (task-agnostic) `SOURCE`/`SINK` lists.

Implementation Details. Unless otherwise specified, we use `gpt-4o-mini` as the underlying LLM in our pipeline. In RQ5, we compare several LLMs to analyze how this choice impacts the effectiveness of TaskFlow.

5.2 RQ1: How effective is TaskFlow at generating task-specific lists of SOURCE methods for Android taint analysis?

This section evaluates the effectiveness of TaskFlow in generating task-specific lists of SOURCEs for Android taint analysis. The core idea of our approach is to produce SOURCE lists tailored to specific analysis tasks and user goals. To assess TaskFlow's capability to include relevant task-specific methods, we define a set of tasks and evaluate whether TaskFlow correctly identifies and incorporates these methods in its output. The evaluation process involves the following steps:

Tasks Definition. As a starting point in our evaluation process, we need to define some tasks and corresponding task-specific SOURCEs to evaluate our approach, TaskFlow. Specifically, we aim to determine whether TaskFlow can correctly include these task-specific methods in its final output list. The evaluation process follows the steps outlined below. First, we defined three tasks, ensuring diversity in their specificity—ranging from broad and more "common" goals, such as inferring the device's location, to more precise cases, such as those addressed by ClipboardScope. The tasks we defined (and which are used as input in the second phase of TaskFlow) are as follows:

> **Task 1 [Location]:** Which APIs return data that an attacker could use to infer the location of the user/device, even indirectly?

> **Task 2 [Microphone]:** Which APIs access microphone-related data (e.g., audio input state or stream), which could be useful for an attacker aiming to eavesdrop or gather information about the user's environment without their consent?

> **Task 3 [Clipboard]:** Which APIs interact with clipboard content, potentially exposing sensitive information such as passwords, addresses, or personal data [7]?

It is important to highlight that these three tasks were selected to demonstrate the flexibility of TaskFlow across realistic and diverse scenarios, rather than to claim completeness. Indeed, it would be impossible to create an exhaustive list of tasks since users can define their own, making the space of possible tasks (i.e., TaskFlow inputs) inherently unbounded. Our choice therefore includes intuitive cases like location tracking and literature-based scenarios such as clipboard misuse (ClipboardScope) as illustrative, not exhaustive, examples.

Ground Truth Preparation. Once the three tasks were defined, we had to build a ground-truth dataset of SOURCE methods per task to assess TaskFlow's ability to generate task-specific SOURCE method lists. While for Task 3 it is possible to rely on the ground-truth list built by the authors of ClipboardScope [7], there is no such dataset for Tasks 1 and 2. Therefore, for both of these tasks, we relied on manual labeling to build our ground-truth. However, since it is challenging to generate such lists from the entire set of around 45k API methods manually, we used the set of SOURCEs in the CoDoc Ground-Truth (231 SOURCE methods) as a "starting set" for the manual labeling.

Three experts in Android app analysis independently analyzed and manually labeled each method from this set. The three annotators consisted of two postdoctoral researchers and one PhD student, all with extensive experience in Android static analysis and taint analysis. For every method, they carefully reviewed the official Android documentation, analyzing the behavior and usage context of each API to determine its relevance. Each method was classified as either

relevant to Task 1, Task 2, Task 3, or as unrelated to all tasks. The initial agreement among the three annotators was substantial, with a Fleiss' Kappa inter-rater reliability (IRR) score of 0.77. For methods where there was no unanimous agreement, the experts held discussions until a consensus label was reached. This process resulted in a ground-truth dataset for each task. A method is labeled as SOURCE for a given task if the experts explicitly identified it as relevant to that task; otherwise, it is labeled as NOT_SOURCE. As for Task 3, no methods were manually identified as SOURCE by the experts; we incorporated the SOURCE methods used by the authors of ClipboardScope [7]. These methods were automatically labeled as SOURCE for Task 3 and as NOT_SOURCE for Tasks 1 and 2. In total, the final ground-truth set includes 46 SOURCE methods for Task 1, 4 for Task 2, and 3 for Task 3. All remaining methods in the labeling set (i.e., those not marked as SOURCE for a given task) are considered NOT_SOURCE for that task.

Evaluation Metrics. We evaluated TaskFlow on the three defined tasks using task-specific ground truth datasets previously constructed. We then computed Precision, Recall, and F1-Score based on the following definitions of the confusion matrix components:

- **True Positive (TP):** A method manually labeled as SOURCE and correctly included in TaskFlow's output.
- **False Positive (FP):** A method manually labeled as NOT_SOURCE but included in TaskFlow's output.
- **False Negative (FN):** A method labeled as SOURCE in the ground truth but missing from TaskFlow's output.
- **True Negative (TN):** A method labeled as NOT_SOURCE and not present in TaskFlow's output.

It is important to note that we first report results using only our task-specific ground truth datasets, as Recall can only be computed with respect to our labeled data (since computing overall Recall would require labeling all 45,000+ API methods). However, TaskFlow may also identify SOURCE methods that are not present in our ground truth dataset. We address the potential impact of these additional methods later in this section under the 'Notes on Precision' paragraph. The evaluation results over the task-specific ground truth datasets are summarized in Table 3.

Table 3. TaskFlow Performance in Task-Specific SOURCE List Generation.

Task	Precision	Recall	F1-Score
Task1	0.82	0.87	0.84
Task2	1.00	0.75	0.86
Task3	1.00	1.00	1.00

Results. Table 3 shows that TaskFlow performs well across all three tasks. In **Task 1 (Location)**, TaskFlow achieved a balanced performance with a precision of 0.82 and recall of 0.87, resulting in an F1-Score of 0.84. Upon further inspection, the slightly lower precision of Task 1 actually reflects some borderline cases where the experts initially disagree before reaching an agreement. However, the relatively high recall indicates that TaskFlow captured a large proportion of the location-related methods. For **Task 2 (Microphone)**, it maintained perfect precision (1.00), indicating all identified methods were correct, though a recall of 0.75 suggests that a microphone-related SOURCE was missed. Lastly, in **Task 3 (Clipboard)**, TaskFlow achieved perfect precision and recall (1.00), leading to an F1-Score of 1.00. While these results may appear unexpectedly ideal, they can be explained by the characteristics of the task and the way the dataset was constructed. Task 3 involved only three SOURCE methods, based on prior work [7], making it a small, well-defined target. The limited number of relevant APIs, combined with their clear and unambiguous functionality, likely reduced false positives and negatives. Additionally, clipboard access APIs are few and have explicit roles, which makes them easier

for TaskFlow to identify correctly. Nonetheless, we further investigate this later in the 'Notes on Precision' paragraph, where we manually assess methods returned by TaskFlow outside the ground truth.

Notes on Precision. The precision values reported so far are based on the task-specific ground truth datasets that we constructed following the procedure described earlier in this section. However, TaskFlow also outputs additional methods not included in our ground truth. This is expected, since to build the ground-truth datasets we started from a reduced set of API methods (those in the CoDoc Ground Truth) rather than evaluating across the full set of 45000+ Android APIs.

