DamFlow: Preventing a Flood of Irrelevant Data Flows in Android Apps

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

State-of-the-art tools like FlowDroid have been proposed to detect data leaks in Android apps, but two main challenges persist: ① false alarms and ② undetected data leaks. One contributing factor to these challenges is that a tool such as FlowDroid relies on predefined lists of privacy-sensitive source and sink API methods. Generating such lists is complex; incomplete or inaccurate lists result in both false alarms (i.e., irrelevant data flows) and undetected data leaks. Additionally, data leaks are highly context-dependent. For instance, GPS data flowing from a navigation app is expected, but the same flow in a calculator app is suspicious. Even when FlowDroid identifies a source-to-sink path, it may not be relevant to privacy analysis, further increasing false alarms.

To tackle these issues, we propose a novel approach named DamFlow, which, by combining backward taint analysis with context-aware anomaly detection, prevents a "flood" of irrelevant data flows while at the same time finding data leaks missed by existing approaches. Our evaluation demonstrates that DamFlow significantly reduces reported leaks per app while uncovering previously undetected leaks, enhancing FlowDroid's practicality for real-world data leak detection.

CCS Concepts: • **Security and privacy → Domain-specific security and privacy architectures**.

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. 2018. DamFlow: Preventing a Flood of Irrelevant Data Flows in Android Apps. *J. ACM* 37, 4, Article 111 (August 2018), 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Ensuring the security and privacy of mobile applications (apps) has become paramount due to the widespread use of mobile devices and the increasing number of apps employed by users every day. Various techniques have been developed to analyze Android apps to meet this demand. Among these, FlowDroid [1], based on Soot [2], represents one of the most influential state-of-the-art

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.

ACM 1557-735X/2018/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

Android app static analysis tools. It focuses on uncovering data leaks through static data flow analysis and modeling Android-specific challenges like the app lifecycle. FlowDroid employs taint analysis, a specific type of data flow analysis. This technique involves tagging (tainting) variables and monitoring their flow through the code to determine if data from a designated `SOURCE` method (or its derived data) is transmitted to a `SINK` method. A `SOURCE` method is an API function that accesses privacy-sensitive information, while a `SINK` method is an API function that could potentially expose or mishandle this data, such as sending it externally via a network connection. The accurate identification of sources and sinks is essential for properly configuring data flow analysis tools such as FlowDroid and others.

In practice, these sources and sinks are usually identified through automated techniques such as the popular and well-known SUSI approach [3] or the more recent DocFlow framework [4]. However, in real-world scenarios, this approach (FlowDroid with a predetermined list of sources and sinks) is impractical due to two main issues:

(1) **False alarms.** Users are frequently overwhelmed by a "flood" of reported data leaks (sometimes hundreds per app), most of which are irrelevant to the problem at hand. By irrelevant data flows, we mean that while FlowDroid correctly identifies data flowing from a `SOURCE` to a `SINK`, the flow is irrelevant to security and privacy concerns and does not constitute a real data leak. This issue arises when the `SOURCE` is not considered relevant in the context of the analysis.

(2) **Missed Data Leaks.** A predetermined list of sources and sinks may be incomplete, leading to missed data leaks. By missed data leaks, we mean that data flows from a `SOURCE` to a `SINK`, but FlowDroid fails to identify it because the `SOURCE`, the `SINK`, or both are absent from the predetermined list.

In this paper, we make FlowDroid more practical by proposing DamFlow, a novel approach that prevents a "flood" of irrelevant data flows while simultaneously reducing missed data leaks. This is achieved by combining backward taint analysis and context-aware anomaly detection. Next, we describe our core insights regarding both goals.

On the need to perform Backward Analysis: Generating lists of sources and sinks is not trivial [5]. The core challenge lies in distinguishing, a priori, between privacy-sensitive sources and non-privacy-sensitive APIs [6]. Approaches that generate lists of sources and sinks for data leak identification, such as SUSI, often misclassify many non-privacy-sensitive Android API methods as sources, resulting in numerous irrelevant data flows [6–9]. Moreover, since these lists are generated automatically, they may overlook some sources that should instead be considered. For example, the list generated by DocFlow excludes API methods like "`<android.telephony.gsm.GsmCellLocation: int getCid()>`", which retrieves the cell ID (CID) of the device's current GSM location. This information is privacy-sensitive, as it can reveal the device's approximate physical location, posing a potential data leak risk.

Recognizing the challenges in compiling an exhaustive and precise list of privacy-sensitive sources, we shift our focus toward reimagining the analysis framework itself, thereby eliminating the need for such a list. We adopt a fundamentally different strategy that abandons the traditional notion of sources entirely. More specifically, our approach starts a backward taint analysis from a set of pre-defined SINK methods. Hence, our approach requires a list of sinks, but no sources. We argue that, while it is not trivial to distinguish between privacy-sensitive sources and non-privacy-sensitive APIs, sinks do not require a notion of sensitivity. Sinks can indeed be more clearly defined as Android API functions that send out one or more types of data [6]. More precisely, sinks make the data available outside the scope of the app, e.g., for other apps or remote servers.

As an example, a SINK would look something like:

```
<android.telephony.ims.stub.ImsSmsImplBase: void sendSms()>
```

On the contrary, it is difficult to determine a priori if the method

```
<android.net.nsd.NsdServiceInfo: java.lang.String getServiceName()>
```

which is considered a SOURCE by SUSI, is truly a SOURCE.

As we show in our pre-study in Section 2, identifying sinks is indeed easier than identifying sources. Recall that sinks are functions that initiate network connections, send SMS messages, or access external storage, which are well-documented and are unrelated to the high-level notion of "privacy". It is important to note that while our approach is based on the backward taint analysis implementation of FlowDroid [10] and focuses exclusively on Android APIs, this methodology can be generalized to all backward data flow trackers.

By performing the backward taint analysis starting from a well-defined list of sinks, our approach computes all the data flows for which the data ending up in the sinks originates from an Android API method. This technique allows us to reduce the *false negatives* by catching *all* potential sources of data flowing to sinks since we do not rely on pre-determined lists of sources.

On the need to perform Context-Aware Anomaly Detection: We then utilize context-aware anomaly detection, tailored to the app's category, to reduce *false alarms* by distinguishing actual data leaks from irrelevant flows.

Specifically, our approach uses One-Class Support Vector Machine (OC-SVM) [11], trained on the data flow pairs extracted from the app, to perform anomaly detection based on the app's category.

This prevents us from misclassifying normal data flows as leaks by considering the app's expected behavior. Consider, for example, a navigation app sending the device's location to a remote server. In its normal configuration, FlowDroid will correctly report this as a data leak. The flow, however, is irrelevant as transmitting the device's location can be considered normal behavior for a navigation app. Conversely, if a similar data flow is found inside a calculator app, there is a high probability that the app contains a real data leak, as the data flow is abnormal in the context of a calculator app. Note that we consider the app in isolation and do not tackle the question of how the receiving server handles the data.

While our approach is not the first to use context-aware anomaly detection in the context of data flow analysis, existing approaches typically rely on traditional list-based methods. For example, AnFlo [12] detects anomalous data flows in Android apps by grouping trusted apps based on functionality and comparing new apps to learned patterns. Other approaches, such as MudFlow [13], are designed for specific tasks like malware detection. In contrast, to the best of our knowledge, DamFlow is the first approach aimed at detecting data leaks without requiring a pre-defined list of sources, while still being a general solution that enhances FlowDroid's performance for general data leak identification purposes.

Evaluation & Contributions: We evaluated our approach, DamFlow, against FlowDroid when used with a pre-determined list of sources and sinks, i.e., in the "traditional way". We utilized the lists generated by the most popular approach, SUSI [3], as well as the list generated by DocFlow, a novel technique proposed by Tileria et al. [4], which derives taint specifications directly from platform documentation and has been shown to outperform SUSI [4]. Our approach significantly reduced the number of data leaks reported per app while simultaneously identifying data leaks that were missed by existing approaches, making FlowDroid more practical for identifying data leaks in real-world scenarios. We then compared our approach with AnFlo, which also employs a context-aware method for detecting anomalous privacy-sensitive data flows but still relies on a predetermined list of sources and sinks [12], demonstrating how our method outperforms it. Our main contributions can be summarized as follows:

- We propose **DamFlow**, a novel approach that prevents a "flood" of irrelevant data flows while also avoiding missed data leaks that can occur due to an incomplete predetermined list of sources and sinks, relying on a combination of backward taint analysis and context-aware anomaly detection. To the best of our knowledge, DamFlow is the first approach to detect data leaks without needing a well-defined list of sources.
- We evaluated our approach against FlowDroid when using a pre-determined list of sources and sinks and demonstrated that DamFlow drastically reduces the number of returned data flows by around 92% while simultaneously finding more data leaks.
- We compared our approach to AnFlo, another context-aware method aimed at detecting abnormal data flows, demonstrating how our method outperforms it.
- We publicly release DamFlow and all our artifacts at:

https://anonymous.4open.science/r/DamFlow-777

2 Motivations

FlowDroid's effectiveness in real-world data leak detection is limited when using a pre-defined list of sources and sinks. It often produces too many alerts, most of which are irrelevant data flows rather than actual leaks, and it can also miss genuine leaks due to an incomplete list.

In the literature, several automated techniques for identifying sources and sinks have been proposed [3, 4, 6, 14–16], as the vast number of Android API methods available to developers makes manual classification impractical [6]. One of the most popular and well-known approaches is SUSI, a feature-based machine-learning method for identifying sources and sinks in the Android framework. Studies have demonstrated that SUSI often misclassifies non-privacy-sensitive Android API methods as sources, leading to irrelevant data flows [6–9]. Luo et al. found that this irrelevant flow rate could reach 80%, undermining tools like FlowDroid in identifying data leaks [9]. Numerous attempts to refine sources and sinks lists for privacy analysis [3, 4, 6, 14–16], along with recent findings from Samhi et al. [6], underscore the challenge of creating accurate lists.

