Inferring permission requests purpose towards improving privacy in mobile devices



Leandro Nascimento Pereira
Master in Information Security
Department of Computer Science
Faculty of Sciences of the University of Porto
2025

Inferring permission requests purpose towards improving privacy in mobile devices

Leandro Nascimento Pereira

Dissertation carried out as part of the Master in Information Security

Department of Computer Science

2025

Supervisor
Prof. João P. Vilela, Assistant Professor, University of Porto

Co-supervisor
Ricardo Mendes, Independent Researcher

“ Arguing that you don’t care about the right to privacy because you have nothing to hide is no different than saying you don’t care about free speech because you have nothing to say. ”

Edward Snowden

Acknowledgements

I would like to express my gratitude to all those who, in some way, allowed this thesis to come to fruition.

First, I would like to thank Professor João Vilela and Ricardo Mendes for their invaluable guidance, encouragement, and support throughout the course of this research. Their expertise and thoughtful feedback have been essential to the completion of this work. I would also like to thank Mariana Cunha for her helpful input and for pointing me toward key references that supported the completion of this thesis.

In closing, I wish to dedicate a special acknowledgment to my mom, dad, sister, and grandmother. Their unwavering faith in me and their constant emotional and mental support have accompanied me at every stage of this work. Through every challenge and doubt, they have provided comfort, motivation, and unconditional love. For always believing in me, even when I struggled to believe in myself, I am endlessly thankful.

Resumo

Dispositivos móveis geralmente dependem de um modelo de permissões em tempo de execução em que as permissões são concedidas através de um mecanismo único e centralizado que fornece a essa aplicação acesso a um recurso sensível durante um intervalo de tempo definido, como apenas durante aquela sessão ou de forma indefinida. Obter uma compreensão mais profunda dos propósitos subjacentes a esta recolha de dados por aplicações móveis representa um passo crucial para a proteção da privacidade dos utilizadores no uso de smartphones. Ao esclarecer estas práticas, os utilizadores podem ser dotados do conhecimento necessário para tomar decisões mais conscientes e informadas sobre as suas interações digitais. No entanto, o propósito da recolha de dados é frequentemente ambíguo ou até mesmo não divulgado pelas próprias aplicações, uma falta de transparência que afeta de forma significativa a possibilidade de exercer um controlo rigoroso e eficaz sobre a privacidade pessoal. Esta dissertação aborda o desafio da classificação dos propósitos em dispositivos Android através de uma análise sistemática das práticas de recolha de dados realizadas pelas aplicações, com o objetivo de revelar a finalidade subjacente a essas recolhas. Para alcançar isto, nós propomos LeakHound, uma ferramenta que monitoriza e analisa o tráfego de rede gerado por uma aplicação, enriquece estas observações com detalhes contextuais relacionados com os pedidos e utiliza a combinação desta informação para criar uma base de dados usada para classificar o propósito de cada pedido de rede efetuado. Os nossos resultados mostram que este contexto adicional sobre os pedidos de rede melhora a classificação com uma taxonomia de propósito geral em 8.2 pontos percentuais sobre o estado de arte, simultaneamente identificando os conjuntos de *features* de contexto que são, não só necessários mas também suficientes para classificar cada um dos tipos de informação, permitindo uma classificação de dados mais eficiente, especifica e direcionada.

Palavras-chave: Privacidade, Android, Inferencia de Propósito, Análise de Rede, *API Hooking*, *Machine Learning*

Abstract

Mobile devices usually rely on a runtime permission model in which permissions are granted through a singular and central mechanism that provides that application access to a sensitive resource for the defined time window, such as during that session or in perpetuity. Gaining a deeper understanding of the underlying purposes behind this data collection of mobile applications represents a critical step toward safeguarding user privacy in smartphone usage. By shedding light on these practices, users can be empowered with the knowledge necessary to make more conscious and informed decisions about their digital interactions. However, the purpose of data collection is often ambiguous or even undisclosed by the applications themselves, a lack of transparency that significantly affects the possibility of exercising precise and effective control over one's personal privacy. This thesis addresses the challenge of purpose classification for Android devices through a systematic analysis of the data collection practices carried out by applications, to uncover the purpose behind said collections. To achieve this, we propose LeakHound, a tool that monitors and analyzes the network traffic produced by applications, augments these observations with contextual details related to the requests, and leverages the combined information to generate a dataset used to classify the intended purpose of each outbound network interaction. The results show that this additional context over the network requests improves the classification with a general-purpose taxonomy by 8.2 percentage points over the state of the art, while simultaneously identifying the subset of contextual features that are both necessary and sufficient for accurately classifying each data type, enabling more efficient and targeted data classification.

Keywords: Privacy, Android, Purpose Inference, Network Analysis, API Hooking, Machine Learning

Table of Contents

List of Figures vii
List of Abbreviations viii

1. Introduction 1
2. State of Art 4
   2.1. Background 4
      2.1.1. Android Permissions 4
      2.1.2. Foundational Concepts 6
         2.1.2.1. Taint Analysis 6
         2.1.2.2. Hooking 8
   2.2. Related Work 9
      2.2.1. General Overview 10
      2.2.2. Inter-Work Comparison Limitations 15
      2.2.3. MobiPurpose 23
      2.2.4. FlowIntent 26
      2.2.5. Text mining approach by Wang et al. [70] 28
   2.3. Limitations 31
3. Proposed Methodology 36
   3.1. Problem 36
      3.1.1. Threat Model 37
   3.2. Proposed Solution 38
      3.2.1. LeakHound 40
         3.2.1.1. Static Analysis 43
         3.2.1.2. Dynamic Analysis 43
         3.2.1.3. Call Stack 45
         3.2.1.4. Runtime information 49
      3.2.2. Key Feature Selection 50
         3.2.2.1. Network Traffic 50
         3.2.2.2. Data Flow 53
         3.2.2.3. Third Party Libraries 57
      3.2.3. Purpose Classification 58
   3.3. Challenges 61
4. Evaluation 65
   4.1. Dataset 66
   4.2. Type Inference 66

4.3. Purpose Classification 67
4.3.1. Analysis 70
4.4. Limitations 75
5. Conclusions and Future Work 78
5.1. Future Work 80

Bibliography 83
A. Purpose Classification Taxonomy 96
B. Evaluation Tables 102

List of Figures

2.1. MobiPurpose Workflow - Extracted from [55] 23
2.2. FlowIntent system architecture - Extracted from [40] 26
2.3. Text mining tool, Static Approach - Extracted from [70, 71] 29
2.4. Text mining tool, dynamic approach - Extracted from [70, 71] 30