Therefore, to further assess the quality of TaskFlow's output, we conducted a manual analysis of these additional methods. Specifically, we manually reviewed the methods not present in the ground truth and labeled each as either a True Positive (i.e., relevant to the task) or a False Positive (i.e., not relevant). We then incorporated these findings to recompute precision. The updated results are presented in Table 4. Note that this extended evaluation can only affect precision, as recalculating recall would require exhaustive labeling of the entire Android API set of 45000+ methods, which is, of course, ① highly challenging; and ② the reason why we develop a technique to automate this very process.

Table 4. Extended Precision Evaluation Based on Manual Analysis of Additional SOURCEs Outside the Ground Truth Set.

Task	# Additional Methods	Precision on Ground-Truth	# Additional True Positive	# Additional False Positives	Recomputed Precision
Task1	151	0.82	101	50	0.71
Task2	16	1.00	14	2	0.89
Task3	9	1.00	6	3	0.75

As shown in Table 4, the precision values slightly decrease after accounting for the additional methods. However, two important observations should be emphasized:

(1) **Nature of False Positives.** Even when TaskFlow includes false positives, they are often not entirely irrelevant to the task. For instance, in Task 2 (Microphone), one method identified by TaskFlow was `<android.media.MicrophoneInfo: int getDirectionality()>`. While this method may not be strictly categorized as a SOURCE for Task 2, it still pertains to microphone metadata, thus maintaining thematic relevance. Such false positives might be less problematic than including entirely unrelated methods (e.g., those concerning location or networking), which might instead occur when relying on a generic, broad list of SOURCEs.

(2) **Compactness and Practical Impact.** Despite the presence of some false positives, TaskFlow generates much smaller, focused lists compared to conventional, generic SOURCE sets, which often include 10000–15000 entries. The reduced size of the task-specific lists makes them more interpretable and manageable for developers and analysts, potentially streamlining downstream analysis.

Answer to RQ1: TaskFlow effectively generates task-specific SOURCE lists, achieving high F1-scores across all tasks. Beyond ground-truth data, TaskFlow maintained high precision, with most additional methods relevant or closely related to the task. Its lists are smaller and more focused than generic SOURCE sets, improving interpretability and actionability.

5.3 RQ2: How does TaskFlow compare against existing predetermined and generic SOURCE lists from the literature?

Unlike RQ1, where we evaluated TaskFlow against task-specific ground truth, direct comparison with existing general-purpose SOURCE lists is not straightforward. This is because TaskFlow generates SOURCE lists tailored specifically to particular tasks, while existing approaches [5, 43, 46] are designed as task-agnostic and predetermined lists. While some of these approaches associate SOURCEs with coarse semantic categories (e.g., SuSi assigns labels such as LOCATION_INFORMATION), these categories are still general-purpose and do not align with user-defined tasks in the same way as TaskFlow does. For instance, there is no SuSi category that directly corresponds to Task 2 (Microphone) or Task 3 (Clipboard) from RQ1. We return to this limitation later in this section, where we explicitly analyze the mismatch between SuSi categories and task-driven specifications.

As a consequence, directly evaluating existing approaches against the task-specific ground truth used in RQ1 would be both unfair and potentially misleading. Unfortunately, to the best of our knowledge, there is no independent, objective benchmark that enables a large-scale, tool-agnostic comparison of Android SOURCE lists across all APIs and analysis goals. To enable a meaningful comparison, we therefore need to adopt a task-agnostic evaluation strategy. Specifically, as described in Section 4, TaskFlow labels Android methods as either ADVANTAGE_GAIN or NO_INFORMATION during preprocessing, filtering out irrelevant methods to produce task-specific SOURCE lists consisting exclusively of ADVANTAGE_GAIN methods. For RQ2, we apply this same labeling scheme to the methods in existing SOURCE lists. In the absence of an external benchmark, adopting a shared operational definition of what constitutes an informative potential SOURCE is the only practical way to perform a large-scale, automated comparison across all methods and tools. We therefore use the ADVANTAGE_GAIN criterion as a common reference point, allowing us to assess the extent to which existing SOURCE lists include methods that are unlikely to contribute meaningful data flows when used with taint analysis tools such as FlowDroid.

We analyze the following approaches:

- **SUSI:** A feature-based machine learning approach by Arzt et al. [5] for classifying Android SOURCEs and SINKs using code features.
- **CoDoc:** Proposed by Samhi et al. [43], CoDoc combines code and documentation to improve SOURCE/SINK identification.
- **DocFlow:** Tileria et al. [46] introduce DocFlow, which extracts taint specifications from documentation and claims improved accuracy over SUSI. We evaluate both its generated SOURCE list and the manually labeled training set used to train their model.

The results are shown in Table 5.

Table 5. Comparison of SOURCE List Composition.

Approach	% ADVANTAGE_GAIN	% NO_INFORMATION
TaskFlow*	100%	0%
SUSI	55.24%	44.76%
CoDoc	47.87%	52.13%
DocFlow - Result	34.46%	65.54%
DocFlow - Training	40.18%	59.82%

*TaskFlow, by design, only return methods with the ADVANTAGE_GAIN label.

Analysis of Results: The data in Table 5 highlights a clear distinction in performance. By design, TaskFlow returns only methods satisfying this criterion, while existing approaches exhibit between 44.76% and 65.54% of methods labeled as NO_INFORMATION. It is worth noting that DocFlow while positioning itself as an improvement over SUSI, exhibits the highest percentage of NO_INFORMATION methods in its generated SOURCE list (65.54%). This may be attributed to its manually labeled training set, which itself contains a high percentage of NO_INFORMATION methods (59.82%). However, these results should be interpreted with care. Methods labeled as NO_INFORMATION are considered unlikely to be of practical interest under the ADVANTAGE_GAIN criterion, which we adopt as a common reference point reflecting typical assumptions in static taint analysis and prior literature. Accordingly, these numbers are not an absolute measure of correctness. Rather, they allow us to assess the extent to which existing SOURCE lists include methods that are unlikely to contribute meaningful data flows when used with taint analysis tools such as FlowDroid (see Section 4.3).

To ensure robustness and account for the probabilistic nature of LLM outputs, each method was queried five times and labeled by majority vote. In addition, we manually inspected statistically significant random samples for each approach using a 95% confidence level and a 10% margin of error—analyzing 96 methods for CoDoc, 96 for SUSI, 96 for DocFlow, and 89 from DocFlow's training set (similarly to what we did in Section 4). These samples contained methods both labeled as ADVANTAGE_GAIN and NO_INFORMATION since the methods are directly sampled from all the SOURCEs generated by each approach. Our manual inspection confirmed that the LLM-generated labels were almost always correct. Only a few exceptions were found: two SUSI methods and one CoDoc method were incorrectly labeled as NO_INFORMATION instead of ADVANTAGE_GAIN. In contrast, all labels in the DocFlow samples were found to be correct. These minimal discrepancies suggest that, while the exact percentages in Table 5 may vary slightly, the overall trends are both consistent and reliable. To bring additional confidence to our findings—and to further underscore the prevalence of NO_INFORMATION methods in existing SOURCE lists, we now provide a few illustrative examples found while manually analyzing the random samples of methods. These examples highlight methods that were correctly labeled as NO_INFORMATION by the LLM but were nevertheless included as SOURCEs by the corresponding approaches. For example, the CoDoc list included `<RecyclerView.LayoutManager: int getMinimumHeight()>`, while SUSI listed `<ArcMotion: float getMaximumAngle()>` and `<Canvas: int getWidth()>`. Similarly, DocFlow's training set included methods such as `<DecimalFormatSymbols: char getPercent()>` and `<Math: int getExponent(float)>`, which likely contributed to its elevated proportion of NO_INFORMATION results in the output list.