This complexity has prompted us to reconsider current methods, opting instead for a backward taint analysis approach that starts with a set of well-defined sinks and uses context-aware anomaly detection based on the two motivations explained hereafter.

We hypothesize that sinks are easier to identify than sources. The rationale is that a source is defined as a method providing privacy-sensitive data. The notion of sensitivity is key to this definition of source. However, sensitivity is a subjective notion on which we can disagree: *John finds that this method X is privacy-sensitive, but not Elena.* In contrast, sinks do not involve any notion of sensitivity as they are defined as Android API functions that send out one or more types of data [6], and sending data out can be factually determined.

To validate this hypothesis, we conducted a survey in which participants were asked to label a list of 100 Android API methods as either SOURCE, SINK, or NEITHER. The goal of this analysis is to measure the level of agreement among respondents regarding the classification of Android methods. We created the list of 100 Android API methods as follows: 50 were chosen randomly from all Android API methods, 25 were selected randomly from the list of sinks generated by SUSI, and 25 were chosen randomly from the list of sources generated by SUSI. If we had simply collected 100 random Android API methods, we likely would have ended up with a list predominantly consisting of methods that are neither sources nor sinks. Note that we do not assume the SUSI labels to be fully correct.

The survey was then distributed to 43 grad students, who were enrolled in a Program Analysis course. To quantify the level of agreement among respondents in classifying Android methods into SOURCE, SINK, and NEITHER categories, we computed an **Agreement Score**, calculated as follows:

$$\text{Agreement Score} = \frac{\text{\# Most Assigned Label Counts} - \text{\# Other Labels Counts}}{\text{\# Total Responses}}.$$

This metric effectively captures the extent to which the majority label stands out in comparison to other responses, rather than merely considering the count of the most frequently assigned label. Indeed, a larger gap between these counts indicates a stronger consensus among respondents (e.g., a score of 100% reflects complete agreement), whereas a smaller gap suggests less agreement. Figure 1 shows the average agreement score for each label.

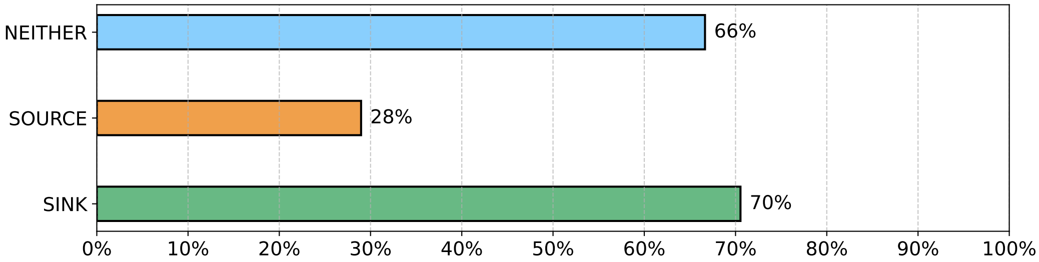

Fig. 1. Average Agreement Score across 100 Android API methods.

As can be observed from the plot, the percentage agreement score on sinks is more than double that of sources, demonstrating that sinks are indeed easier to define than sources. This result is a strong motivation for our approach, which is based on backward taint analysis requesting only a list of sinks and no sources.

The context of the analyzed app should be considered. The second issue with FlowDroid's usability is that it often reports many data flows as data leaks that are actually irrelevant to privacy analysis when the context of the analyzed app is actually taken into account. While evaluating our approach and comparing it against the state of the art, we identified a weather app (com.accurate.weather.forecast.live) as a concrete and suitable example. Running FlowDroid with the list of sources and sinks generated by DocFlow returns a flow originating from the method `<android.location.Location: getLatitude()>`, which retrieves the device's GPS coordinates. However, even though FlowDroid reports this as a data leak, this data flow does not constitute a real data leak, as accessing the device's location can be considered normal behavior for a weather app. Later in this paper, we will show how our approach, DamFlow, is capable of filtering out such irrelevant data flows.

To address this issue, we drew inspiration from CHABADA by Gorla et al. [17], which categorizes Android apps based on their descriptions and employs multiple unsupervised One-Class Support Vector Machine (SVM) anomaly detection models to differentiate between normal and abnormal API usage patterns. Similarly, we decided to categorize apps and train multiple category-based anomaly detection models to distinguish between relevant and irrelevant data flows based on the app's category.

3 Background

In this section, we describe *Backward Taint Analysis* and *Anomaly Detection*, the two main techniques we use in combination in our approach, DamFlow.

Backward Taint Analysis: Taint analysis is an instance of data flow analysis that tracks the movement of particular values throughout a program. A variable V becomes tainted when it is assigned a value from designated functions known as sources. This taint spreads to other variables V' if they obtain a derived value from V. When a tainted variable is passed as an argument to specific functions called sinks, it means that at runtime, we might have a leak. In this paper, we rely on backward taint analysis which operates inversely compared to forward taint analysis. It traces from sink methods backward through the program to identify which variables or inputs could potentially contribute to data reaching these points.

Anomaly Detection: When analyzing data, anomaly detection involves identifying data points that exhibit a substantial difference from the majority of data within the same class. Various techniques, such as One-Class Support Vector Machine (OC-SVM) [11], which we use in our paper, have been proposed for this purpose [18]. The resulting trained model can then be used to predict whether a new sample can be considered an anomaly with respect to the initial group of elements. We rely on multiple category-based OC-SVM models to filter out irrelevant data flows from FlowDroid's output.

4 Approach

Goal: Our approach aims to improve the *usability* of FlowDroid in identifying data leaks in real-world scenarios by ❶ reducing the number of irrelevant data flows typically returned (i.e., reducing *false alarms*) and ❷ avoiding missed data leaks that can occur due to an incomplete predetermined list of sources and sinks (i.e., reducing *false negatives*).

Intuition: As previously introduced, our idea involves starting with a set of well-defined sink methods and retrieving all the Android Framework methods calls reachable through backward taint analysis for a given app, without the need for a list of sources. We then employ category-based anomaly detection to accurately spot data leaks and prevent normal (irrelevant) data flows from being mistaken for data leaks. This prevents the misclassification of normal behavior, like location-related data flows in navigation apps, as data leaks.

App Categorization For our context-aware anomaly detection, we utilize apps belonging to the same "category," defined as a group of apps that share a common purpose and similar functionalities, following the motivation introduced in Section 1 and Section 2. One might initially consider using the Google Play Store's app categories; however, these are often too broad and inconsistent [17, 19–22] to be reliable for fine-grained anomaly detection. To overcome this limitation, we rely on description-based categorization, which allows for a more precise grouping of apps. In particular, we adopt the G-CatA approach [19], which categorizes apps based on semantic features extracted from their descriptions. G-CatA has been shown to achieve a high Adjusted Rand Index (ARI) score of 0.91—indicating near-perfect categorization performance and outperforming existing description-based categorization approaches [19].

Nonetheless, since even a small categorization error can introduce bias and affect downstream results, we first evaluate DamFlow in a controlled experimental setting (RQ1–RQ4). Specifically, we use AndroCatSet, a manually labeled ground-truth dataset [19] consisting of 5000 benign Android apps across 50 distinct categories (e.g., calculators, navigation, weather). This clean dataset allows us to isolate and measure the effectiveness of our backward taint analysis and category-based anomaly detection without the confounding effects of categorization noise. Since the apps are benign, we assume they reflect typical behavior within each category—for example, location-related data flows are expected in navigation apps. However, we acknowledge that this setting does not fully represent real-world deployment, where the category of a new app is typically unknown. To address this, we also evaluate DamFlow in a realistic scenario (RQ5) using a second dataset of real-world apps without predefined categories. In this setting, we apply the previously mentioned

G-CatA approach [19] to automatically assign each app to a category prior to anomaly detection. This allows us to demonstrate DamFlow's applicability in practical, real-world conditions.

Overview: In Figure 2, we provide an overview of our approach. The lower part illustrates the Application Phase, which includes all the steps executed whenever a new app is given as input to be analyzed. In contrast, the upper section of Figure 2 represents the Training Phase, which is performed only once to train multiple category-based anomaly detection models.

In our controlled experiments, we rely on a pre-categorized ground-truth dataset, so there is no need to categorize apps during either the training or application phases—each app's category is already known. However, in a real-world setting, the DamFlow pipeline would require an additional preliminary step to categorize incoming apps before analysis. This categorization can be performed using automated tools such as G-CatA [19], which infer app categories based on the app description.

Both phases can be divided into three key subphases: Backward Taint Analysis, Embedding, and Anomaly Detection. In the following subsections, we will explain them in more detail.

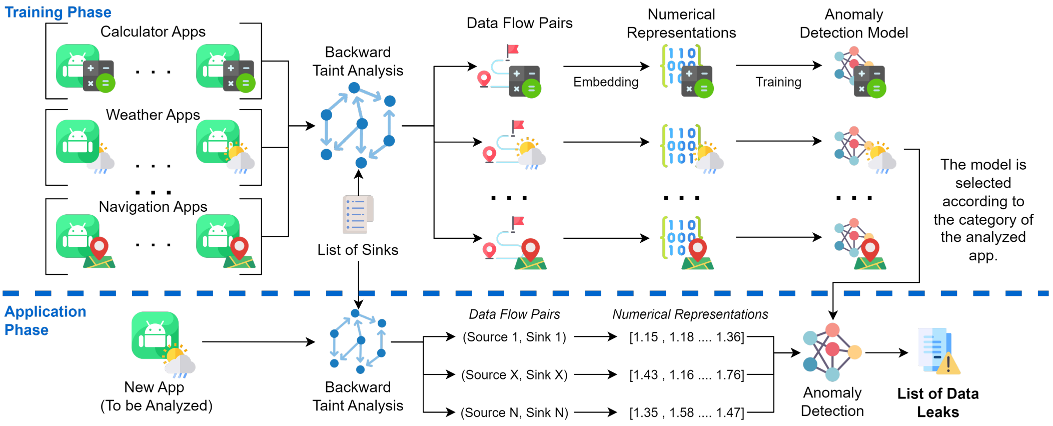

Fig. 2. DamFlow Approach Overview.