3.1. LeakHound system architecture diagram 41
3.2. Callstack Reconstruction with truncated function names. 49
3.3. Callstack Distance Example Graph 55

4.1. Type Inference Results 67

List of Abbreviations

A&T Advertisement and Tracking. 12, 20, 58

API Application Programming Interface. 9, 40, 45, 46, 50, 62

APK Android Package. 20, 40, 42, 79

ART Android Runtime. 11

ATS Advertisement and Tracking Services. 3, 17, 21, 78

CNN Convolutional neural network. 14

DC Device Controller. 43– 45

DVM Dalvik Virtual machine. 11

GDPR General Data Protection Regulation. 2

GUI Graphical User Interface. 2

IP Intelectual Property. 9

IRM Identifier Renaming. 33

KVP Key-Value Pairs. 23– 25, 50– 52, 54, 58– 60, 64, 69

ME Maximum Entropy. 25

MitM Man-in-the-Middle. 14, 21, 23, 42, 43, 45, 78

ML Machine Learning. 23, 27, 40, 60, 63, 64, 66, 79

NIDS Network Intrusion Detection System. 27

NLP Natural Language Processing. 23, 31, 52, 59, 60, 71, 81

OCR Optical Character Recognition. 26

OS Operating System. 2, 14, 17, 20, 22, 34

PII Personally Identifiable Information. 13, 14, 16, 18, 20, 81, 83

RAM Random Access Memory. 42, 61, 77

RASP Runtime Application Self-Protection. 9

RPC Remote Procedure Call. 10

SDK Software Development Kit. 1, 15, 31, 32

SQLi SQL Injection. 7, 27

SVM Support Vector Machine. 28

TPL Third Party Libraries. 13, 17, 20, 21, 36, 37, 40, 43, 50, 57, 58, 64– 66, 70, 72, 78– 80

UI User Interface. 7, 12, 15, 21, 29, 64

UUID Universally unique identifier. 23

VM Virtual Machine. 11

VPN Virtual Private Network. 21, 40, 42, 44

VR Virtual Reality. 17, 18

XSS Cross Site Scripting. 7

1. Introduction

Most Android users routinely grant permission requests without fully understanding how their personal data will be accessed, used, or shared. While these permissions are often presented as necessary for app functionality, they frequently enable extensive data collection that goes far beyond the user's expectations. These data harvesting practices of personal information have become a standard practice in mobile apps' ecosystems, raising serious concerns about transparency, informed consent, and user privacy. The scale and severity of such privacy violations have only grown in recent years. For instance, in early 2025, a class-action lawsuit accused Amazon of secretly collecting detailed geolocation data through its Ads Software Development Kit (SDK) embedded in thousands of third-party Android applications. The situation with third-party SDK is so problematic that for Android 14, Google launched the SDK Runtime[1], which isolates the SDK from the actual application to strengthen users' privacy and control. Although these data collection practices require user consent, as in the user granting access to that data type, the context surrounding these requests is often ambiguous, misleading, or missing completely, as found in Liu et al. [1], Girish et al. [2], which looks at the appropriateness of permission requests' rationale and their presence, or lack thereof. This issue is further enunciated by the frequency at which these requests occur. According to Mendes et al. [3], users would, if the permission grants were not cached, be prompted with, on average, $\sim$35 requests/hour. Yet, by caching these decisions, we essentially create a unique central point of consent which will apply for all usages subsequent to the permission granting, with exceptions for newer Android versions where permission grants can have different durations. These conditions significantly impair users' ability to make informed, privacy-preserving decisions, despite formal consent mechanisms being in place.

In the pursuit of better controls, prior research on mobile privacy done by Liu et al. [4] reinforced the push towards purpose-centric permission models, as purpose is one of, if not the, best feature for decision making when it comes to permission granting in mobile applications. Knowing for *what* your data is being collected for better equips the user to make decisions on their own about what information they are willing to share with that app/service, as privacy is subjective and each user will have a different threshold for what they are comfortable sharing. This idea is further complemented by a survey done

[1] `https://privacysandbox.google.com/private-advertising/sdk-runtime`

by Smullen et al. [5], where the authors confirmed that purpose has a significant effect on participants' expressed preferences for app permissions. Thus, these approaches heavily rely on a purpose inferring model capable of accurately and consistently identifying purpose associated with the data harvesting practices of these apps and services.

However, the gap between nominal consent and users' actual understanding of data usage undermines their overall privacy and directly contradicts the principles enforced by the General Data Protection Regulation (GDPR) [6]. According to Article 4(11) GDPR, consent must be "freely given, specific, informed and unambiguous" as to be considered valid and acceptable. Furthermore, Article 7(4) states that, "When assessing whether consent is freely given, utmost account shall be taken of whether [..] the performance of a contract [..] is conditional on consent to the processing of personal data that is not necessary for the performance of that contract.", which invalidates any consent of personal information usage, if this consent is bundled with other non-essential uses, in a manner such that, the user either accepts everything, or is barred from using the service. As an example for this situation, the consent is invalid for a transportation service which requests geolocation information to provide the service **and** for advertisement purposes, while making it so that the request for both use cases is the same, essentially forcing the user to accept all use cases or none at all. In response to the increasing privacy concerns, significant research has been conducted in this niche field, but the diverse scope of objectives defined by different studies has led to a fragmented landscape. Most of these works look at who sent which data to whom, but often overlook the why. This is a very important question as it not only helps in identifying potential privacy violations but also empowers non-technical users with the knowledge of what their data is being used for. For the works that do look at the why, each defines its own end goals, resulting in a lack of standardization, usually represented as, but not exclusive to, different taxonomies with different granularity, different methods of classification (e.g. classifying per network request vs app-wise) or environment/ Operating System (OS)/Architecture. These approaches also cover many diverse methodologies: Looking at Graphical User Interface (GUI) information and its relation with the collection of personal data, inferring the purpose of data flows within the application, studying the application's description and privacy policies, or analyzing the network traffic. Among these approaches, network analysis stands out, as it gives us visibility over who is collecting this information. That context allows us to better assess the purpose of the data collection, since if we know, for example, that the data is going to an advertising service, the purpose will most likely be advertising. Furthermore, as Falaki

et al. [7] and Reardon et al. [8] concluded, HTTP(s) is the most common protocol for communication in Android, as well as the most frequently used protocol for data leakage and exfiltration. However, for this methodology, existing state-of-the-art approaches either completely omit the context surrounding the network requests they analyze, or when context is included, it is tailored towards a different objective, such as identifying Advertisement and Tracking Services (ATS) traffic only or detecting privacy policy violations. As a result, the contextual features they introduce are shaped by those goals and not directly applicable to the problem at hand: inferring the purpose of data collection practices towards improving privacy controls.