On fairness of comparison. It is important to note that this evaluation is not perfect, as it relies on TaskFlow's own ADVANTAGE_GAIN and NO_INFORMATION criterion to determine what constitutes an informative SOURCE. Nevertheless, in the absence of an independent and widely accepted benchmark for this notion, this criterion provides a reproducible and scalable basis for comparison. Moreover, the observed trends are consistent with prior work reporting that existing SOURCE lists frequently contain methods that may be irrelevant, corresponding to NEITHER APIs [17, 28, 43, 47]. The inclusion of such non-informative methods has critical implications. It increases the false alarm rate in taint analysis, may burden security analysts with irrelevant data flows, and dilutes attention from genuine threats.

The main challenge in comparing TaskFlow with existing approaches lies in the difference between task-driven and task-agnostic specifications. It is true that SuSi provides semantic categories for its SOURCE list (e.g., LOCATION_INFORMATION). However, there are two main issues in comparing against SuSi categories: ❶ SuSi categories are general-purpose and do not necessarily align with a user-defined task in the same way as TaskFlow. For example, there is no SuSi label corresponding to Task 2 (Microphone) or Task 3 (Clipboard) from RQ1; ❷ There is no one-to-one mapping between SuSi categories and a potential user-defined task. To illustrate this mismatch, we analyzed Task 1 (LOCATION) from RQ1. We took all the methods returned by TaskFlow for this task and checked whether they are present in the SuSi source

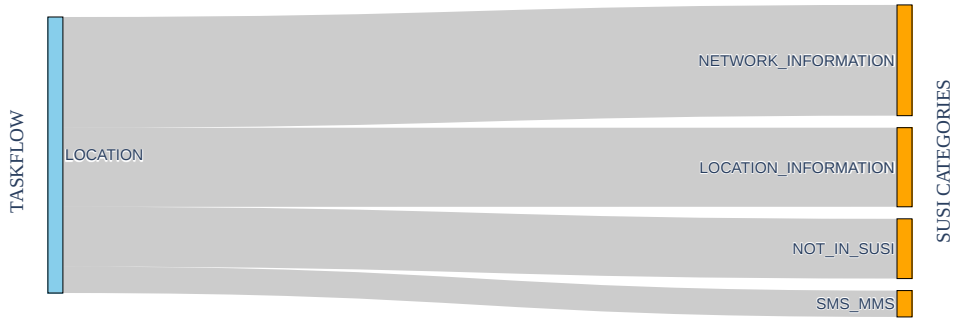

Fig. 2. Sankey diagram showing how methods associated with TaskFlow's task-driven label (LOCATION) distribute across SuSi categories, illustrating that category-based filtering does not yield a one-to-one mapping.

list and their corresponding SuSi categories. Figure 2 presents a Sankey diagram illustrating how methods associated with our task-driven label distribute across SuSi categories. For instance, methods that allow inferring location from network data do not belong to the LOCATION category in SuSi. Notably, some methods returned by TaskFlow are not included at all in the SuSi list, meaning that our approach can actually find potential leaks missed by existing lists (which we will further explore in RQ7).

This highlights that, while SuSi categories provide some semantic grouping, they are insufficient to fully capture the task-specific precision that TaskFlow achieves, underscoring the inherent challenge in performing a fully fair comparison.

Answer to RQ2: In the absence of an independent and widely accepted benchmark to compare task-specific and task-agnostic lists, we relied on the definition of ADVANTAGE_GAIN for a comparison. Existing approaches may include up to 65.54% NO_INFORMATION methods, potentially adding noise and analyst overhead. In contrast, TaskFlow returns only ADVANTAGE_GAIN methods by design, enhancing the focus and usefulness of the SOURCE lists.

5.4 RQ3: How does the granularity of the user-defined task description affect the precision and completeness of the SOURCE lists generated by TaskFlow?

This research question investigates the sensitivity of TaskFlow to the level of detail in the user-defined task description. Since TaskFlow relies on natural language input to generate task-specific SOURCE lists, it is important to understand how varying the specificity of these inputs affects its performance. To explore this, we focus on **Task 1 (Location)** and evaluate TaskFlow using three prompts with varying levels of detail:

Highly Detailed: Which Android API methods return information which could reveal the user's physical location? This includes precise location (e.g., GPS, Wi-Fi, cell tower data), approximate location (e.g., IP address, network configuration), or broad geographic indicators such as timezone, country, SIM/network region, settings. Include any methods that could assist an attacker in estimating user location at any granularity, even indirectly.

Medium Detailed: Which APIs return data that an attacker could use to infer the location of the user/device, even indirectly? (This is the same prompt used in RQ1.)

Lowly Detailed: Which APIs return location data?

We then run TaskFlow independently with each of these task descriptions and compare the resulting sets of identified SOURCEs against our manually annotated ground truth. The medium-detail prompt serves as the baseline (used in RQ1), and we examine how TaskFlow's performance shifts when provided with more or less detailed instructions.

Table 6. Impact of Task Description Granularity on TaskFlow Performance (Task 1).

Details Level	Precision	Recall	F1-Score
High	0.84	0.83	0.84
Medium	0.82	0.87	0.84
Low	1.00	0.15	0.26

Results and Analysis: As evident from Table 6, the performance of TaskFlow is affected by the granularity of the task description. The recall—and consequently the F_1 score—drops significantly when the low-detail prompt is used, with a recall of just 0.15, resulting in a poor F1 score of 0.26. The main reason is that, with this prompt, the LLM identified only a narrow subset of location-related APIs, i.e., the most direct ones (e.g., `getLatitude()` and similar), while missing many APIs that could indirectly reveal location, such as those related to network access. The perfect precision indicates that the methods identified using the low-detail prompt were highly relevant, but the low recall highlights a significant lack of coverage. In contrast, the highly and medium detailed prompts yielded similar and relatively high F1-scores (0.84). The high-detail prompt resulted in slightly better precision (0.84 vs. 0.82 for medium), while the medium-detail prompt achieved higher recall (0.87 vs. 0.83 for high).

To better understand the differences between the sets of methods returned by these prompts, we measured their overlap and visualized it using a Venn diagram. Figure 3 illustrates the comparison of the SOURCE sets identified by TaskFlow for each level of detail.

The Venn diagram (Figure 3) shows that all the methods identified by the low-detail prompt are also included in the results of both the medium and high-detail prompts. As previously mentioned, these methods correspond to APIs from which the device/user's location can be retrieved directly.

Regarding the remaining methods, there is substantial overlap between those identified by the medium and high-detail prompts, with each also returning a few unique methods. Upon closer examination of these unique methods, we observed that both sets contained APIs relevant to the task. For instance, the high-detail prompt—explicitly mentioning time and timezone—led to the inclusion of related APIs. Conversely, the medium-detail prompt uniquely identified some lower-level sensor and signal-related APIs (such as GNSS/GPS measurements and telephony signal strength), which were not explicitly referenced in the high-detail version.