4.1 Backward Taint Analysis

Our first step consists of performing backward taint analysis starting with a predefined list of sinks. To achieve this, we developed a custom Java program using FlowDroid's backward data flow implementation [10].

The need for backward analysis was extensively discussed in Sections 1 and 2. We fundamentally reconsider the existing approaches and rely on backward taint analysis, starting only from a list of well-defined sinks. This approach avoids the difficult task of identifying a predefined list of sources and instead focuses on the clearer definition of sinks.

The sink list used in our analysis was originally developed by Samhi et al. [6] via a rigorous manual labeling process involving two expert annotators, who achieved full agreement on 130 Android API sink methods such as `sendTextMessage()` in `android.telephony.SmsManager` and similar.

Note that when performing the backward analysis, FlowDroid computes the data flow pairs for which the data ending up in the sink originates from an Android API method. In other words, in an obtained data flow pair (SOURCE, SINK), both SOURCE and SINK are Android API methods.

FlowDroid Configuration: First, we configured the taint analysis to run in the backward direction instead of the default forward direction. FlowDroid provides various options for dealing with calls to library classes. Since analyzing the Android framework implementation together with each app entails severe scalability issues, we rely on StubDroid library models for the API instead [23].

We further apply an optimization to FlowDroid's path reconstruction. By default, FlowDroid builds a taint graph on the interprocedural control flow graph during its IFDS-based [24] taint analysis. In a subsequent step, FlowDroid reconstructs paths through the taint graph to return paths as sequences of statements between source and sink [25]. We found this additional step to be prohibitively expensive in computation time for some apps in our data set. To alleviate this problem, we configured FlowDroid with a simpler post-processing of the taint graph that only returns the links between sources and sinks (i.e., a data flow pair), without the statements along the path.

We set a hard timeout of 12 hours for analyzing each app, resulting in 99% of AndroCatSet being analyzed within this time limitation. Note that 94% of the apps were analyzed in less than 2 hours. The 60 apps not analyzed are excluded from the study.

Output: The outcome of this initial phase is a list of data flow pairs for each app. Each data flow pair consists of a `SOURCE` method and a `SINK` method. Recall that sources and sinks are calls to API methods in our model. The collection of data flow pairs for an app can be represented as:

$$\text{DataFlowPairs(app)} = [(\text{source}_1, \text{sink}_1), (\text{source}_2, \text{sink}_2), \ldots]$$

Each pair is stored as a JSON object. Here is an example of a real data flow pair :

- **`SOURCE`**: `<android.location.Location: double getLatitude()>`
- **`SINK`**: `<android.telephony.SmsManager: void sendTextMessage()>`

The pairs are next used to train anomaly detection models.

4.2 Embedding

Before the data flows extracted through backward taint analysis can be used to train anomaly detection models, they first need to be converted into numerical representations. FlowDroid relies on Soot and its Jimple intermediate representation [26], and hence a specific format for method signatures. The APIs that appear as sources and sinks for a data flow are such method signatures in our case.

We embedded the `SOURCE` method signature and the `SINK` method signature, then concatenated their numerical representations into a single array, resulting in an array with a length double that of the individual embeddings generated by the model.

$$\text{EMB(Pair)} = [\text{EMB}_{\text{source}} + \text{EMB}_{\text{sink}}]$$

We chose not to use traditional code embedding models such as Code2Vec [27], as there are none specifically trained for Jimple. Instead, we opted for transformer-based models like CodeBERT [28], which better capture contextual information and handle intermediate representations like Jimple more effectively. To avoid bias from relying on a single model, we tested three different embedding techniques, detailed below:

(1) First, we decided to test CodeBERT [28], a more "traditional" approach specifically trained for code. CodeBERT learns representations directly from code tokens using Transformer-based models trained on large-scale code repositories. This model generates 768-dimensional embeddings.

(2) Then, we chose to use one of the most popular transformer-based embedding models, `text-embedding-3-small` by OpenAI [29], the creators of ChatGPT. This model can be accessed through the official OpenAI API. Despite being primarily designed for text embedding, it can also be used to embed code, as suggested by the official website. This model generates 1536-dimensional embeddings [30].

(3) Finally, since our second model is a paid text-embedding model, we decided to explore other free text-embedding models. We referred to the Massive Text Embedding Benchmark (MTEB) Leaderboard on Hugging Face [31]. At the time we started our experiments, the best-performing model on the MTEB Leaderboard was `SFR-Embedding-Mistral` [32], which is based on `Mistral-7B-v0.1` [33]. We selected this as our third model. This model generates 4096-dimensional embeddings.

The outcome of this phase is a list of data flow pairs embedded in their numerical representations for each app.

$$\text{DataFlowPairs}(\text{app}) = \left[\text{EMB}_{\text{pair1}}, \text{EMB}_{\text{pair2}}, \ldots\right]$$

4.3 Anomaly Detection

The final step of the Training Phase involves training multiple category-specific anomaly detection models using the embeddings generated in the previous phase.

Specifically, we train one model per app category (either the category from AndroCatSet or the cluster identified by G-CatA [19])

with the goal of learning to distinguish between normal and abnormal data flows within that category. We adopt the One-Class Support Vector Machine (OC-SVM)[11] for this task, as it has demonstrated strong performance in prior work on category-based anomaly detection[17, 34, 35]. In the case of our controlled experimental setting (RQ1–RQ4), from the 5000 apps in AndroCatSet (100 per category), we use 90% (i.e., 90 apps) for training each OC-SVM model[1]. These trained models—one for each of the 50 categories—are saved locally using `joblib` [36] and reused during the analysis of new apps.

During the Application Phase, a new app is first subjected to backward taint analysis and embedding. Next, the appropriate OC-SVM model is selected based on the app's category

(either directly provided in the case of AndroCatSet or automatically inferred using G-CatA [19]). This model then processes the embedded data flow pairs, filters out those considered category-normal, and returns only those deemed anomalous—i.e., likely data leaks.

5 Evaluation

This section evaluates our approach in two distinct settings: ❶ **Controlled Setting:** using a manually labeled ground-truth dataset to isolate and assess the effectiveness of DamFlow without the influence of categorization errors and ❷ **Real-World Setting:** applying DamFlow to real-world apps without predefined categories, where categorization is performed automatically using G-CatA [19]. We define a set of research questions (RQs) for each setting, as detailed below.

❶ **Controlled Setting.** Since we are, to the best of our knowledge, the first to attempt identifying data leaks solely from sinks, our initial goal is to examine the effectiveness of relying on backward taint analysis using a well-defined set of sinks. This involves investigating the intermediate results of DamFlow to reinforce the motivation previously discussed in Section 2 by addressing the following questions:

- **RQ1:** For a given app, how many methods of the Android Framework are reachable from sink methods when performing backward taint analysis?
- **RQ2:** How do these reachable methods differ across various categories of apps?

Secondly, we aim to evaluate our approach, DamFlow, as a standalone tool capable of detecting data leaks in Android apps. Specifically, we seek to answer the following questions:

[1] Due to timeouts during the taint analysis phase (cf. Section 4.1), some categories may include slightly fewer apps, with a minimum of 87 per category. The evaluation set always includes 10 apps per category.

- **RQ3:** How does DamFlow perform compared to FlowDroid with a pre-determined list of sources and sinks?
- **RQ4:** How does DamFlow perform compared to other category-based approaches, such as AnFlo, for detecting abnormal data flows?

❷ **Real-World Setting.** While the controlled setting offers clean comparisons, real-world scenarios require app categorization at runtime. To simulate this, we consider a second dataset of Android apps that do not have predefined category labels. For this setting, we integrate the G-CatA categorization approach [19] into the DamFlow pipeline as a preprocessing step.

- **RQ5:** How does DamFlow perform when applied to real-world Android apps whose categories are inferred using an automated categorization method (G-CatA)?

5.1 RQ1: Android API methods reachable from sinks.

As described in Section 4.1, the first step of our training phase involves performing backward taint analysis using only a set of well-defined sink methods, thereby completely eliminating the need for a list of sources.

After running the first step on the analyzed apps present in AndroCatSet, i.e., 4940 apps (5000-60), we examined the intermediate results of DamFlow, specifically the data flow pairs returned at the end of the backward taint analysis performed by FlowDroid. Upon analyzing these pairs, we found that, on average, only 233 distinct method calls of Android Framework methods are reachable backward from the sinks in a given app, out of 14 520 distinct Android API method calls present on average in each app. This means that, on average, less than 2% of the Android API method calls present in an app are actually reachable from the well-defined sink methods.

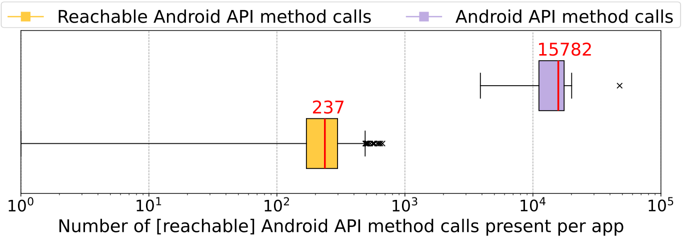

Fig. 3. Distribution of Android API method calls (in violet) vs. reachable Android API Method calls (in yellow) per app.