In this thesis, we aim to develop a model capable of comprehensive and generic purpose classification, in particular, one that considers all potential uses of data, as opposed to focusing on ATS, and their surrounding context. To achieve our goal, we leverage network analysis to identify *when*, *which*, and to *whom* the data is sent, while extracting other metadata that further captures the context of the data collection practice. Since no dataset or tool to collect all this information is available, we propose LeakHound[1], a framework that collects network requests, static, and runtime information about the application. We then correlate all of these sources of data to construct a context surrounding each network request, and train a machine learning model that classifies the information being sent out of the application using a comprehensive and generic purpose classification taxonomy. In the end, we were able to achieve an average performance improvement of 8.2 percentage points over the state of the art, MobiPurpose, across all data types, by the introduction of 7 new context features over 13 different purposes. Due to the lack of publicly available datasets for this task, we make ours public[1].

The remainder of this thesis is structured as follows. Chapter 2 reviews related work and provides the necessary background for understanding the technical and conceptual foundations of this research. Chapter 3 details the methodology used, including the threat model, system design, data collection, feature selection, and challenges. Chapter 4 presents the results of our analysis, discusses key findings, and goes over the threats to validity possibly affecting those results. Chapter 5 reflects on the implications of these findings and concludes the thesis by outlining directions for future work.

[1] https://github.com/LeoPer02/LeakHound

2. State of Art

This chapter provides an understanding of the current state of the art by reviewing the literature on data harvesting purpose inference on mobile devices and applications. We begin by covering some background knowledge essential to understanding the methodologies used/proposed in the state of the art review. We then follow this with a general overview of the state of the art, covering diverse methodologies in the field of purpose classification and related topics. From there, we introduce a considerable limitation on the purpose classification of inter-work comparison. Following this, we introduce, with greater detail, 3 works we believe to be representative of efforts towards purpose classification in mobile devices/apps and the diversity of methodologies applied. Finally, we conclude by addressing the limitations associated with the state of the art.

2.1. Background

2.1.1 Android Permissions

Android applications can request access to various permissions within the system. These permissions allow the applications to access specific resources, enabling them to provide services to users, such as accessing the user's location to offer navigation. However, different permissions pose different levels of risk to the user's privacy/security. Given the potential underlying risks, Android has categorized the permissions into different permission groups.

Install-time permissions are those that, as the name suggests, are granted at the moment of installation without requiring the user's consent. These permissions generally pose a lower privacy risk to the user and, as a result, are only presented to the user as a warning during the installation process. In the install-time permission category, there are two distinct protection levels. Permissions with a "normal" protection level provide access to data and actions outside the app's sandbox, but pose minimal risk to the user's privacy. Examples include INTERNET, ACCESS_NETWORK_STATE, and WAKE_LOCK. On the other hand, "signature" protection level permissions are automatically granted only if the requesting app is signed with the same certificate as the app or Operating System that implements the permissions.

For operations that present a significant risk to the user's privacy, there are runtime permissions. These are classified under the "dangerous" protection level and allow access to

sensitive information, such as CAMERA, ACCESS_FINE_LOCATION, and SEND_SMS, among others. Due to their sensitive nature, starting from Android 6 (API level 23), these permissions must be requested during the application's execution, requiring explicit user consent. Lastly, operations that access particularly sensitive system resources or resources not directly tied to user privacy are under special permissions, classified under "appop" protection level. One example of these permissions is FOREGROUND_SERVICE_SPECIAL_USE. Additionally, to utilize any of these permissions, they must be explicitly declared in the application's Manifest file. Nonetheless, since Android 6, several privacy-preserving improvements have been introduced. Notably, the "Ask Every Time" or "While in Use" permission options provide more granular control over when an application can access information and for how long. Essentially, these permissions operate on an "ask-on-first-use" basis, while also allowing users to limit the duration of granted consent.

Nonetheless, a common misconception about the runtime permission model is that permission requests are inherently tied to a single, immediate use of the associated data. While it may seem intuitive to assume that requesting a permission implies a one-time data retrieval, this is not entirely accurate. When an application requests a permission, it is essentially requesting access rights, akin to acquiring a "data access ticket". The application does not retrieve the data itself but instead gains the ability to access it when needed.

This distinction has significant implications. Because a permission request is not directly linked to a specific instance of data retrieval in the application's code, let alone a specific and singular usage, it becomes challenging to label a permission request with a definitive purpose. Consider an analogy: the requested information resides in a locked box, and the permission request serves as a request for the key to unlock the box, not the box's contents. Even if this was not the case, as covered by Mendes et al. [3], a user would experience around $\sim$35 permission requests per hour, making it completely impractical and unbearable for the users to carefully and attentively understand the request and act on it. This establishes a centralized point for granting permissions, allowing the application to access resources for all users within the specified duration, which can range from a single session to indefinitely.

In practice, this means that when an app requests a permission, systems attempting to infer the purpose of the actual permission request have essentially two options for communicating that to users. They can either (1) inform users of all potential uses of the permission throughout the app or (2) highlight only those uses that might be deemed ”worrisome” or sensitive. Yet, both of these options remove any granularity on the classification which we might have expected from classifying the purpose at the permission request, essentially reducing the alert to “The app will use data X for the following purposes: ...”, which is no better than an app-wise classification, meaning that the classification of the purpose at the permission request event provides no additional benefit.

Having established an understanding of how Android permissions function and the role they play in regulating application access, it is now important to shift focus toward the foundational concepts that support the analysis in this work. In particular, the following section introduces the key tools and methodologies that provide the basis for examining application behavior, used to classify the data collection purpose, in greater depth.

2.1.2 Foundational Concepts

Inferring the purpose of data usage is challenging, and different methodologies can be considered. Among the methodologies found in the state of the art, two stand out: Taint Analysis and API Hooking, or simply Hooking. The following sections detail these methodologies.

2.1.2.1 Taint Analysis

In the privacy realm, taint analysis is an essential tool given its capability to detect leaks of information. Its goal is to identify data flows between sources, where sensitive data originates, and sinks, where that data leaves or is used within the system in potentially dangerous ways. Taint analysis works by tainting, i.e., marking or tagging, data that originates from a source and tracking its propagation through the program. When such a flow is identified, that is, when sensitive data travels from a source S_o to a sink S_i, a potential data leak is flagged. This information can be employed to understand when and where this data is being used, which is essential when trying to classify the purpose of data collection. It is also worth mentioning that taint analysis is not strictly used for data leak detection. As any other tool, taint analysis can be adapted in order to achieve different

goals such as detecting Cross Site Scripting (XSS) attacks or SQL Injection (SQLi). Furthermore, taint analysis can be performed either statically or dynamically, each presenting a different trade-off on performance and usability, as detailed below.