This suggests that different levels of detail in the prompt can guide TaskFlow to emphasize slightly different aspects of the task. As future work, we may, for example, consider leveraging the LLM itself to refine or expand upon a user-provided task description, potentially improving performance. However, in this paper, we aimed to assess the LLM's ability to work directly with a more natural, human-like task definition.

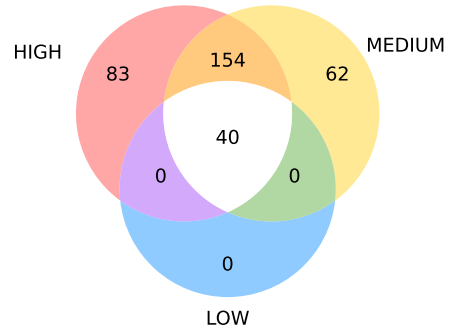

Fig. 3. Venn diagram illustrating the number of SOURCE methods identified by TaskFlow for varying levels of detail in the task description (High, Medium, and Low).

Answer to RQ3. Task description granularity significantly affects TaskFlow's performance. Low-detail prompts achieve high precision but very low recall, while medium and high-detail prompts better balance precision and recall, leading to more effective identification of task-specific SOURCEs.

5.5 How does removing the Preprocessing Labelling Phase affect the performance and computational cost of TaskFlow? (Ablation Study)

As discussed in Section 4, Phase 1 was introduced to address scalability and efficiency challenges. Given the large Android API surface (45k methods), analyzing all methods at runtime per user task is computationally expensive and inefficient. The Preprocessing Labelling Phase mitigates this by (1) filtering irrelevant methods and tagging sensitive ones as ADVANTAGE_GAIN, and (2) assigning privacy labels to describe return value types. This enables faster, more targeted analysis in Phase 2. This RQ evaluates the necessity of this preprocessing step via an ablation study, measuring performance and runtime when skipping Phase 1 and applying task-specific refinement directly over the entire Android API surface. To address this RQ, we ran TaskFlow in two configurations:

- **Full pipeline:** Both Phase 1 and Phase 2, as detailed in Section 4.
- **Ablated pipeline:** Phase 1 was skipped, and task-specific refinement was applied directly to the initial set of approximately 45k methods.

As before, we relied on our manually annotated ground truth. In addition to the F1 score, we also measured the time required for the entire process. The F1 score, elapsed time, and the differences between the two configurations are reported in Table 7. To quantify the changes, we also computed the percentage increase or decrease in both execution time and F1 score using the following formula:

$$\text{Increase/Decrease} = \frac{V_{\text{ablated}} - V_{\text{original}}}{V_{\text{original}}} \times 100$$

where V represents the value of interest, which can be either the F1 score or the execution time.

Table 7. Comparison of execution time and F1-Score for the full TaskFlow pipeline and the ablated version (without Phase 1).

Task	Metric	TaskFlow [Original]	TaskFlow [Ablated]	Reduction/Increase [Percentage]
Task1*	Time	44 min	29h 43min	+3952%
	F1-Score	0.84	0.46	-45.24%
Task2	Time	21 min	31h 32min	+8909%
	F1-Score	0.86	0.75	-12.79%
Task3	Time	31 min	29h 6min	+5532%
	F1-Score	1.00	0.55	-45.00%

**Task1 here is referring to the Medium Detailed version (same as RQ1).*

Table 7 highlights the critical role of the Preprocessing Labelling Phase. Skipping Phase 1 led to a dramatic and consistent increase in runtime—up to 90× slower—with execution times rising from minutes to 30 hours. This confirms Phase 1's effectiveness at filtering irrelevant methods, significantly reducing the workload for task-specific refinement.

F1-score also declined across all tasks, primarily due to reduced precision, since recall remained largely unchanged (i.e., the filtering step does not cause SOURCEs to be missed). We report only the F1 score because it effectively summarizes overall accuracy by balancing precision and recall, which avoids redundancy since recall remains stable. While the drop in F1 score for Task 2 was relatively modest; the extreme runtime increase still makes preprocessing essential for practical use. Additionally, while not directly shown in the table, this preprocessing also significantly reduces financial costs when using proprietary LLMs, as the full API set needs to be processed only once. These findings strongly support the necessity of the complete TaskFlow pipeline. The Preprocessing Labelling Phase not only ensures runtime scalability but also preserves high accuracy by narrowing the focus to a smaller, more relevant set of methods.

Answer to RQ4. The Preprocessing Labelling Phase is essential for both the efficiency and accuracy of TaskFlow. Removing it leads to extreme runtime increases (up to 9000%) and significant F1 score drops due to lower precision. The complete pipeline is crucial for practical and effective SOURCE list generation.

5.6 RQ5: How does the choice of the underlying LLM impact the effectiveness of TaskFlow?

Up to this point, all evaluations have relied on OpenAI's `gpt-4o-mini`. However, our LLM-based pipeline is model-agnostic and can, in principle, operate with any instruction-following LLM. In this RQ, we quantify how the choice of the underlying model impacts TaskFlow's effectiveness by comparing `gpt-4o-mini` against six open-source models (see Section 4.3 for details). To this end, we re-ran TaskFlow on a representative task using each model, keeping prompts, preprocessing steps, and evaluation metrics identical. We focus on Task 1 (Location) from RQ1, as it represents a realistic and challenging setting that includes both direct and indirect location signals. As in RQ1, we evaluated the generated SOURCE lists against our manually constructed ground truth and computed the F1-Score. The results are reported in Table 8.

Table 8 shows that the choice of LLM substantially affects the quality of the generated task-specific SOURCE lists. `gpt-4o-mini` achieves the highest F1-Score (0.84), motivating our choice of using it throughout the rest of the paper. Among the tested open-source alternatives, `gpt-oss-20b` performs best (F1-Score 0.78), while the other models achieve

Table 8. Impact of the underlying LLM on TaskFlow effectiveness (Task 1: Location), measured as F1-Score against the task-specific ground truth. Models marked with * failed to complete the pipeline due to repeated violations of the required output format; we report F1-Score as 0 in these cases.

Model	F1-Score
gpt-4o-mini	0.84
gpt-oss-20b	0.78
deepseek-r1-32b	0.71
qwen3-30b	0.71
gemma3-27b	0.68
phi3-14b	0.00*
llama3.1-8b	0.00*

lower but roughly comparable scores. It is important to note that, given the limited size of the ground truth, even a few misclassified methods can lead to noticeable changes in F1-Score.

Two models (`phi3-14b` and `llama3.1-8b`) did not complete the pipeline. Despite a retry mechanism, they repeatedly failed to produce the expected structured output format, preventing TaskFlow from generating a final list of task-specific sources. It is possible that different prompt designs or larger variants within the same model families could yield different results. Nevertheless, the primary goal of this RQ is not to exhaustively benchmark all available LLMs, but to motivate the choice of the model used for the rest of the paper (`gpt-4o-mini`). Indeed, even testing other, more powerful (and more expensive) closed-source models might actually achieve higher performance.