Figure 3 illustrates the distribution of the number of Android API method calls in an app (top boxplot, in violet). We compare this data to the distribution of the number of Android API method calls reachable from the sinks per app (bottom boxplot, in yellow), using a logarithmic scale on the X-axis. The median values are quite similar to the average values: 237 for the number of reachable Android API method calls and 15 782 for the number of Android API methods. This indicates that there are not many outliers significantly skewing the average values. Additionally, we note that there is a reduction of two orders of magnitude (and in some cases, even three) when considering the reachable method calls. This highlights that the search space for data leaks is considerably smaller than all the Android Framework method calls in a given app. This explains why our anomaly detection is conducted specifically as a second step on the results of the backward taint analysis, rather than being applied directly to all Android API method calls used in a given app.

Answer to RQ1: *We found that less than 2% of the Android API method calls present in apps are actually reachable backward from the well-defined sink methods.*

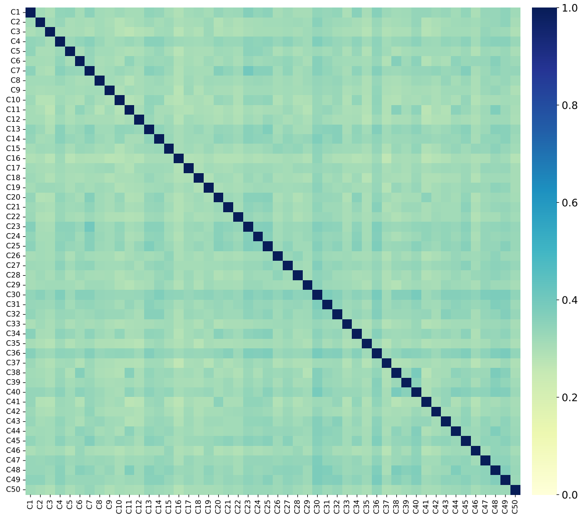

Fig. 4. Pairwise overlap of reachable Android API method calls across the 50 distinct categories of AndroCatSet.

5.2 RQ2: reachable Android API methods across app categories.

While the first research question analyzed how many Android Framework method calls are reachable from the sinks across the entire dataset, this RQ examines how they differ among different categories of apps in AndroCatSet.

Figure 4 presents a heatmap visualization that illustrates the pairwise overlap of Android API method calls reached through backward taint analysis across 50 distinct categories of apps. Each cell in the heatmap represents the degree of overlap (expressed through the pairwise Jaccard Index) between two categories, with darker cells indicating a greater degree of overlap. For instance, let us consider a category X and another category Y, and let M_X and M_Y be the sets of Android API method calls reached through backward taint analysis for both categories X and Y. The lower $|M_X \cap M_Y|$ is, the lower the Jaccard Index and the brighter the color in Fig. 4.

As we can see across the heatmap, the Jaccard Index values are generally low (lighter), implying minimal overlap across different categories. This observation highlights that, although the same list of sinks is used for backward taint analysis, the methods generating the data reaching these sinks vary by category, thereby reinforcing the rationale for a category-based approach.

The differences among the various categories of apps are also illustrated in Figure 5, which shows the distribution of the number of data flows found in each app by category. Each boxplot in the figure represents the distribution for a single category. We can see that the number of data flows varies between categories, with some categories, such as Messenger, containing significantly more data flows compared to categories like Calculators. This highlights how certain types of apps are more prone to having Android API method calls reachable from sinks.

Answer to RQ2: *The Android API methods, which are reachable from sinks, have been shown to differ among app categories, reinforcing the rationale for using a category-based anomaly detection*

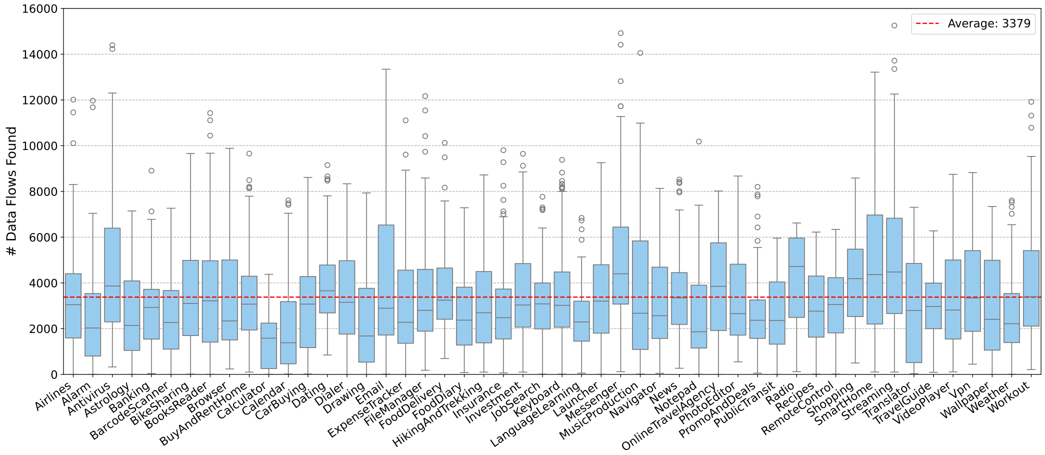

Fig. 5. Distribution of the number of data flows per category.

5.3 RQ3: Comparison with FlowDroid.

In this research question, we compare DamFlow with FlowDroid's output when used in its normal settings i.e., with a predefined list of sources and sinks, referred to as our "baseline". The goal is to determine whether our approach can reduce the number of irrelevant data flows and to assess if DamFlow can identify leaked data missed by FlowDroid due to limitations in the underlying analysis. We first present the raw results of our experiment, followed by a qualitative analysis through manual inspection, as no ground truth data exists for this task.

To ensure a fair and controlled comparison, we decided to compare DamFlow against FlowDroid using the same set of sinks as DamFlow, but with a traditional pre-determined list of sources. For sources, we use two automatically generated lists: one produced by SUSI, the well-known approach proposed by Arzt et al.[3], and the other by DocFlow, a recent technique developed by Tileria et al.[4] that extracts taint specifications directly from platform documentation. The evaluation in [4] demonstrated that DocFlow outperforms SUSI, enabling a more comprehensive security and privacy analysis of Android apps.

As a dataset, we use the remaining part of AndroCatSet, i.e., 500 apps (10 for each of the 50 categories), which we did not use for training the models and reserved specifically for evaluating our approach. We tested DamFlow using all three embedding models introduced in Section 4.2. We refer to them as CB for CodeBERT, GPT for `text-embedding-3-small`, and SFR for `SFR-Embedding-Mistral`. Table 1 displays the total number of data flow pairs returned by each approach, along with the average and median values across the 500 analyzed apps.

	# Data Flow Pairs		
Approach	**Total**	**Average per app**	**Median**
FlowDroid [DocFlow]	36 196	104	86
FlowDroid [SUSI]	16 043	46	35
DamFlow [CB]	2732	8	6
DamFlow [GPT]	4632	13	10
DamFlow [SFR]	53 623	140	130

Table 1. Number of data flow pairs returned per approach.

With the exception of SFR, which still returns a significant number of data flow pairs, DamFlow greatly reduced the number of data flows reported to the user, especially when considering FlowDroid used with the DocFlow list. Indeed, when compared to FlowDroid[DocFlow], the average number of data flow pairs returned per app shows a significant reduction: 92.31% for CB and 87.50% for GPT. Even when considering the median number of data flow pairs returned, the value is relatively high for the baseline (86), indicating that it is not just a few apps returning many data flow pairs (which increases the average), but rather a general trend across all analyzed apps. When compared with FlowDroid[SUSI], the reduction is 82.61% for CB and 71.74% for GPT.

This reduction is in line with our expectations, as a user relying on FlowDroid for data leak identification, such as a security analyst, would no longer need to deal with the overwhelming number of 321 data flow pairs per app (a ‘flood’ of 36 196 data flow pairs in total from our dataset) such as in the case of DocFlow list. However, it remains crucial to assess whether the returned data flow pairs are irrelevant or represent actual data leaks, i.e., whether they are significant in the context of security analysis. On the other hand, computing the recall of our approach—specifically, determining if it fails to identify data flow pairs found by FlowDroid[SUSI] and FlowDroid[DocFlow]—is challenging due to the extremely high number of data flow pairs these tools return, as we will better explain in Section 6.

Manual Inspection. To assess the presence of data leaks, we manually inspect the data flow pairs returned by both the baseline approach and DamFlow. Given that manually inspecting all returned data flows across numerous apps would be excessively time-consuming, we addressed this by randomly selecting a statistically significant sample for each approach. This sampling was conducted with a confidence level of 95% and a confidence interval of ±10%. The first column of Table 2 shows the exact number of data flow pairs inspected for each approach. The samples inspected are distinct subsets of data flow pairs for each approach.

For each inspected data flow pair, we assigned one of the following three labels based solely on the nature of the **source** since all sinks are predefined high-quality sinks, as explained at the beginning of this Section. The meaning of each label is explained below:

- **EXPLICITLY RELEVANT:** This label is assigned when the source explicitly performs an action directly associated with privacy-sensitive data that could lead to a data leak. An example of this is the source method: `android.location.Location: double getLatitude()`. This method directly extracts privacy-sensitive location information, which may result in a data leak.
- **POTENTIALLY RELEVANT:** This label is assigned when the source performs an action that may or may not lead to a data leak depending on context (e.g., when reading something). For instance, the method `java.io.InputStream: int read()` falls into this category. The potential for data leakage depends on the data being read from the `InputStream`. Although the source is linked to a well-defined sink, the actual risk of a data leak varies depending on the content read and how it is subsequently used or exposed.
- **IRRELEVANT:** This label is assigned when the source either (i) does not provide any meaningful information (e.g., a UI method such as `<Canvas: int getWidth()>`, which we found in the DocFlow source list), or (ii) provides data that is aligned with the expected behavior of the app and does not represent a privacy risk. In both cases, the source is not associated with privacy-sensitive information that could result in a data leak.

Table 2 reports the results of our manual inspection.