Static Taint Analysis: Static taint analysis inspects the program code without executing it, enabling large-scale analysis but potentially leading to false positives. It achieves this by analyzing the bytecode or source code and tracking data flows throughout the program. One of the primary advantages of this approach is its ability to provide better coverage, as it theoretically examines all possible execution paths (except for those that involve dynamic code loading). Additionally, static taint analysis can be performed pre-execution, requiring only the program's binary or source files, making it useful for early detection (i.e., before permission requests) of data leaks. In contrast, static taint analysis tends to produce a higher rate of false positives, since it flags flows that are impossible in reality. Additionally, it has a limited ability to handle dynamic behaviors such as dynamic code loading, reflection, and similar runtime mechanisms. Finally, static analysis is oblivious to runtime information, such as system states, variable values, and multi-threading, which further limits its accuracy.

Dynamic Taint Analysis: Dynamic taint analysis examines an app during execution by tracking the actual states of variables and flagging leaks when tainted information reaches a sink. This approach typically results in very few false positives. Additionally, by analyzing the program during runtime, it has access to current system information, external conditions, and actual variable states. This capability provides a substantially more accurate analysis in terms of precision by avoiding over-approximations. However, because dynamic taint analysis is tied to execution, it suffers from low coverage of the program, particularly for functionalities that are only triggered under specific conditions (e.g., when the phone is low on battery or functionality is triggered only on certain days). As a result, this leads to a higher false negative rate. To improve coverage, this process is typically supplemented with User Interface (UI) exercisers, tools designed to attempt to trigger every possible state of the app. Nevertheless, even with these tools, achieving decent coverage remains a challenge as highlighted by [9–17].

Additionally, taint analysis tools present various sensitivities, which work as a set of characteristics, such as order and context, that the tool takes into account while performing

the analysis. These sensitivities determine the level of granularity or precision that these tools achieve when tracking data flows, making it important to have a comprehensive understanding of them. As outlined in [18], a tool may exhibit the following sensitivities:

Flow-Sensitivity: This approach considers the sequence of statements during program execution, calculating distinct analysis information for each statement. In contrast, flow-insensitive analysis generates a single overarching solution for the entire program, ignoring the order of execution.

Context-Sensitivity: This method accounts for the context in which procedures are called, enabling the analysis to produce separate information for different invocations of the same procedure. In contrast, context-insensitive analysis combines all invocation events of a procedure.

Object-Sensitivity: This analysis type differentiates between various ”host” objects related to the same field (or attribute) within an object. In contrast, object-insensitive analysis aggregates data for a field across all instances of the same class, disregarding the specific context of each object.

2.1.2.2 Hooking

Hooking, or method/API Hooking, is a dynamic analysis technique commonly used in reverse engineering and software instrumentation. It is a technique that taps into defined methods of a program during runtime to observe, modify, or redirect execution. With hooking, one can inject code inside of a defined function, at runtime, either complementarily, by adding the code in the beginning, middle, or end of the original code, or in substitution, by completely changing the method behavior, to control the program functionality. As an illustrative example, suppose that we have a function called $checkLicenseKey()$. This function will, in some way, verify whether the current program has a valid license key to utilize the software or not. A way one could try to bypass this check would be by hooking, i.e., injecting custom code, into this $checkLicenseKey()$ function and making it always return true. Method hooks are also extremely useful when it comes to retrieving runtime information from the program. Say, for example, that you want to know the values of the arguments reaching a specific function within 3^{rd} party software. By hooking this method, you can not only control its behavior, but also check the values of the arguments passed

during execution, which can help debug/reverse engineer software. These hooks can be used for multiple ends, both benign and malicious; however, they serve as a powerful tool for dynamic analysis and reverse engineering.

In the privacy realm, this technique can be used to monitor Application Programming Interface (API) calls and which information they provide, such as sensitive data like Location and Contacts, retrieve keys to break encryption, perform network analysis, by hooking the network APIs used, among other use cases. There are multiple hooking frameworks which support the Android Environment, such as Frida [19] and XPosed/LSPosed [20]. Frida, for example, accomplishes this by injecting JavaScript into a running application's process. These scripts can modify a function's code or logic to change its behavior or, when a Frida server is installed on the device, create a communication bridge that enables remote inspection and extraction of runtime information. There are also other hooking frameworks which are mainly focused on different environments like Windows, Linux, macOS, etc. In this category, we have tools such as Detours [21] and Intel Pin [22]. It is also important to understand that, although powerful, these frameworks have their limitations, especially when it comes to Android. In most scenarios, hooking an app in Android may require the device to be rooted/jailbroken, with exceptions such as in the case of Frida, where, by repackaging the app to include a Frida-gadget or using a debugger, one can possibly achieve the same result. Nonetheless, it requires elevated permissions which, depending on the context, may open the door to root checks–when the app checks if it's running inside a rooted device, which usually is undesirable, due to security, privacy and Intelectual Property (IP) concerns. Furthermore, commercial apps, especially those from the financial sector, employ protection mechanisms called Runtime Application Self-Protection (RASP) which look for hooks, rooted devices, integrity violations, among other checks, to ensure the app is running within a safe environment. Examples of these protections are Dexguard [23], AppDome [24], and AppSealing [25].

2.2. Related Work

This section reviews previous work relevant to the topic of this thesis, specifically, approaches that try to infer the purpose of permissions in mobile devices. In section 2.2.1, we will cover a vast number of works in the field of purpose inference and works closely related to that goal, but with less detail. We will then, in section 2.2.2, take a look at a common challenge in the topic of purpose inference of inter-work comparison. Following that, we will narrow our focus to 3 works that are better representative of the diverse

methodologies and approaches for the problem of purpose inference, which will be reviewed in greater detail. And finally, we will address the limitations associated with the state of the art.