However, given the balance between reliability, performance, and cost, `gpt-4o-mini` remains our preferred choice. We believe that our choice of `gpt-4o-mini` represents a reasonable trade-off between effectiveness and cost.

Answer to RQ5. The choice of the underlying LLM has a clear impact on TaskFlow's effectiveness: `gpt-4o-mini` achieves the highest F1-Score on Task 1 (0.84), while the best-performing open-source model reaches 0.78.

5.7 RQ6: How effective is TaskFlow at generating task-specific lists of SINK methods for Android taint analysis?

So far, we have focused primarily on generating task-specific lists of SOURCE methods. As discussed in the Introduction (Section 1) and Preliminary Experiments (Section 3), SOURCEs represent the most challenging case due to the lack of a unique definition for what qualifies as a SOURCE, whereas SINKs are generally more straightforward since they represent exit points. However, there may be scenarios where a user wants to focus on a specific type of SINK (e.g., only SINKs that send data via SMS). Therefore, in this RQ we explore the performance of TaskFlow in generating task-specific lists of SINKs.

To achieve this, we reuse the two-phase pipeline introduced earlier with minor modifications. For example, instead of having the preprocessing phase classify methods into ADVANTAGE_GAIN and NO_INFORMATION, we ask the LLM to distinguish between POTENTIAL_EXIT_POINT and NO_INFORMATION. The preprocessing step is then used to remove methods that do not match the definition of SINKs. The remainder of the pipeline—clustering, labeling, and final refinement—remains unchanged, with the prompts adapted appropriately for SINKs instead of SOURCEs.

Given the absence of an existing ground truth, we followed the same strategy as in RQ1. First, we defined three tasks corresponding to specific SINK scenarios, similar to what we did previously with SOURCEs. These three tasks serve as input to TaskFlow:

Task 1 [Telephony Outbound Channels]: Which Android APIs enable sending data through SMS, MMS, or other telephony-based outbound channels that could be abused to exfiltrate sensitive information?

Task 2 [Network Outbound Channels]: Which Android APIs allow transmitting data over the network (e.g., HTTP, TCP/UDP, sockets, or other Internet-based channels) that could be used to exfiltrate sensitive information outside the device?

Task 3 [External and Shared Storage]: Which Android APIs allow writing or exporting application data to external storage, shared media directories, document providers, or other writable locations accessible by external apps or processes?

For each task, we relied on manual labeling to construct the task-specific ground truth. As in RQ1, we used the CoDoc Ground-Truth (130 SINK methods) as a starting point for manual labeling. This resulted in a task-specific ground truth: 11 methods for Task 1, 17 for Task 2, and 9 for Task 3. All remaining methods (i.e., those not marked as SINK for a given task) were labeled as NOT_SINK for the corresponding task. We then used the same evaluation metrics as in RQ1: Precision, Recall, and F1-Score, computed over the task-specific ground truth for each of the three SINK tasks. The results are reported in Table 9.

Table 9. TaskFlow Performance in Task-Specific SINK List Generation.

Task	Precision	Recall	F1-Score
Task 1 [Telephony]	0.85	1.00	0.92
Task 2 [Network]	0.84	0.88	0.86
Task 3 [Storage]	0.88	0.78	0.83

Table 9 shows that TaskFlow performs well in generating task-specific SINK lists, with F1-scores ranging from 0.83 to 0.92. Task 1 achieves perfect recall (1.00) and high precision (0.85), indicating that nearly all relevant telephony-related methods were identified. Task 2 reaches an F1-score of 0.86, though some false positives exist, such as `<android.hardware.location.ContextHubClient: int sendMessageToNanoApp(...)>`, which only communicates with a local NanoApp and should likely be labeled NOT_SINK. Task 3 shows slightly lower recall (0.78), but high precision (0.88), meaning a few relevant storage SINKs were missed, yet the generated list remains accurate and focused. Overall, TaskFlow effectively produces task-specific SINKs with only minor edge-case errors.

These results suggest that users can leverage TaskFlow to also generate compact, targeted lists of SINKs, which can later be combined with task-specific SOURCEs to enable task-specific end-to-end taint analysis on real-world apps, as explored in RQ7 (Section 5.8).

Answer to RQ6. TaskFlow effectively generates task-specific SINK lists, achieving high precision and recall across all tasks. False positives are minimal, and the produced lists are compact, focused, and suitable for task-driven taint analysis.

5.8 RQ7: How does FlowDroid perform on real-world apps when using TaskFlow-generated taint specifications compared to existing ones (e.g., SuSi)?

So far, we have primarily evaluated TaskFlow at the level of the generated SOURCE and SINK specifications, without analyzing its impact on real-world applications. However, a crucial question remains unanswered: how do these differences manifest when a static taint analysis tool, such as FlowDroid, is executed on real-world Android apps?

To address this gap, this research question evaluates TaskFlow in an end-to-end setting by integrating its task-specific SOURCE and SINK lists into FlowDroid and applying the analysis to a large set of real-world Android apps. In this experiment, we simulate a realistic user scenario by considering the following concrete analysis goal:

User Task: Is location-related data being sent outside the device through the network?

Prior to TaskFlow, addressing such a question would typically require either running FlowDroid with a generic specification (e.g., DocFlow or Susi) and manually filtering the results to remove irrelevant flows, or manually constructing a custom SOURCE/SINK list from scratch. Both approaches are time-consuming, error-prone, and require substantial domain expertise. This experiment is designed to precisely simulate a realistic usage scenario and assess whether task-specific specifications generated by TaskFlow can reveal relevant flows that are missed when relying on generic specifications, while focusing solely on a specific task.

We configured FlowDroid in two different ways, differing exclusively in the SOURCE/SINK specifications provided as input, while leaving all other FlowDroid parameters at their default values: ❶ the TaskFlow configuration, created by combining the task-specific SOURCEs generated for Location (see RQ1) with the task-specific SINKs generated for network-related data exfiltration (see RQ5); and ❷ the "baseline" configuration, where we rely on the generic SOURCE/SINK specification provided by SuSi. We focus exclusively on SuSi as a baseline, as it remains the most widely adopted and well-known specification for FlowDroid and often represents the default choice for practitioners. We then constructed a dataset of 10000 recent Android apps randomly sampled from AndroZoo [4], consisting of apps that were added to the repository in 2025 (i.e., apps that were available on Google Play during that year). Both FlowDroid configurations were then executed on this newly created dataset. Figure 4 shows the overlap between the taint flows identified by FlowDroid using the two configurations.

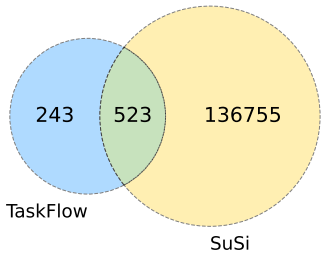

Fig. 4. Overlap between data flows found by FlowDroid using TaskFlow and SuSi.