As shown in Table 2, FlowDroid [DocFlow] did not identify any explicitly relevant data flows, while FlowDroid [SUSI] identified only two. During our manual inspection of DocFlow’s results, we found that many sources offer no meaningful information —for instance, `<android.content.Resources: int getColor(int)>` or `<android.view.Display: int getHeight()>`. In contrast, our approach,

	# Data Flow Pairs			
Approach	**Inspected**	**Irrelevant**	**P. relevant**	**E. relevant**
FlowDroid [DocFlow]	96	88 [92 %]	8 [8 %]	0 [0 %]
FlowDroid [SUSI]	96	88 [92 %]	6 [6 %]	2 [2 %]
DamFlow [CB]	93	83 [89 %]	6 [7 %]	4 [4 %]
DamFlow [GPT]	95	81 [85 %]	3 [3 %]	11 [12 %]
DamFlow [SFR]	96	92 [96 %]	3 [3 %]	1 [1 %]

Table 2. Results of our manual inspection of data flow pairs.

DamFlow, achieved better results. Specifically, manual inspection revealed that 12% of the data flows reported by DamFlow[GPT] were explicitly relevant, compared to 4% for DamFlow[CB]. This demonstrates a reduction in the number of irrelevant data flows compared to both baselines.

While the absolute reduction in irrelevant flows shown in Table 2 may seem modest, it is important not to interpret these results in isolation. Instead, they should be evaluated alongside the data presented in Table 1. As we have consistently highlighted throughout this paper, our primary goal is to reduce the excessive number of false alarms, thereby enhancing FlowDroid's practicality for identifying data leaks. As a result, the output from DamFlow is more actionable than that of FlowDroid when used in the "traditional" manner with a pre-determined list of sources and sinks, such as SUSI and DocFlow. For instance, a user relying on FlowDroid for data leak identification, such as a security analyst, would no longer need to handle the overwhelming number of 104 data flow pairs per app (a 'flood' of 36 196 data flow pairs in total from our dataset), as is the case with FlowDroid when using the list generated by DocFlow. Instead, by using DamFlow, they would get a significantly smaller number of "high-quality" data flow pairs per app, typically no more than 20 (excluding SFR, which has shown disappointing performance). This reduction, combined with the percentages of explicitly relevant data flow pairs we found with manual inspection, shows how DamFlow greatly enhances the practical aspect of FlowDroid in real-world scenarios addressing the issue of excessive false alarms.

Once again, we note that among the three embedding models we are using, SFR appears to be the least effective, as it does not reduce the number of irrelevant data flows compared to the baseline. One possible reason could be that SFR is primarily developed for text embedding tasks, whereas GPT embeddings have been shown to be effective for code embedding tasks as well [30].

Sources missing from SUSI and DocFlow lists. So far, we have demonstrated how our approach can report fewer data flow pairs to users while simultaneously identifying more explicitly relevant data flow pairs, thereby helping to address the false alarms problem. However, as discussed in Section 1, a pre-determined list of sources and sinks may be incomplete, leading to missed data leaks (i.e., the false negative problem). For this reason, we want to investigate whether DamFlow can also resolve this issue by identifying data leaks that are actually missed when using a pre-determined list, such as those generated by SUSI and DocFlow.

Based on our manual analysis results, we examined all explicitly relevant data flow pairs confirmed across the three embeddings used in DamFlow [CB], [GPT], and [SFR]. Out of 16 total pairs, we checked if the sources of these data flow pairs were included in the lists generated by SUSI and DocFlow. Our findings are as follows:

- **We identified 2 sources not present in the DocFlow list:**
  - `<android.telephony.gsm.GsmCellLocation: int getCid()>`
  - `<android.hardware.usb.UsbDevice: int getDeviceProtocol()>`
- **We identified 1 source missing from both SUSI and DocFlow lists:**
  - `<java.util.Locale: java.lang.String getCountry()>`

These findings suggest that DamFlow, which operates by beginning solely with a list of sinks, can identify new sources not included in the SUSI and DocFlow lists. This capability supports our approach of relying exclusively on sinks and working backward, effectively addressing the limitations of list-based methods like DocFlow and SUSI, which may produce incomplete lists leading to false negatives.

A practical example: android.Location package. Similar to what the authors did in the DocFlow paper [4], we focus on the Android.Location package as a case study and examine the results. This package provides a popular API with many privacy-sensitive methods. Specifically, in our case, we aimed to observe how using a category-based anomaly detection method has impacted the final results. We started by considering all the data flow pairs returned by the Baseline approach and all the data flows returned by DamFlow (using GPT, as it was the most effective embedding model). We retained only the data flow pairs where the source belongs to the Android.Location package. Next, we analyzed the categories of the apps where location-related data leaks were found. Table 3 displays the top 3 categories of apps, ordered by the number of location-related data flows.

	Category	
Ranking	**FlowDroid[DocFlow]**	**DamFlow [GPT]**
#1	HikingAndTrekking	Antivirus
#2	PublicTransit	BuyAndRentHome
#3	Weather	SmartHome

Table 3. Top 3 categories of apps ordered by number of location-related data flows.

From Table 3, we can observe that when using the baseline approach, location-based data flows are primarily found in apps belonging to categories clearly related to location-based functionalities, such as HikingAndTrekking, PublicTransit, and Weather. In contrast, DamFlow identified location-related data flows in apps from categories not typically associated with location data, such as Antivirus, BuyAndRentHome, and SmartHome. This validates our decision to train multiple anomaly detection models tailored to different categories. As our goal is to filter out normal data flows based on the app's category, encountering location-related data flows in unexpected categories is a positive indication that our method is working effectively.

To further assess the correctness of the filtering performed by DamFlow, we conducted a small-scale manual analysis of data flows removed by anomaly detection in the three top categories of FlowDroid[DocFlow] in terms of location-related flows: HikingAndTrekking, PublicTransit, and Weather, to check that we do not remove relevant flows (i.e., perform worse than FlowDroid[DocFlow]). For each category, we computed the set difference between the flows before anomaly detection and those remaining after anomaly detection. We then randomly sampled the removed flows using a 95% confidence level and ±10% margin of error. The sampled flows were classified using the same definitions introduced earlier in the section: EXPLICITLY_RELEVANT, POTENTIALLY_RELEVANT, and IRRELEVANT. The results are summarized in Table 4.

	# Data Flow Pairs						
Category	Before A.D	After A.D	Removed by A.D	Inspected	Irrelevant	P. relevant	E. relevant
HikingAndTrekking	889	110	779	86	85 [99%]	1 [1%]	0 [0%]
PublicTransit	1117	75	1042	89	88 [99%]	1 [1%]	0 [0%]
Weather	5505	176	5392	95	95 [100%]	0 [0%]	0 [0%]

Table 4. Results of our manual inspection on a sample of data flow pairs removed by DamFlow through anomaly detection. (A.D. = Anomaly Detection)

It can be observed that nearly all flows removed by DamFlow were irrelevant, with 99–100% of inspected flows classified as such. No explicitly relevant flows were removed, and only a negligible number of potentially relevant flows were filtered out (one flow each for HikingAndTrekking and PublicTransit). Both flows are actually related to the `read()` method, which could potentially be more common in certain categories, causing the anomaly detector to treat them as normal, whereas DamFlow focuses on detecting explicitly relevant flows. Further discussion is provided in Section 6. These results suggest that DamFlow's filtering step effectively removes irrelevant flows while retaining relevant ones, providing additional evidence of correctness. This supports the validity of our anomaly detection-based approach and reinforces the practical usefulness of the filtered outputs.

Runtime Performance Comparison. To evaluate whether DamFlow introduces significant overhead compared to the baseline, we measured the analysis time of DamFlow against FlowDroid configured with the SUSI source list, which is the most commonly used configuration in prior research. Indeed, SUSI has been FlowDroid's primary source list for nearly a decade and is widely used in both academic studies and practical applications [6], reflecting the scenario that a typical user would encounter. Figure 6 shows the distribution of elapsed times for both approaches.

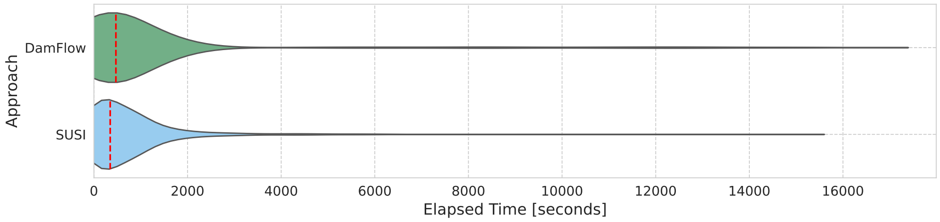

Fig. 6. Distribution of elapsed analysis time (in seconds) for DamFlow and FlowDroid[SUSI]. The red dashed line indicates the median time for each tool.

Both tools exhibit a wide range of analysis times with some apps requiring substantially longer processing, resulting in a long-tailed distribution. DamFlow shows a slightly wider concentration of runtimes between 1000 and 2000 seconds. However, the median elapsed times are comparable: 348.05 seconds for FlowDroid[SUSI] and 470.05 seconds for DamFlow. This modest increase in runtime is likely due to DamFlow performing backward taint analysis from sinks, rather than relying on a predefined list of sources as FlowDroid[SUSI] does. Nevertheless, the small overhead introduced by DamFlow can be considered acceptable given its significant benefits in reducing irrelevant data flows.

LLMs as anomaly detectors. To compare with our OC-SVM-based approach, we tasked a large language model (Llama3.1-405B) with directly identifying abnormal data flows based solely on its understanding of the app category. We provided it with the data flow pairs and the app's category. Note that the LLM was used as-is and has not been specifically trained on data flows. We found that the LLM flagged many flows as abnormal, even though they are actually expected for their respective categories. For example, it identified the sharing of device location in navigation or weather apps as abnormal. A thorough analysis of an LLM-based variant of DamFlow is left to future work.