2.2.1 General Overview

Starting with low-level code analysis, static taint analysis tools are instrumental in identifying data leaks, tracing the flow of information from a source (point of data origin) to a sink (point of data egress or exit). Among these tools, *FlowDroid* [26, 27] stands out as one of the most effective static analysis tools for Android applications, particularly in the domain of privacy analysis. This can largely be attributed to the fact that the tool is actively maintained by its authors, with continual improvements to its detection methods and to the compatibility with new versions and methods. This superiority has also been demonstrated in a comparative study by Pauck et al. [28], where FlowDroid achieved higher compatibility, favorable true-to-false positive ratios, and strong results in taint analysis benchmarks. Since that study was conducted in 2018, and the tool has continued to receive active development, its performance can reasonably be expected to have improved even further. This tool is not alone in this category, however. DroidSafe [29] analyzes both Intents and Remote Procedure Call (RPC) calls, but lacks support for inter-app analysis. IccTA [30], later integrated into FlowDroid, focuses on tracking flows through Intents but does not cover RPC. And finally Amandroid [31], which was subsequently introduced to address the limitations of these earlier tools. These tools can equip us with useful information about the data flow of the app, which in turn provides visibility where and when the information is being used, however, just like every approach, it presents its own challenges. Not every behavior can be determined by simply looking at the code. Take, for example, dynamic linked libraries, dynamic user inputs, conditional clauses based on system states, or even multi-threading. In these cases, static analysis tools will have to adopt a best-effort approach, trying to “guess” what will happen in these situations, leading to increased false positives.

To overcome these challenges, dynamic taint analysis tools can also be employed. TaintDroid [32], one of the oldest among the list, is a dynamic analysis tool that performs taint analysis during the execution of the application. As mentioned in section 2.1.2.1, this will allow the tool to have access to runtime information, as well as tracking the information across threads, dynamically loaded code, and paths influenced by dynamic user input. After TaintDroid, multiple dynamic taint analysis tools were developed such as ARTist

[33] and TaintArt [34] which instrument the app using the, at the time, newly implemented Android Runtime (ART) that came to substitute the Dalvik Virtual machine (DVM) as the new default runtime; DroidScope [35] that combines taint analysis with system call monitoring and API tracing for deep behavioral analysis; CopperDroid [36] that performs taint propagation at the system-call level, abstracting it from the Virtual Machine (VM), making it unaffected to the change of runtime. And finally, ViaLin [37], which takes a different approach in the way that it stores the taint tags, allowing a wider tracking with the possibility to reconstruct the path of the leak. More dynamic taint analysis tools have been proposed and developed; however, we chose only a portion that we considered to be relevant based on supported Android versions, available documentation, and the last update of the tool.

Looking for a more high-level view, some works try to approach a similar problem in a slightly different way. DeepIntent [38] takes user expectancy and compares it with the actual application behavior to determine if the actions behind the UI correspond to what a normal user would expect to happen; Take, for example, an app with a globe icon (or pin icon usually associated with geolocation or map functionality) which leads to the request of the microphone permission. That would be suspicious since it would be unexpected from the user's point of view. This provides yet another approach to inferring purpose, or at least checks if it is logical within the application context; however, it will fail in cases where the permission misuse is related to the icon. Say, for example, a case where we have the same globe icon, which leads to the location information being transmitted for ads. Another example of slightly different approaches would be COSMOS [39], a tool which is also used by FlowIntent [40], covered in more detail in section 2.2.4. This tool uses foreground information about an app by collecting information about the sensitive APIs and widgets used, and associated app layouts, which lead to their execution. With this information, using a machine learning model, a prediction is then made to attempt to classify this behavior as legitimate, user-dependent, or illegitimate.

As we will also cover in section 2.2.3, using network information to extract the purpose, or information that may be used for that goal, is actually a very common approach. This is to be expected given that this is one of, if not the, most common protocol used for communication [7] in Android. It is easy to understand that most of these requests are made through the network layer; therefore, if we monitor the request leaving the device, we may be able to infer what the information is being used for. Apart from MobiPurpose, which will be covered in Section 2.2.3, some other approaches also consider this, such as

AGRIGENTO [41], which uses differential analysis to identify which requests contain sensitive data, and which sensitive data is being transmitted. Differential Analysis is a study into how different input information can affect the output information. For the specific case of AGRIGENTO, it runs the application multiple times and retrieves the network requests, after removing the sources of non-determinism such as timestamps and nonces. Following that, AGRIGENTO modifies the device's sensitive data (such as the location and contacts) and reruns the application to generate the network requests, checking for differences in the output which, if identified, indicate the transmission of that type of information. In this scenario, although we do not have access to the textual features of the request body, such as the naming used for the variables, which may help deduce the purpose, we still have a resilient way to identify the leakage of sensitive information and to where, assuming the URL provides enough information. This tool is also used in LeakDoctor [42], which took the network analysis a step forward; Instead of only identifying when information is leaving the device, it applied the idea of AGRIGENTO to check which responses are influenced by the sensitive data and from there, using Flowdroid, it checks if the request leads to any noticeable UI change and if the sensitive information was required for that change, i.e., if modifying the information that is sent leads to the same change, then the usage of the sensitive information is not needed, and therefore, not "acceptable". This provides yet another way to identify the usage of information; however, this approach does not take into consideration non-perceivable but essential tasks performed by applications such as Telemetry or Contact syncing, which, although they do not perform any noticeable UI change, are still critical for the core functionality of the application. This is further worsened by the fact that use cases like targeted advertisement would be considered as acceptable since most of them change based on the information (usually location) and result in a perceivable UI change.

There are also slight variations of this problem, such as NoMoADS [43], which detects and blocks ads, exclusively, from network communications. It does this by making use of EasyList [44] and AdBlockPlus [45] lists of advertisement domains, which it compares with the traffic captured with AntMonitor [46]. A later extension of NoMoADS was proposed, NoMoATS [47], which aims to extend this detection to requests associated with Advertisement and Tracking (A&T) services. To achieve this, NoMoATS redesigns the way traffic is collected by employing Frida [19] and LibRadar++ [48] to identify A&T packages involved in network requests, in which case it blocks them. TraceDroid [49] was another 2022 proposal to analyze traffic in Android applications and identify flows to third-party libraries.

It accomplishes this by hooking network APIs with Frida [19] and analyzing them to understand if they were triggered by a Third Party Libraries (TPL). If that is the case, and the request contains sensitive data, then TraceDroid flags that request. This approach is mostly focused on understanding which data gets collected to TPL rather than inferring the purpose of such collection. Kohli et al. [50] aims to improve the current state-of-the-art approaches on identifying Personally Identifiable Information (PII) in network requests, it does this by employing SBert [51] and Autoencoders. Although not focused on identifying purposes behind data harvesting practices, it provides yet another stepping stone towards that goal. Lastly, Trigger relationship aware mobile traffic classification (TRAC) [52] shifts its focus to incorporating metadata about what triggered the request. The idea is that a given request containing sensitive information can be classified differently depending on who initiated the request. Looking at per-app network traffic may seem enough; however, in some instances, a request from application A can be initiated by application B. Let's say that B sends an intent to A to perform a certain task, for example, assuming A to be a social media app, B can perform an intent to A to share a certain image on the user's page/feed, something which is common in most social media applications. TRAC considers this by constructing what it calls the Trigger Relationship Graph, a graph that represents the relationships between applications and the trigger events between them, i.e., which applications trigger which applications for which requests. With this information, TRAC employs the Application Label Corrector to update the original classification using the new information, improving the classification model underneath.