As shown in Figure 4, the difference between the flows detected by the two configurations is substantial. This is expected, as SuSi relies on a generic specification that is likely to introduce a large number of flows unrelated to the user-defined task. In other words, much of the SuSi-only flows may represent "noise" from the perspective of the specific

analysis goal, i.e., they do not correspond to location-to-network data flows. In contrast, TaskFlow produces a more focused set of flows tailored to the analysis goal. Notably, there exists a non-negligible set of flows that are detected *only* when using the TaskFlow configuration and are missed by SuSi. This suggests that task-specific specifications enable FlowDroid to uncover relevant flows that generic specifications fail to capture. However, to confirm that these differences correspond to the intended user task, further analysis is required.

We therefore conducted a manual inspection of the detected flows. A flow was considered *task-relevant* if it originated from a location-related `SOURCE` and reached a network-related `SINK`. We emphasize that we are not interested in the nature of the data flow itself (i.e., whether it is malicious or benign). For instance, a navigation app may legitimately transmit location data to remote servers for computation. Assessing intent or legitimacy is outside the scope of this paper, which focuses exclusively on identifying data flows using precise, task-specific specifications. We first manually inspected a random sample of 70 flows (95% confidence level, 10% margin of error) from the TaskFlow-only region ($F_{\text{TaskFlow}} \setminus F_{\text{SuSi}}$, i.e., the blue part). We found that 95% of the inspected flows matched the user task, indicating that these are task-relevant flows that are *missed* when relying on the generic SuSi specification. We then repeated the same procedure on a random sample of 96 flows (95% confidence level, 10% margin of error) from the SuSi-only region ($F_{\text{SuSi}} \setminus F_{\text{TaskFlow}}$, i.e., the yellow part). In this case, 0% of the inspected flows matched the intended task, confirming that most SuSi-only flows are indeed "noise" with respect to the user-defined goal.

These results highlight a practical advantage of task-specific specifications. However, it is important to note that this comparison is not perfectly “fair”. As discussed in RQ2 and illustrated by the Sankey diagram (see Section 5.3), performing a fully fair comparison between generic and task-specific specifications is extremely challenging, even when considering the macro-categories of SuSi. Nonetheless, this RQ reflects the typical scenario faced by users, where the only alternative to TaskFlow is a generic taint specification that generates numerous "irrelevant flows", which must then be analyzed manually.

Answer to RQ7. When integrated into FlowDroid and applied to real-world Android apps, TaskFlow enables the detection of task-relevant data flows that are missed by generic specifications such as SuSi, while simultaneously filtering out "irrelevant flows", focusing the analysis on flows that matter for the user-defined task. These results demonstrate that task-specific taint specifications are essential for accurately answering concrete, user-driven analysis questions.

6 Limitations and Threats to Validity

This section outlines key limitations of our approach.

On Ground-Truth Creation and Task Selection. Creating a comprehensive ground-truth dataset for taint source identification in Android apps remains an inherently challenging and largely unsolved problem. Unlike tasks with well-established benchmarks, identifying taint sources requires extensive manual effort, deep domain expertise, and meticulous code inspection, making it both time-consuming and resource-intensive. Building such ground truth through manual labeling would require analyzing the entire set of approximately 45,000 Android API methods, which is practically infeasible. Therefore, in this paper, we used the CoDoC [43] ground-truth dataset, containing 231 labeled `SOURCE` methods, as a starting point for our manual effort. We selected three specific tasks to showcase the flexibility of TaskFlow across diverse and realistic scenarios, spanning different scopes and characteristics, ranging from a broad, cross-cutting objective involving multiple Android subsystems to highly specific tasks involving a very small set of APIs. Since users can define an unbounded number of tasks, each potentially unique, it is impossible to create an

all-encompassing list. Consequently, it is impossible to define a closed or representative set of tasks that would "cover" this space in a principled manner. The relatively small size of ground-truth sets for some tasks reflects the availability of carefully validated data rather than the full breadth of all potential taint sources, as we must build upon pre-existing ground truth. Expanding the ground truth, whether by increasing the number of labeled methods or adding more tasks, would require a massive manual effort. While we experimented with a few additional tasks and observed similarly strong performance, these experiments did not reveal new insights beyond those already discussed. Moreover, the limited number of methods associated with some tasks is consistent with prior work in the literature. For instance, task-specific analyses such as ClipboardScope intentionally focus on a very small number of relevant methods. This directly supports our argument that tasks are not necessarily broad or subsystem-spanning, and further highlights that TaskFlow is explicitly designed to handle both broad tasks (e.g., location inference spanning multiple subsystems) and extremely narrow ones (e.g., clipboard usage). We believe our evaluation represents a balanced and practical approach. Our choice of tasks is intended to be illustrative rather than comprehensive, demonstrating meaningful improvements.

Limitations of the Two-Phase Filtering Approach. As extensively discussed in the Design Choices subsection (see Section 4.3), TaskFlow implements a two-phase pipeline as a deliberate trade-off between scalability and completeness. While Phase 1 conservatively retains methods that could plausibly contribute to security- or privacy-oriented tasks (as this is the main usage from the typical application of taint analysis), the classification inherently involves heuristic-based judgment. Tasks that do not fit the "attacker gains something" paradigm and are therefore outside the scope of the current filtering approach. Additionally, methods filtered out at this stage cannot be recovered in Phase 2, whereas methods erroneously retained can still be refined.

LLM-related problems A core limitation of using LLMs in TaskFlow is the risk of hallucination, i.e., producing inaccurate or nonsensical outputs. We mitigated this by prompt engineering and analyzing each method five times, selecting the majority label to enhance robustness. Still, hallucination remains a risk that can affect the accuracy of task-specific `SOURCE` lists. Future work could explore ensemble prompting or confidence scoring to further reduce its impact. Additionally, a potential LLM-related threat to validity stems from the possibility that the model has seen parts of the data used (e.g., the CoDOc ground truth released in 2021) during pretraining, given that the cutoff date for GPT-4o-mini is October 1, 2023 [34]. Unfortunately, no comparable earlier manually curated datasets of Android `SOURCE`s and `SINK`s exist, and manually labeling a new large-scale ground truth is beyond the scope of this work. Finally, although TaskFlow is designed to work with a wide range of LLMs, our evaluation primarily relied on `gpt-4o-mini`, a relatively lightweight model, to balance cost efficiency with strong performance, as demonstrated in RQ5 (see Section 5.6). More capable models may produce more accurate or more complete task-specific specifications. We note that using different LLMs could lead to variations in results, and therefore represents a potential threat to the reproducibility of our findings. As LLM capabilities continue to advance, we expect future models to further improve the quality of the generated `SOURCE` and `SINK` lists while simultaneously reducing costs. Beyond incremental improvements in model performance, the rapid evolution of LLM paradigms may fundamentally reshape how systems such as TaskFlow are designed and deployed. In particular, recent advances in agentic AI, where LLMs operate as autonomous agents capable of multi-step reasoning and tool interaction, open new possibilities for extending our approach. Rather than relying on a single, stateless pipeline, TaskFlow could be integrated into a broader agentic workflow composed of multiple specialized components. For instance, one agent could focus on identifying task-specific `SOURCE` methods, while another could specialize in `SINK` identification, and additional agents could handle refinement, validation, or interaction with external knowledge sources such as API documentation. Such a modular and collaborative setup could improve adaptability, enable iterative refinement, and better support complex or evolving analysis tasks. While exploring these directions is

beyond the scope of this work, we believe that integrating TaskFlow into agentic pipelines represents a promising avenue for future research.