Answer to RQ3: *DamFlow significantly reduces the number of data flow pairs returned per app by up to 92%, effectively filtering out irrelevant data flow pairs (false alarms problem) while simultaneously identifying data flow pairs that FlowDroid misses when using a predetermined list of sources and sinks (false negative problem).*

5.4 RQ4: Comparing DamFlow against AnFlo.

In RQ3, we compared DamFlow with FlowDroid using a traditional setup, which involves a pre-generated list of sources and sinks. However, since our approach is context-aware—meaning it considers the category of the analyzed app—our goal in this research question is to compare our tool with AnFlo, which also employs a context-aware approach [12]. In their paper, Demissie et al. present a method for detecting anomalous privacy-sensitive data flows in Android apps by grouping trusted apps based on functional categories and learning their typical data flows. Anomalies are flagged when an app's data flow deviates from those of trusted apps within the same category, enhancing detection accuracy [12]. However, their approach differs slightly from ours in that they do not report a list of data flows to the users, such as pairs of SOURCE and SINK methods. Instead, Android APIs (both sources and sinks) are grouped according to the special permissions required for execution. For example, all network-related sink functions—such as openConnection(), connect(), and getContent()—are modeled as Internet sinks because they all require the INTERNET permission to run. They achieved this classification using the privileged APIs available from PScout [37]. As a result, their approach returns a list of outliers based on permission pairs. For instance, according to their results, GPS data sent through the Internet (GPS ⟶ Internet) is common for travel apps, but GPS data sent via Bluetooth (GPS ⟶ Bluetooth) is anomalous and flagged. Further details on their approach can be found in their paper [12].

To compare DamFlow with their approach, we slightly modified their code so that AnFlo would return not only a single permission pair (e.g., GPS ⟶ Internet) but a full list of anomalous data flow pairs, similar to DamFlow. We then ran this modified version of AnFlo on the same set of 500 apps used for RQ3. The output of AnFlo is reported in Table 5.

#Occurences	Source	Sink
3	<android.app.Activity: android.content.Intent getIntent()>	<android.app.Activity: void startActivity()>
1	<android.net.ConnectivityManager: boolean isActiveNetworkMetered()>	<android.content.Context: android.content.Intent registerReceiver()>
1	<android.app.DownloadManager: long enqueue()>	<android.content.Context: android.content.Intent registerReceiver()>
1	<android.media.RingtoneManager: android.media.Ringtone getRingtone()>	<android.media.Ringtone: void setStreamType()>

Table 5. Output of AnFlo when used on our test set of 500 Android apps.

After analyzing 500 Android apps, AnFlo returned a list of 6 anomalous data flows across 6 different apps, which is significantly lower than the results of both our approach and FlowDroid. However, this number is somewhat consistent with their original paper, where they found only 25 abnormal data flows in approximately 600 apps. If we examine the nature of these reported data flow pairs, we can classify them as IRRELEVANT according to the definition previously introduced in RQ3. We searched for these 6 data flow pairs in the results of DamFlow[GPT], which represents our best approach, and did not find them. This indicates that our approach successfully filtered out these irrelevant data flow pairs. On the other hand, AnFlo was unable to detect the 10 Explicitly Relevant and 3 Potentially Relevant data flow pairs that DamFlow[GPT] identified, and we manually confirmed in RQ3, showing that AnFlo missed data flows that DamFlow successfully identified.

> **Answer to RQ4:** *DamFlow outperformed AnFlo, another category-based approach, by identifying explicitly relevant data flow pairs that AnFlo missed, while correctly filtering out irrelevant data flow pairs that AnFlo included.*

5.5 RQ5: DamFlow in a real-world scenario.

While the previous research questions (RQ1–RQ4) evaluate DamFlow in a controlled setting using a manually categorized dataset, real-world applications typically lack such predefined labels. Therefore, the goal of RQ5 is to assess the practical applicability of DamFlow when app categories are unknown and must be inferred automatically. To this end, we leverage the G-CatA approach [19], a description-based categorization method shown to achieve high accuracy, to categorize apps prior to analysis. This research question investigates how well DamFlow performs in this realistic scenario, bridging the gap between controlled experiments and real-world deployment with a newly built dataset.

Since 2020, AndroZoo has also been collecting and sharing app metadata such as app descriptions [38] which we need for the fine-grained automatic categorization. We therefore sampled apps from AndroZoo [39] from the last five years—i.e. since the collection of descriptions began. To mirror the setting of RQ3, we randomly collected 4500 apps with a VirusTotal [40] score of zero to ensure they are benign and reflect expected behavior. We then randomly selected an additional 500 apps as a test set, without placing any constraints on their VirusTotal score, to better simulate a real-world scenario where suspicious apps may be encountered. We applied G-CatA to categorize the apps in the training set and trained the respective OC-SVM models accordingly, identical to the setup used in the controlled setting. When analyzing the 500 test apps, we first assigned each app to a cluster using G-CatA and then applied the appropriate category-specific OC-SVM model to perform anomaly detection.

As in RQ3, we compared DamFlow against FlowDroid configured with two pre-determined source lists—SUSI and DocFlow—while using the same sink list as DamFlow. For this RQ, we used only the DamFlow[GPT] variant, which previously demonstrated the strongest performance. The results are reported in Table 6.

	# Data Flow Pairs		
Approach	**Total**	**Average per app**	**Median**
FlowDroid [DocFlow]	36 219	90	47
FlowDroid [SUSI]	15 006	37	26
DamFlow [GPT]	3194	8	4

Table 6. Number of data flow pairs returned per approach using real-world settings.

As Table 6 shows, DamFlow continues to drastically reduce the number of reported flows, even under real-world conditions. Specifically, compared to FlowDroid[DocFlow], the average number of reported flows drops from 90 to just 8—a 91.1% reduction. Compared to FlowDroid[SUSI], the reduction is 78.4%. These results closely match those observed in the controlled setting of RQ3, where DamFlow[GPT] achieved 87.5% and 71.7% reductions, respectively. This demonstrates that

DamFlow's filtering capabilities generalize well beyond the clean controlled environment, and that using automatically inferred categories via G-CatA does not significantly impact effectiveness. We then carried out a manual analysis using the same methodology described in RQ3. Specifically, we randomly sampled data flow pairs for each approach, using a 95% confidence level and a ±10% margin of error. The results are reported in Table 7.

	# Data Flow Pairs			
Approach	**Inspected**	**Irrelevant**	**P. relevant**	**E. relevant**
FlowDroid [DocFlow]	96	89 [93 %]	6 [6 %]	1 [1 %]
FlowDroid [SUSI]	96	88 [92 %]	7 [7 %]	1 [1 %]
DamFlow [GPT]	94	78 [83 %]	4 [4 %]	12 [13 %]

Table 7. Results of our manual inspection of data flow pairs using real-world settings.

Table 7 confirms the trends observed in RQ3. DamFlow not only reduces the number of reported flows, but also returns a substantially higher proportion of explicitly relevant flows—13% compared to just 1% for both baselines. Despite operating in a noisier setting with automatically inferred categories, DamFlow's false alarm rate (83% irrelevant) remains lower than that of FlowDroid[DocFlow] (93%) and FlowDroid[SUSI] (92%). Moreover, the number of explicitly relevant flows reported by DamFlow[GPT] in this setting (12) is comparable to its performance in RQ3 (11), suggesting that automatic categorization via G-CatA introduces minimal degradation. These results reinforce DamFlow's suitability for practical deployment and highlight its robustness to real-world conditions.

As we anticipated when commenting on the results of RQ3, the number of explicitly relevant flows should not be interpreted in isolation. It is important to consider these figures alongside the drastic reduction in total flows reported by DamFlow. A smaller absolute count of explicitly relevant flows is expected when the tool aggressively filters out noise. What matters in practice is the density of meaningful flows within the output, which DamFlow significantly improves. We return to this point in the Discussion section (Section 6), where we further contextualize the trade-off between comprehensiveness and usability in real-world settings.

Answer to RQ5: *DamFlow achieves substantial reductions in the number of reported data flows (up to 91% compared to FlowDroid[DocFlow]) even when using automatically inferred app categories. It continues to identify more explicitly relevant data flows than baseline approaches, confirming its applicability in real-world deployment scenarios.*

6 Discussion

In RQ3, we demonstrated how DamFlow can address the issue of an incomplete, pre-determined list of sources and sinks, which may lead to missed data leaks, by detecting new sources not included in the SUSI and DocFlow lists. This is achieved through our approach of using backward analysis, relying exclusively on a well-defined set of sinks.

However, computing the recall of our approach—specifically, determining if it fails to identify data flow pairs found by FlowDroid[SUSI] and FlowDroid[DocFlow]—is challenging due to the extremely high number of data flow pairs these tools return. For instance, FlowDroid[DocFlow] identifies 36 196 data flow pairs for the apps in our test set. Indeed, in the absence of ground truth data for leak detection, our only option would be to rely on manual inspection, which is challenging given the tremendous number of data flow pairs returned by FlowDroid when using the DocFlow and SUSI lists. Although ground truths like `DroidBench`[1] and `TaintBench`[41] exist, they consist either of test apps built specifically for benchmarking or of extremely simple real-world apps that lack descriptions or crash when run on a device, thus not representing real-world scenarios. Additionally, this makes it impossible to categorize them into one of the 50 categories of AndroCatSet (as well as to categorize them using G-CatA) for testing our approach or other categorization-based approaches such as AnFlo.