Still in the realm of network analysis, Lalaine [53] is a proposal for iOS, based on the App Store privacy labels provided for the apps. Privacy labels are essentially tags indicating which type of information is being collected by each app. These labels span from Contact Info, Identifiers, and User Content to Search History and Usage Data. This approach focuses on identifying network data flows containing sensitive data, retrieved from iOS APIs, and comparing those data flows to what would be expected from the app's privacy labels. Similarly, PurPliance [54] identifies inconsistencies in network requests by comparing them with the privacy policies of the application, resorting to the classification from MobiPurpose [55], work which will be covered in greater detail in section 2.2.3, with purposes from the policy, extracted with PoliCheck [56], PolicyLint [57], and Polisis [58]. Unfortunately, purposes in privacy policies are extremely limited beyond "core functionality" and third-party services. OVRseen [59] also considers the privacy policies

when classifying network request purposes in VR systems and apps. It does this by employing Frida [19], AntMonitor [46], PoliCheck [56], PolicyLint [57], and Polisis [58]. By identifying PII in the network requests, and looking at the network requests and privacy policies, OVRseen classifies the request with a purpose, which, similarly to the previous 2 approaches, mostly focuses on Advertisement and Tracking/Analytics (i.e. the other labels are things like “basic feature”, “additional feature” and “security”).

Moving away from the privacy policies, we also found some work in the field of OS privacy. These works study data collection practices from the OS itself and not from applications in the device. The main goal in these studies is to understand what data the vendors are collecting and to determine if these harvesting practices are legitimate or fall far beyond what is considered necessary. Although it has a different target when compared to the previous works reviewed, OS Privacy analysis is, in its core, a purpose classification problem, but targeting system-level network traffic. Works which pursue this goal are Liu [60], which employees Man-in-the-Middle (MitM) attacks, Frida [19], mitmdump [61], among other tools, to detect PII in network requests performed by the device itself, not only associated with a given application. This work aims to identify data collection practices in different geographic regions. Something similar is done in Liu et al. [62], which performs a similar analysis, however, now shifting the focus into the data harvesting practices between different vendors, namely Samsung, Xiaomi, Huawei, and Realme Handsets. It employs similar tools as the previous one, mitmproxy [61], Frida [19], Edxposed [63], and Riru [64], but once again, it is more focused on OS-based network request analysis.

Certain approaches focus on the application’s description, as it often indicates what the app is designed to do. From this information, it is possible to anticipate which types of sensitive data the app may need to access. Trying to improve on the previous approaches of description-based analysis, such as AutoCog [65], Feichtner and Gruber [66] proposed a different approach to mitigate some of the shortcomings. To accomplish that, the authors propose the use of a Convolutional neural network (CNN) to link the permission use to the app description and a new embedding model to capture semantic links from the app description. This new approach can still effectively work with unlabeled and potentially flawed application descriptions. Using the description to understand what permissions the application should be using provides a solid base; however, the application description may not declare the complete application functionality, which may lead to some false negatives. Nonetheless, it can still be used in a complementary way with other approaches to understand the purpose of sensitive information flows.

Finally, there is also research done in the context of MiniApps, which are a hybrid application format that incorporates web technologies such as CSS and JavaScript with capabilities of native apps [67]. This works by having a **super app** acting as a host for the other applications, the MiniApps, enabling their execution. This removes the need for installation since, like web pages, it only needs to load the content, centralizing everything in one app; Instead of needing to install a transportation app, it can be loaded onto the super app, if approved, and used. The most popular examples of these apps come from China with Baidu Smart and AliPay. TaintMini [68] proposed a static analysis approach to Mini-programs (or MiniApps) to detect flows of sensitive data. This paved the way to the analysis of MiniApps. However, relying only on static taint analysis leaves many flows undetected. Dynamically loaded packages are examined, and sensitive communication between mini-apps may go unnoticed. This is a significant issue, since in many cases, one mini-app with access to sensitive information will share that information with another mini-app without those permissions. To address this, MiniScope [69] was later proposed, based on a two-phased hybrid approach. In the first phase, MiniScope would use automated UI exploration to load all packages. In mini-apps, packages are loaded on a need-basis, and generate a complete application control flow graph, and then, in the second phase, analyze said graph to understand the privacy practices of the mini-apps.

2.2.2 Inter-Work Comparison Limitations

Although extensive research has been conducted on mobile device privacy, and considerable progress has been made in malware detection, the specific problem of purpose classification remains comparatively underexplored. Multiple approaches aim to perform this task; however, each one of these works sets different goals. Some approaches are simply interested in the behavior of third party SDK and their data leakage, some approaches want to classify the purpose for the application as a whole, others classify network request by network request, one classifies key-value by key-value found inside of each network request sent, and most importantly, almost every single one proposes its own taxonomy for this classification. Some taxonomies are less specific, such as *Core Functionality* and *Non Core Functionality*, while others perform fine-grained classification, such as *Map Navigation* and *Network Optimization*, some do not even care about fine-grained purpose classification, opting instead for understanding which SDK service gets the data. This creates a considerable discrepancy in how to compare models.

MobiPurpose compares itself with Text Mining by Wang et al. [70, 71], works which will be discussed in more detail in Sections 2.2.3 and 2.2.5, however, the latter performs the classification per leak, as in data flow leaks, and using textual features, not by network requests. The way these two approaches are compared is simply by looking at the average score of MobiPurpose's models and comparing it with the average score of Text Mining by Wang et al. [70, 71]. This does not accurately demonstrate the actual benefits and drawbacks of each approach. For example, Text Mining by Wang et al. [70, 71] depends on TaintDroid [32] to get the leaks, which, since its latest supported version is Android 4.3, is not a very realistic tool to use today and relies on the chance that the app is not completely obfuscated to extract meaningful tokens. On the other hand, MobiPurpose fails when the information in the request body is not enough or when the URL does not contain sufficient information about who is getting the information. Those are problems exclusive to each approach, and thus, when they conclude that the performance between the two approaches is similar, since the model scores are somewhat the same, a lot of nuance and context is lost. Within these approaches, there are also differences in the granularity of the data being monitored. Some works like Djob and Bromberg [72] focus more on the identification of PII leakage, not necessarily data acquired from permission requests or device information. As is the case with works from Wang et al. [42] and Xiao et al. [53]. This problem is not exclusive to MobiPurpose, as any comparison with another approach, even only within the realm of network-based analysis, will need to consider the different system designs, classification point (where the actual classification is made, e.g., app-wise or per request), as well as the different taxonomies and their granularity, making the comparison somewhat unreliable/unfruitful.