Methods with callbacks. TaskFlow currently focuses on methods where tainted data is returned directly, limiting its effectiveness with callbacks. Some Android APIs, such as `LocationManager.requestLocationUpdates()`, return data asynchronously via callbacks, often using void methods (i.e., methods that do not return any value). These may be filtered out during preprocessing, leading to incomplete data flow analysis. Incorporating callback-based sources is a promising direction.

Handling of Control-Flow Guards and Implicit Flows. TaskFlow currently focuses on identifying *explicit* SOURCE methods, that is, APIs whose return values directly introduce data into the program's dataflow. As such, the framework does not model *implicit flows*, where sensitive information influences control decisions without being explicitly propagated as data. APIs such as `isMicrophoneMute()` illustrate this distinction: depending on the threat model, such methods may act purely as control-flow guards (e.g., enabling or disabling a behavior), or they may become explicit sources when their return values are intentionally transmitted or logged. Since TaskFlow generates SOURCE sets based on task-specific semantics, whether such APIs are treated as explicit sources depends on how the task is defined. Nevertheless, fully reasoning about implicit flows and distinguishing control-flow guards from true data-returning sources remains outside the scope of this work and represents an interesting avenue for future extensions of the framework.

Methods related to user inputs. Some existing approaches, such as SUPOR [16] and UIPick [30], focus on detecting at runtime whether user-input-related methods (e.g., `editText.getText()`), file reads, or network calls should actually be considered as `SOURCE`s in a given execution context. While TaskFlow can be instructed to generate a list of user-input-related `SOURCE`s (by defining this as a specific task), it does not perform such runtime reasoning itself, as it outputs a list of methods. Nevertheless, TaskFlow can be used as a complementary tool to these approaches by automatically generating task-specific candidate lists that can then be fed into runtime or hybrid analyses to guide their focus.

Task granularity. The quality of task-specific `SOURCE` lists depends heavily on how clearly users articulate their analysis goals. As shown in RQ3 (Section 5.4), task description detail significantly affects list precision and recall. The LLM's interpretation can vary with wording, even for similar goals. Users can mitigate this by iterating on prompts or starting broadly. Future work may explore automatic prompt refinement, for instance, using the LLM to expand vague prompts into more structured definitions.

7 Related Work

LLM for Static Analysis As LLMs are increasingly applied across a wide range of domains with promising outcomes, recent research has begun to explore their integration into static analysis. Regarding taint analysis, LATTE [25] by Liu et al. employs LLMs for static binary analysis to detect vulnerabilities in compiled code, but it focuses on binaries. Another approach, IRIS [24] by Li et al., leverages GPT-4 to infer taint specifications in Java repositories via CodeQL, targeting a broad range of CWE categories. However, neither of these approaches focuses on the Android ecosystem. More broadly, in the context of combining LLM with static analysis, AppPoet [52] by Zhao et al. introduces a multi-view Android malware detection system that combines static feature collection with LLM-generated function descriptions and behavioral summaries. LAMD [38] by Qian et al. builds on this by extracting security-critical contexts from Android apps and constructing structural representations for malware detection via LLMs. While these systems demonstrate the potential of LLMs to enhance static analysis, to the best of our knowledge, TaskFlow is the first approach to focus on generating task-specific taint specifications within the Android ecosystem.

Generating Lists of SOURCEs for Android. Android taint analysis tools often depend on automatically generated lists of SOURCE and sink methods. SUSI [5], the most widely used tool, applies machine learning to classify API methods based on features like names and return types. While easy to integrate and scalable, SUSI often misclassifies irrelevant methods, resulting in false positives [28, 43]. Gibler et al.[14] proposed a permission-based approach, but it can miss sensitive methods that don't require permissions[15]. More recent tools like CoDoC [43] and DocFlow [46] extract SOURCEs using code and documentation. CoDoC uses code and JavaDoc embeddings to improve interpretability, but struggles with unseen or ambiguous methods [43]. DocFlow, based entirely on documentation, covers a wide range—including proprietary SDKs—but lacks task-specific customization and suffers from noise in its training data. We propose shifting from global, static lists to custom SOURCE lists tailored to specific analysis tasks. TaskFlow uses LLMs to incorporate task goals, enabling precise, context-aware SOURCE identification.

8 Conclusion

The paper introduces TaskFlow, a novel LLM-driven framework designed to generate task-specific SOURCE lists for Android taint analysis. Addressing the limitations of traditional methods, which often produce incomplete or overly broad lists due to a one-size-fits-all approach and a universal definition of SOURCE, TaskFlow leverages the reasoning capabilities of LLMs to generate tailored SOURCE lists aligned with specific analysis goals. This approach reduces irrelevant taint flows and improves the precision of taint analysis. Our evaluation shows that TaskFlow is effective in generating task-specific SOURCE lists, achieving high F1 scores across all evaluated tasks. It also benefits from a preprocessing phase that makes the process faster and more efficient. In summary, this work highlights the potential of LLMs to adapt to the evolving landscape of Android APIs and diverse analysis requirements. TaskFlow represents a step toward more intelligent, context-aware taint analysis tailored to specific user goals.

9 Acknowledgement

This research was funded in part by the Luxembourg National Research Fund (FNR), REPROCESS grant reference C21/IS/16344458, and UNLOCK grant reference C23/IS/18154263.

References

[1] M. Abdin et al. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. arXiv:2404.14219 [cs.CL] https://arxiv.org/abs/2404.14219

[2] Marco Alecci, Jordan Samhi, Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2024. Improving Logic Bomb Identification in Android Apps via Context-Aware Anomaly Detection. *IEEE Transactions on Dependable and Secure Computing* 21, 5 (2024), 4735–4753.

[3] Marco Alecci, Jordan Samhi, Marc Miltenberger, Steven Arzt, Tegawendé F Bissyandé, and Jacques Klein. 2025. DamFlow: Preventing a Flood of Irrelevant Data Flows in Android Apps. *ACM Transactions on Software Engineering and Methodology* (2025).

[4] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[5] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2013. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114* (2013).

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices* 49, 6 (2014), 259–269.

[7] Yongliang Chen, Ruoqin Tang, Chaoshun Zuo, Xiaokuan Zhang, Lei Xue, Xiapu Luo, and Qingchuan Zhao. 2024. Attention! your copied data is under monitoring: a systematic study of clipboard usage in android apps. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[8] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*. 239–252.

[9] DeepSeek-AI et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

[10] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[11] Ming Fan, Le Yu, Sen Chen, Hao Zhou, Xiapu Luo, Shuyue Li, Yang Liu, Jun Liu, and Ting Liu. 2020. An empirical evaluation of GDPR compliance violations in Android mHealth apps. In *2020 IEEE 31st international symposium on software reliability engineering (ISSRE)*. IEEE, 253–264.

[12] Pietro Ferrara, Fausto Spoto, et al. 2018. Static analysis for GDPR compliance. In *CEUR Workshop Proceedings*, Vol. 2058. CEUR Workshop Proceedings, 1–10.