Due to the challenges in computing the recall, we acknowledge that DamFlow might miss relevant data flow pairs. For instance, if we consider the manually confirmed results from FlowDroid[DocFlow] or FlowDroid[SUSI] (Table 2 and Table 6), DamFlow missed a few potentially relevant flows (2 in Table 2 and 1 in Table 6), while correctly identifying all explicitly relevant ones. This behavior is likely due to methods such as `read()` being common in certain categories, causing the anomaly detector to treat them as normal, whereas DamFlow focuses on detecting explicitly relevant flows. However, its significant advantage in drastically reducing false alarms perfectly aligns with the needs of practitioners [42–44] at the cost of potentially missing a few relevant data leaks [45]. Our long-term goal is indeed to produce results that are not only technically sound but also directly actionable by developers and security analysts. DamFlow takes an important step in this direction by dramatically cutting down the number of reported flows while still uncovering leaks that state-of-the-art tools like FlowDroid may miss. Nevertheless, our results in Table 2 show that a non-negligible share of the remaining flows still does not reflect real privacy risks. This points to a clear opportunity for future work: integrating richer semantic analysis, better flow interpretation, or more sophisticated ML techniques (e.g., fine-tuned LLMs or graph-based models) could further improve the precision of anomaly detection and bring us closer to fully actionable outputs. For years, practitioners using static analysis tools have raised concerns about the prevalence of false alarms, which can overwhelm their teams [42–45]. These false alarms not only consume valuable time but also lead to frustration and discouragement in the analysis process. The excessive noise generated by these tools often obscures real issues, hindering analysts' ability to prioritize and address critical problems. Below, we reproduce two concerns raised by practitioners:

> *"Users dislike false positives, often intensely.*
> *If given a choice between a configuration that reports 40 real defects and 10 false positives and a configuration that reports 50 real bugs but with 50 false positives, our experience is that users will almost always prefer the former, even though it is finding fewer real defects."*

> *"False positives and noise are among the biggest*
> *reasons why developers start using static analysis and then don't continue."*

This highlights the need for approaches that filter out false alarms from automated approaches to make their results *actionable*. Our approach aligns perfectly with these demands, as DamFlow drastically reduces the number of false alarms.

On Data Flow Intent and Contextual Interpretation. DamFlow uses app categories (or automatic categorization) as a practical way to approximate an app's expected behavior, highlighting data flows that are uncommon compared to those typically observed in other apps with similar functionalities. We acknowledge that context alone cannot fully determine whether a data flow is appropriate. For example, if location transmission is rare among calculator apps, the occurrence of such a flow in one app is unusual, even if there may be legitimate reasons (e.g., ad preferences). Similarly, if one app uses remote ads while another does not, this could represent a negative (less privacy-compliant) outlier that merits closer examination. DamFlow flags flows as outliers not because they are inherently "unacceptable", but because they are rare among other apps in the same category. These flagged flows are intended to assist analysts by drawing attention to behaviors that deviate from the norm, without making any assumptions about their purpose or acceptability.

7 Limitations and Threats to Validity

Our research is susceptible to threats to validity, which arise from various limitations in our approach. Below, we outline the most significant threats and limitations.

Human Error and Subjectivity. As explained in Section 6, while some ground truths for verifying data leak detection do exist, they cannot be used to test our approach. This implies that, in the absence of suitable ground truth for verifying data-leak detection in a category-based approach, certain aspects of our assessment rely on manual analysis based on our own expertise. It is important to note that, despite following a consistent process, human subjectivity may still influence specific elements. To mitigate this threat to validity, we have shared all our artifacts and code.

FlowDroid Limitations. Our tool is built upon FlowDroid, which inherently carries its limitations that may also affect our approach. For example, scenarios involving user acknowledgment—where an application requests permission to access a privacy-sensitive API—could be misinterpreted as anomalous data flows. While these limitations are noteworthy and warrant further investigation, they fall outside the scope of this paper, which primarily focuses on the effectiveness of DamFlow within the existing framework provided by FlowDroid.

Data Flows Sanitizers. Our current approach does not attempt to detect or model sanitizers along data flow paths—i.e., methods that process or clean data to remove or neutralize sensitive information before it reaches a sink. As a result, some of the flows we report may not represent actual privacy leaks if the sensitive data is sanitized before reaching a sink. Addressing sanitization is an important direction for future work. Similar to tools like FlowDroid, our tool returns a list of potential data leaks. However, DamFlow increases the actionability of these results, which can help developers better assess and prioritize security risks.

8 Related Works

In this section, we present the related works available in the literature that are close to our work.

Most Similar Approaches. The two most related approaches to our work are probably AnFlo [12], against which we demonstrated better performance in RQ3 and MUDFLOW [13]. Demissie et al. proposed AnFlo, a method for detecting anomalous data flows in Android apps by categorizing apps functionally and learning their typical data behaviors, flagging deviations as anomalies. However, unlike our approach, they do not display detailed data flow pairs to users. Instead, they group Android APIs by permission types and identify outliers based on these permission-based groupings [12]. Another relevant approach for detecting abnormal data flows in Android apps is MUDFLOW by Avdiienko et al. [13]. Unlike our approach, however, MUDFLOW functions as a malware detector for Android apps, using abnormal data flows as features to train a classifier that distinguishes between benign and malicious apps. As a result, the user only receives information on whether the analyzed app is malware or not. Both methods rely on a list of sources and sinks generated by SUSI and thus face the same limitations discussed in Section 1, i.e., false alarms and false negatives problems. In contrast, DamFlow is, to the best of our knowledge, the first approach that does not rely on a predefined list of sources, thus addressing the issues associated with such lists, while at the same time, it serves as a general solution for data leak identification purposes.

Backward DataFlow Analysis. Backward data flow analysis has a long-standing history in security research. For example, Krinke and Snelting introduced Valsoft [46], which applied slicing and constraint-solving techniques to validate measurement software by tracing data flows backward. Hammer and Snelting further formalized this notion in JOANA [47], proposing a security theorem and applying backward, flow-sensitive, and context-sensitive information flow control on Java programs using program dependence graphs. In the mobile domain, R-Droid [48] extended this idea to Android by using backward data-dependence slicing to analyze how sensitive data might be influenced throughout an application. However, while these works laid the foundation for backward data flow analysis, they rely on traditional, manually curated lists of sources and sinks. In contrast, DamFlow eliminates the need for a predefined source list and operates solely from sinks, allowing for broader and more flexible identification of abnormal data flows in Android apps.

Context-aware Approaches in General. In our paper, we trained anomaly detection models across app categories, inspired by CHABADA by Gorla et al. [17], which detects malicious apps by identifying deviations from app descriptions. CHABADA classifies Android apps based on their descriptions and then applies a One-Class SVM for anomaly detection on API usage patterns. This paper, along with other studies [34, 49, 50], does not rely on taint analysis to detect data leaks, as we do in DamFlow, but serves as an example of the effectiveness of category-based anomaly detection approaches.

More recently, Malviya et al. [51] use control-flow graph embeddings and machine learning for fine-grained, context-aware permission classification. Unlike our approach, which focuses on backward taint analysis and anomaly detection for data leak identification without relying on source lists, their method targets precise permission misuse detection through graph-based modeling of app behavior.

Data Leaks Detection in Android apps. While FlowDroid [1] represents the current state of the art for static taint analysis, other tools have been developed over the years, such as Amandroid [52], LeakMiner [53], DroidSafe [54], IccTA [55], and many others [56–58]. DeepFlow [59], a deep learning-based approach by Zhu et al., analyzes data flows in Android apps. Similar to MUDFLOW, it uses abnormal data usage to identify malware. LeakSemantic [60] by Fu et al. is a framework designed to automatically identify abnormal sensitive network transmissions in mobile applications by combining hybrid program analysis with machine learning. Both DeepFlow and LeakSemantic rely on the list generated by SUSI. LeakDetector [61] by Zhou et al. is a tool that identifies vulnerabilities in the Android framework that allow unauthorized apps to intercept privacy-sensitive data from Intent objects. LeakDetector relies on permissions to identify privacy-sensitive APIs, similar to what AnFlo does. Chen et al. proposed ClipboardScope [62], which uses static program analysis to examine clipboard data usage in mobile apps by analyzing its validation and data flow. The authors of ClipboardScope use a list of only three methods as sources due to their focus on clipboard data usage. These tools, therefore, focus only on specific problems, such as clipboard data, intents, network transmission, and malware detection. Furthermore, if they rely on the list generated by SUSI, they will still suffer from the usual issues of false alarms and false negatives. Instead, our approach, DamFlow, serves as a general solution for data leak identification purposes while not relying on a pre-determined list of sources and sinks. We leave for future work the possibility of observing how the performance of these data leak identification tools changes when backward analysis with only a list of sinks is used instead of relying on the SUSI list.

9 Conclusions

While FlowDroid is a state-of-the-art tool for the static analysis of Android apps, it is impractical for identifying data leaks in real-world scenarios due to ① a “flood“ of false alarms caused by the difficulty in correctly identifying privacy-sensitive sources, and ② undetected data leaks, which may result from incomplete lists of sources and sinks. In this paper, we propose DamFlow, a novel approach that combines backward taint analysis with context-aware anomaly detection, using only a list of well-defined sinks. This addresses both issues simultaneously, making FlowDroid more practical.

We compared DamFlow against FlowDroid’s output using lists of sources and sinks generated by automated approaches such as SUSI and DocFlow. Our results demonstrate that DamFlow drastically reduces the number of returned data flows, filtering out irrelevant data flows while also detecting sources not present in the pre-determined lists from SUSI and DocFlow. We also compared DamFlow against the category-based approach AnFlo, showing that our approach outperforms it. In summary, DamFlow advances Android app analysis by reducing noise and enhancing source detection, providing a more effective solution for real-world data leak identification.

10 Acknowledgement

This research was funded in part by the Luxembourg National Research Fund (FNR), grant reference NCER22/IS/16570468/NCER-FT, REPROCESS grant reference C21/IS/16344458, and UNLOCK grant reference C23/IS/18154263.