To that end, we looked at recent works whose goal aligns, in some capacity, with ours and whose focus falls primarily on network-based analysis, to understand this discrepancy. It is important to note that this table is not an extensive characterization of these papers; instead, it focuses on providing some more information about these works, namely the data sources and their ultimate goal, while keeping it concise. These goals are then represented in Table 2.1 through the use of ● indicating the main focus of the work, i.e., this work is mostly aiming to perform/detect/identify this operation; ◐ indicates that the work, while trying to achieve its goal, uses or incorporates this information/feature; Finally, ○ indicates that the work does not make use of that feature/process. Furthermore, we also collected tools used across these papers, as shown in Table 2.2, to provide an overview

of what tools and methodologies are commonly applied in the process of purpose classification. This table allows us to have a better understanding of the problem of comparison in this area. As mentioned previously, each of the works will have a slightly different end goal, which makes the comparison either extremely hard to perform or completely impossible, such as in cases where the environment is different (Android, iOS, Virtual Reality (VR), etc.).

If we look at Table 2.1, starting with the first column (Advertisement and Tracking Detection), we will see works which either aim, almost exclusively, at detecting ATS or incorporate that detection in their global goal, those which want to identify more than Ads and Tracking. If we take TraceDroid [49], Lalaine [53], Text Mining by Wang et al. [71], Safeguarding User Security and Privacy [60], OVRseen [59], PurPliance [54], MobiPurpose [55], all of these works, in some way or another, incorporate the detection of ads and tracking services in their main goal; MobiPurpose provides a more general purpose inference model, which includes identifying ads and tracking, but not solely focused on that; Another example is Lalaine [53], which although focused on identifying inconsistencies between Privacy Policies and actual data harvesting practices, it is interested in ads and tracking detection since that is a very significant point when it comes to privacy policies and data usage policies. On the other hand, we have works like NoMoADS [43], NoMoATS [47] (extension of NoMoADS), and DiffAudit [73] whose focus rests entirely on detecting Ads and tracking services, although NoMoADS is only concerned about ads.

The second column (TPL Behavior) covers the goal of identifying TPL data harvesting practices and the use of TPL presence/behavior as features to accomplish another end goal. The only work that is entirely focused on TPL behavior is TraceDroid [49]. TraceDroid looks at Android applications and tries to understand if sensitive data is being sent to a TPL or not, and which data is being sent. On the other side, NoMoATS [47] and DiffAudit [73] use the presence/use/traces of a TPL in these apps to accomplish their overarching goal.

The third column (OS Privacy Analysis) involves a more specific category of works, those interested not in application behavior, but rather the whole Operating System. Due to the nature of this problem, all works either have it as their main focus or do not implement/use it at all. The only works which do look at OS-wise analysis are Safeguarding User Security and Privacy [60] and Android Mobile OS Snooping [62], whose goals are essentially to understand which information the actual OS is sending out, to where, if it makes sense,

and how it changes based on different vendors and/or regions.

The fourth column (PII Detection) covers work done towards detecting PII in network requests or the use of their presence as a feature in other works. Similar to the previous column, only End-to-End PII Detection [50] is exclusively focused on PII detection. The rest of the works present in this column use the PII presence in the requests as a feature, usually to identify which data types are being collected. It is also important to note that, this column only covers works which look at PII as a whole; MobiPurpose for example looks at sensitive information which can be retrieve from the device, usually associated with the Android native APIs such as Location, Contacts, Camera, among others; It does not cover information such as name, address, date of birth, etc., and for that reason it does not qualify for this column.

In the fifth column (Policy Validation), we have the works which look at validating, in some way, the privacy policies and equivalents, for Android/iOS/ VR applications. In this column, we face the same situation of works either fully addressing this problem, or not relying on it at all. Works like Lalaine [53] and PurPliance [54] look at detecting inconsistencies between the statements in the privacy policies and the actual application data harvesting practices, while OVRseen [59] focuses on detecting data flows that were not disclosed in the privacy policies.

Finally, and arguably the central point of discussion for this work, indicates the objective of inferring the purpose of a network request for an application or OS. However, there are some nuances which need to be considered here; For this column, being assigned ● means that the work is mainly focused on classifying purpose without really focusing on categories or specific areas. As an example, MobiPurpose [55] attempts to classify network requests with a significant level of granularity on areas such as map navigation, interface customization, authentication, etc., while works like OVRseen [59], although classifying the purpose, their taxonomy is more policy oriented and less granular with things like ”Advertisement”, ”Tracking”, ”Additional Feature”, ”Basic Feature”, ”Marketing”, and so on. Both are classifying the purpose of the request, but the former is doing so with more granularity and mainly focused on obtaining that classification, while the other classifies the purpose for another goal and with a more specific and less granular taxonomy. In reality, in this list, only the 3 works which will be covered in sections 2.2.3, 2.2.4 and 2.2.5, and our work, are fully focused in the general classification of purpose, while the others

use it as a tool/feature/means to achieve something else, usually to compare those results with policies, perform a OS-wise privacy analysis or identify data harvesting for ATS.