[13] Gemma Team. 2025. Gemma 3 Technical Report. arXiv:2503.19786 [cs.CL] https://arxiv.org/abs/2503.19786

[14] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Trust and Trustworthy Computing: 5th International Conference, TRUST 2012, Vienna, Austria, June 13-15, 2012. Proceedings 5*. Springer, 291–307.

[15] Sigmund Albert Gorski, Benjamin Andow, Adwait Nadkarni, Sunil Manandhar, William Enck, Eric Bodden, and Alexandre Bartel. 2019. Acminer: Extraction and analysis of authorization checks in android's middleware. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*. 25–36.

[16] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. {SUPOR}: Precise and scalable sensitive user input detection for android apps. In *24th USENIX Security Symposium (USENIX Security 15)*. 977–992.

[17] Mohsin Junaid, Donggang Liu, and David Kung. 2016. Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models. *computers & security* 59 (2016), 92–117.

[18] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD Center. 2012. ScanDal: Static analyzer for detecting privacy leaks in android applications. *MoST* 12, 110 (2012), 1.

[19] Maria Kober, Jordan Samhi, Steven Arzt, Tegawendé F Bissyandé, and Jacques Klein. 2023. Sensitive and Personal Data: What Exactly Are You Talking About?. In *2023 10th International Conference on Mobile Software Engineering and Systems 2023 (MobileSoft)*. 70–74. https://doi.org/10.1109/MOBILSoft59058.2023.00016

[20] Kevin Li, Lin Ling, Jinqiu Yang, and Lili Wei. 2026. Automatically Detecting Checked-In Secrets in Android Apps: How Far Are We? *Empirical Software Engineering* 31, 2 (2026), 31.

[21] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F Bissyandé, and Jacques Klein. 2015. Potential component leaks in android apps: An investigation into a new feature set for malware detection. In *2015 IEEE international conference on software quality, reliability and security*. IEEE, 195–200.

[22] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *ICT Systems Security and Privacy Protection: 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings 30*. Springer, 513–527.

[23] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95.

[24] Ziyang Li, Saikat Dutta, and Mayur Naik. 2024. Llm-assisted static analysis for detecting security vulnerabilities. *arXiv preprint arXiv:2405.17238* (2024).

[25] Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhenyang Xu, Zhi Li, Peng Di, Yu Jiang, et al. 2025. Llm-powered static binary taint analysis. *ACM Transactions on Software Engineering and Methodology* (2025).

[26] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[27] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 229–240.

[28] Linghui Luo, Eric Bodden, and Johannes Späth. 2019. A qualitative analysis of android taint-analysis results. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 102–114.

[29] Meta AI. 2024. Introducing Llama 3.1. https://ai.meta.com/blog/meta-llama-3-1/. Accessed: 2025-11-13.

[30] Yuhong Nan, Min Yang, Zhemin Yang, Shunfan Zhou, Guofei Gu, and XiaoFeng Wang. 2015. {UIPicker}:{User-Input} privacy identification in mobile applications. In *24th USENIX Security Symposium (USENIX Security 15)*. 993–1008.

[31] Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan Zhang, Donglai Zhu, and Min Yang. 2018. Finding clues for your secrets: semantics-driven, learning-based privacy discovery in mobile apps.. In *NDSS*.

[32] Ollama. 2025. Ollama: Local LLMs Made Easy. https://ollama.com. Accessed: 2025-10-30.

[33] OpenAI. 2024. GPT-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/ Accessed: 2025-04-24.

[34] OpenAI. 2024. GPT-4o mini Model Documentation. https://platform.openai.com/docs/models/gpt-4o-mini. Accessed: 2025-10-24.

[35] OpenAI. 2024. Text Embedding 3 Small. https://platform.openai.com/docs/models/text-embedding-3-small Accessed: 2025-04-24.

[36] OpenAI. 2025. Introducing gpt-oss: Open-Weight Reasoning Models. https://openai.com/index/introducing-gpt-oss/. Accessed: 2025-11-13.

[37] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. 2017. Dark hazard: Large-scale discovery of unknown hidden sensitive operations in android apps. (2017).

[38] Xingzhi Qian, Xinran Zheng, Yiling He, Shuo Yang, and Lorenzo Cavallaro. 2025. LAMD: Context-driven Android Malware Detection and Classification with LLMs. *arXiv preprint arXiv:2502.13055* (2025).

[39] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. 2014. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *2014 Ninth International Conference on Availability, Reliability and Security*. IEEE, 40–49.

[40] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in Android applications that feature anti-analysis techniques.. In *NDSS*.

[41] Jordan Samhi, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2021. A first look at Android applications in Google Play related to COVID-19. *Empirical Software Engineering* 26 (2021), 1–49.

[42] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2022. Jucify: A step towards android code unification for enhanced static analysis. In *Proceedings of the 44th International Conference on Software Engineering*. 1232–1244.

[43] Jordan Samhi, Maria Kober, Abdoul Kader Kabore, Steven Arzt, Tegawendé F Bissyandé, and Jacques Klein. 2023. Negative results of fusing code and documentation for learning to accurately identify sensitive source and sink methods: An application to the android framework for data leak detection. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 783–794.

[44] Jordan Samhi, Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2022. Difuzer: Uncovering suspicious hidden sensitive operations in android apps. In *Proceedings of the 44th International Conference on Software Engineering*. 723–735.

[45] Venkatesh Gauri Shankar, Gaurav Somani, Manoj Singh Gaur, Vijay Laxmi, and Mauro Conti. 2017. AndroTaint: An efficient android malware detection framework using dynamic taint analysis. *2017 ISEA Asia security and privacy (ISEASP)* (2017), 1–13.

[46] Marcos Tileria, Jorge Blasco, and Santanu Kumar Dash. 2024. DocFlow: Extracting Taint Specifications from Software Documentation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[47] Weiping Wang, Jianjian Wei, Shigeng Zhang, and Xi Luo. 2019. LSCDroid: Malware detection based on local sensitive API invocation sequences. *IEEE Transactions on Reliability* 69, 1 (2019), 174–187.

[48] Nattanon Wongwiwatchai, Phannawhat Pongkham, and Kunwadee Sripanidkulchai. 2020. Comprehensive detection of vulnerable personal information leaks in android applications. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 121–126.

[49] A. Yang et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[50] Mu Zhang and Heng Yin. 2014. Appsealer: automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications.. In *NDSS*, Vol. 14. 23–26.

[51] Qingchuan Zhao, Chaoshun Zuo, Brendan Dolan-Gavitt, Giancarlo Pellegrino, and Zhiqiang Lin. 2020. Automatic uncovering of hidden behaviors from input validation in mobile apps. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1106–1120.

[52] Wenxiang Zhao, Juntao Wu, and Zhaoyi Meng. 2025. Apppoet: Large language model based android malware detection via multi-view prompt engineering. *Expert Systems with Applications* 262 (2025), 125546.

[53] Hao Zhou, Wei Zhang, Fengqiong Wei, and Yunfang Chen. 2017. Analysis of Android malware family characteristic based on isomorphism of sensitive API call graph. In *2017 IEEE Second International Conference on Data Science in Cyberspace (DSC)*. IEEE, 319–327.