References

[1] S. Arzt, S. Rasthofer, C. G. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. Mcdaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:12083354

[2] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[3] S. Arzt, S. Rasthofer, and E. Bodden, "Susi: A tool for the fully automated classification and categorization of android sources and sinks," 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:15997711

[4] M. Tileria, J. Blasco, and S. K. Dash, "Docflow: Extracting taint specifications from software documentation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623312

[5] M. Kober, J. Samhi, S. Arzt, T. F. Bissyandé, and J. Klein, "Sensitive and personal data: What exactly are you talking about?" in *2023 10th International Conference on Mobile Software Engineering and Systems 2023 (MobileSoft)*, May 2023, pp. 70–74.

[6] J. Samhi, M. Kober, A. Kabore, S. Arzt, T. F. Bissyande, and J. Klein, "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)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2023, pp. 783–794. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER56733.2023.00091

[7] W. Wang, J. Wei, S. Zhang, and X. Luo, "Lscdroid: Malware detection based on local sensitive api invocation sequences," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 174–187, 2020.

[8] M. Junaid, D. Liu, and D. Kung, "Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models," *Computers And Security*, vol. 59, pp. 92–117, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404816300037

[9] L. Luo, E. Bodden, and J. Späth, "A qualitative analysis of android taint-analysis results," *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 102–114, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:204781500

[10] T. Lange, "Implementation and evaluation of a static backwards data flow analysis in flowdroid," B.S. thesis, Technische Universität.

[11] B. Scholkopf, J. C. Platt, J. Shawe-Taylor, A. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural Computation*, vol. 13, pp. 1443–1471, 2001. [Online]. Available: https://api.semanticscholar.org/CorpusID:2110475

[12] B. F. Demissie, M. Ceccato, and L. K. Shar, "Anflo: Detecting anomalous sensitive information flows in android apps," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, 2018, pp. 24–34.

[13] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015, pp. 426–436.

[14] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale." in *TRUST*, ser. Lecture Notes in Computer Science, S. Katzenbeisser, E. R. Weippl, L. J. Camp, M. Volkamer, M. K. Reiter, and X. Zhang, Eds., vol. 7344. Springer, 2012, pp. 291–307. [Online]. Available: http://dblp.uni-trier.de/db/conf/trust/trust2012.html#GiblerCEC12

[15] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, jun 2014. [Online]. Available: https://doi.org/10.1145/2619091

[16] Y. Nan, Z. Yang, X. Wang, Y. Zhang, D. Zhu, and M. Yang, “enckfinding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps,” in *Network and Distributed System Security Symposium*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:51952918

[17] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, “Checking app behavior against app descriptions,” in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 1025–1035.

[18] V. Chandola, A. Banerjee, and V. Kumar, “Anomaly detection: A survey,” *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

[19] M. Alecci, J. Samhi, T. F. Bissyandé, and J. Klein, “Revisiting android app categorization,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[20] A. Alsubaihin, F. Sarro, S. Black, L. Capra, M. Harman, Y. Jia, and Y. Zhang, “Clustering mobile apps based on mined textual features,” 09 2016, pp. 1–10.

[21] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, “A survey of app store analysis for software engineering,” *IEEE Transactions on Software Engineering*, vol. 43, no. 9, pp. 817–847, 2017.

[22] D. Surian, S. Seneviratne, A. Seneviratne, and S. Chawla, “App miscategorization detection: A case study on google play,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 8, pp. 1591–1604, 2017.

[23] S. Arzt and E. Bodden, “Stubdroid: automatic inference of precise data-flow summaries for the android framework,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 725–735.

[24] T. Reps, S. Horwitz, and M. Sagiv, “Precise interprocedural dataflow analysis via graph reachability,” in *POPL ’95*, 1995, pp. 49–61.

[25] S. Arzt, “Static data flow analysis for android applications,” 2017.

[26] R. Vallee-Rai and L. J. Hendren, “Jimple: Simplifying java bytecode for analyses and transformations,” Technical report, McGill University, Tech. Rep., 1998.

[27] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: learning distributed representations of code,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, jan 2019. [Online]. Available: https://doi.org/10.1145/3290353

[28] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “Codebert: A pre-trained model for programming and natural languages,” 2020.

[29] OpenAI, “New embedding models and api updates,” 2024, accessed: 2024-06-21. [Online]. Available: https://openai.com/index/new-embedding-models-and-api-updates/

[30] ——, “Embeddings use cases,” 2024, accessed: 2024-06-21. [Online]. Available: https://platform.openai.com/docs/guides/embeddings/use-cases

[31] N. Muennighoff, N. Tazi, L. Magne, and N. Reimers, “Mteb: Massive text embedding benchmark,” 2023. [Online]. Available: https://arxiv.org/abs/2210.07316

[32] R. Meng, Y. Liu, S. R. Joty, C. Xiong, Y. Zhou, and S. Yavuz, “Sfr-embedding-mistral: Enhance text retrieval with transfer learning,” https://blog.salesforceairesearch.com/sfr-embedded-mistral/, 2024, salesforce AI Research Blog.

[33] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, “Mistral 7b,” 2023.

[34] M. Alecci, J. Samhi, L. Li, T. F. Bissyandé, and J. Klein, “Improving logic bomb identification in android apps via context-aware anomaly detection,” *IEEE Transactions on Dependable and Secure Computing*, 2024.

[35] J. Samhi, L. Li, T. F. Bissyande, and J. Klein, “Difuzer: Uncovering suspicious hidden sensitive operations in android apps,” in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 723–735. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3510003.3510135

[36] J. Developers, “Joblib: running python functions as pipeline jobs,” https://github.com/joblib/joblib.

[37] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, “Pscout: analyzing the android permission specification,” in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 217–228.

[38] M. Alecci, P. J. R. Jiménez, K. Allix, T. F. Bissyandé, and J. Klein, “Androzoo: A retrospective with a glimpse into the future,” in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 389–393.

[39] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, “Androzoo: Collecting millions of android apps for the research community,” in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR ’16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[40] VirusTotal, “Virustotal: Free online virus, malware and url scanner,” https://www.virustotal.com/, accessed: 2025-05-15.

[41] L. Luo, F. Pauck, G. Piskachev, M. Benz, I. Pashchenko, M. Mory, E. Bodden, B. Hermann, and F. Massacci, “Taintbench: Automatic real-world malware benchmarking of android taint analyses,” *Empirical Software Engineering*, vol. 27, pp. 1–41, 2022.

[42] A. Hicken, “False positives in static code analysis,” 2023, accessed: 2024-08-01. [Online]. Available: https://www.parasoft.com/blog/false-positives-in-static-code-analysis/

[43] PVS-Studio, “False positives of the static code analyzer,” 2021, accessed: 2024-08-01. [Online]. Available: https://pvs-studio.com/en/blog/terms/6461/

[44] codecurmudgeon, “What went wrong with static analysis?” 2011, accessed: 2024-08-01. [Online]. Available: https://codecurmudgeon.com/wp/2011/08/what-went-wrong-with-static-analysis/

[45] grammatech, “Human factors in evaluating static analysis tools,” 2017, accessed: 2024-08-01. [Online]. Available: https://www.grammatech.com/learn/human-factors-in-evaluating-static-analysis-tools/

[46] J. Krinke and G. Snelting, “Validation of measurement software as an application of slicing and constraint solving,” *Information and Software Technology*, vol. 40, no. 11-12, pp. 661–675, 1998.

[47] C. Hammer and G. Snelting, “Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs,” *International Journal of Information Security*, vol. 8, no. 6, pp. 399–422, 2009.

[48] M. Backes, S. Bugiel, E. Derr, S. Gerling, and C. Hammer, “R-droid: Leveraging android app analysis with static slice optimization,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 129–140.

[49] C. Zhang, H. Wang, R. Wang, Y. Guo, and G. Xu, “Re-checking app behavior against app description in the context of third-party libraries,” in *International Conference on Software Engineering and Knowledge Engineering*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:53240470

[50] S. Ma, S. Wang, D. Lo, R. H. Deng, and C. Sun, “Active semi-supervised approach for checking app behavior against its description,” in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, 2015, pp. 179–184.

[51] V. K. Malviya, Y. N. Tun, C. W. Leow, A. T. Xynyn, L. K. Shar, and L. Jiang, “Fine-grained in-context permission classification for android apps using control-flow graph embedding,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1225–1237.

[52] F. Wei, S. Roy, X. Ou, and Robby, “Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps,” *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, pp. 1–32, 2018.

[53] Z. Yang and M. Yang, “Leakminer: Detect information leakage on android with static taint analysis,” in *2012 Third World Congress on Software Engineering*. IEEE, 2012, pp. 101–104.

[54] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, “Information flow analysis of android applications in droidsafe.” in *NDSS*, vol. 15, no. 201, 2015, p. 110.

[55] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, “Iccta: detecting inter-component privacy leaks in android apps,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE ’15. IEEE Press, 2015, p. 280–291.

[56] J. Samhi, A. Bartel, T. F. Bissyande, and J. Klein, “Raicc: Revealing atypical inter-component communication in android apps,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 1398–1409. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00126

[57] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, “Jucify: A step towards android code unification for enhanced static analysis,” in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2022, pp. 1232–1244. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/3510003.3512766

[58] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, “Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1137–1150. [Online]. Available: https://doi.org/10.1145/3243734.3243835

[59] D. Zhu, H. Jin, Y. Yang, D. Wu, and W. Chen, “Deepflow: Deep learning-based malware detection by mining android application for abnormal usage of sensitive data,” in *2017 IEEE symposium on computers and communications (ISCC)*. IEEE, 2017, pp. 438–443.

[60] H. Fu, Z. Zheng, S. Bose, M. Bishop, and P. Mohapatra, “Leaksemantic: Identifying abnormal sensitive network transmissions in mobile applications,” in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[61] H. Zhou, X. Luo, H. Wang, and H. Cai, “Uncovering intent based leak of sensitive data in android framework,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 3239–3252. [Online]. Available: https://doi.org/10.1145/3548606.3560601

[62] Y. Chen, R. Tang, C. Zuo, X. Zhang, L. Xue, X. Luo, and Q. Zhao, “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*, 2024, pp. 1–13.