Table 2.1: Goal Summary of related work

Work	Data Sources	Advertisement and Tracking Detection	TPL Behavior	OS Privacy Analysis	PII Detection	Policy Validation	Purpose Inferrence
DiffAudit [73]	Network Requests	●	◐	○	◐	○	◐
NoMoATS [47]	Network Requests and APK/App Analysis	●	◐	○	○	○	○
NoMoADS [43]	Network Requests and Advertisement filter rules	●	○	○	○	○	○
TraceDroid [49]	Network Requests, Files, APK/App Analysis	◐	●	○	◐	○	○
Lalaine [53]	APK/App Analysis, Privacy Policy, Network requests	◐	◐	○	○	●	◐
Text Mining by Wang et al. [70, 71]	APK/App Analysis	◐	◐	○	○	◐	●
Safeguarding User Security and Privacy [60]	Network Requests and Device/User data	◐	○	●	◐	○	◐
OVRseen [59]	Network Requests and Privacy Policies	◐	○	○	○	●	◐
PurPliance [54]	Network Requests and Privacy Policies	◐	○	○	○	●	◐
MobiPurpose [55]	Network Requests, Application Metadata	◐	○	○	○	○	●
Android Mobile OS Snooping [62]	Network Requests and Device/User data	○	○	●	○	○	◐
End-to-End PII detection [50]	Network Requests	○	○	○	●	○	○
FlowIntent [40]	Network requests, APK/App Analysis and Device/User Data	○		○	○	○	●
Our Work (LeakHound)	Network Requests, APK/App Analysis	◐	◐	○	○	○	●

● = Main Focus
◐ = Implementation/Usage
○ = No Coverage

Table 2.2: Overview of tools used in network-related work

Tool	Works resorting to Tool	Used For
Frida [19]	Lalaine [53], OVRseen [59], NoMoATS [47], Safeguarding User Security and Privacy [60], TraceDroid [49], DiffAudit [73], Android Mobile OS Snooping [62]	Network Analysis, Stack Trace Analysis, SSL-Pinning Bypass, Dynamic data retrieval
PCAPdroid [74]	DiffAudit [73]	VPN, MitM Proxy bridge or server
Wireshark [75]	DiffAudit [73]	Network Analysis
PolicyLint [57]	OVRseen [59], PurPliance [54]	Identification of contradictions in Privacy Policies
PoliCheck [56]	OVRseen [59], PurPliance [54]	Data flow to policy consistency checker
Polisis [58]	OVRseen [59]	Policy Analyzer which provides shortened representations of the policy
MobiPurpose [55]	PurPliance [54]	Network-based purpose inference tool
PurPliance [54]	Lalaine [53]	Used to extract purpose from privacy policies
mitmproxy and mitmdump [61]	Safeguarding User Security and Privacy [60], Android Mobile OS Snooping [62]	MitM attack proxy, Network Analysis
Edxposed [63] and Riru [64]	Android Mobile OS Snooping [62]	Android Framework Hooking
Spark [76]	TRAC [52]	Framework for Points-to analysis for Java
MonkeyRunner [77]	TRAC [52]	UI Exerciser
LibRadar [78] and LibRadar++ [48]	NoMoATS [47]	TPL detection tool
AntMonitor [46]	NoMoADS [43], OVRseen [59]	Network Monitoring
EasyList [44] and AdblockPlus [45]	NoMoAds [43]	ATS filter rules
SBert [51]	End-to-End PII Detection [50]	Transformer Model
l-mapping [53]	Lalaine [53]	Mapping between l-data (sensitive data) and the APIs which provide access to them
nosmoke [79]	Lalaine [53]	Checks if l-data is being sent out in network requests
Fiddler [80]	Lalaine [53]	Web debugging proxy tool
DroidBot [81]	MobiPurpose [55]	UI exerciser
TaintDroid [32]	Text Mining by [70, 71], FlowIntent [40]	Dynamic Taint Analysis
FlowDroid [26]	FlowIntent [40], COSMOS [39], LeakDoctor [42]	Static Taint Analysis

This analysis is done to help understand which works can be compared with one another, while also providing a concise and general summary of these works. In theory, it is possible to perform partial comparisons between the works. Take, for example, MobiPurpose and NoMoATS; Given that MobiPurpose's taxonomy includes Advertisement and Tracking, it is possible to directly look only at those classes to see how they compare. However, this is not a fair comparison as MobiPurpose's model is not only trying to classify Advertisement and Tracking, but also Map Navigation, Interface Customization, among others, and thus sampling classes from a more general model and comparing it with the results of a model whose entire focus are those classes, would provide little to no insight onto the actual performance differences between the two models. Moreover, the problem is not only

with the goal of the work, but also in the taxonomy used. MobiPurpose compares itself with Wang et al. [70, 71] and concludes that both have an equal performance, given that both works achieve a similar score. This is a very dangerous statement as Wang et al. [70, 71] employs a more fine-grained/specific taxonomy in its classification. Thus, by achieving a similar score, one could argue that the two works are not in fact similar, but Wang et al. [70, 71] is outperforming MobiPurpose, since it is achieving the same performance for a more fine-grained classification (has to generalize for more niche/rare/specific occurrences/purposes). The same would apply for a comparison with FlowIntent. The latter is interested in functional and non-functional usages, so a comparison could be possible if either FlowIntent was trained to now classify for the same taxonomy of MobiPurpose or if the taxonomy of MobiPurpose was collapsed into 2 generic groups of functional and non-functional. However, for the latter option, this comparison, although possible, ignores the fact that MobiPurpose is not concerned with whether a given purpose is considered functional or not for that application, but instead with the purpose of the data collection itself. That leaves us with only the possibility of expanding FlowIntent's taxonomy to MobiPurpose's, which, admittedly, is possible, but will be quite limited as FlowIntent was first, not built to be as specific as to the actual purpose, but rather its legitimacy, and second, all background-related activities would not be detected. This only helps to demonstrate the challenge of performing inter-work comparison for works in this field, and as such, most approaches rely on human expertise to verify the performance of their model, usually by either creating the dataset from scratch and labeling each entry manually by multiple experts, or by validating the results of the model after the classification using a pre-existing dataset. But even these datasets are rare to come across. Since we are looking at the purpose behind sensitive data collection, sensitive data will be present in the dataset, thus creating another barrier to its publishing. The few datasets that are available for purpose classification are usually generated through the use of the tool the author proposes, just as in the case of MobiPurpose, meaning that they contain a lot of noise, derived from the tool's mistakes. This is one of the reasons why depending on these datasets can provide more drawbacks than advantages; however, we will touch on this matter again later in this Thesis. In conclusion, although different works have been proposed in this field, the comparison with those works is usually unreliable, impractical, or unfair due to different purpose classification taxonomies, lack of datasets, different end goals, and OS.

With this issue in mind, we will now discuss 3 works we selected which we found to be representative of the diversity of approaches for the purpose inference in mobile devices

and whose goal directly aligns with what we are trying to achieve. Opposite to the previous review, these works will be covered with greater detail to provide a better insight into the methodologies employed by state-of-the-art works.

2.2.3 MobiPurpose

MobiPurpose [55] is a network-based proposal that aims to infer the purpose of permission requests through the network traffic generated by the target application. To achieve this, it performs a on-device MitM attack to strip away the SSL/TLS layer. This allows interception of all outbound HTTP(S) traffic, where the request bodies are parsed into Key-Value Pairs (KVP), these pairs are then run through a combination of Machine Learning (ML) and Natural Language Processing (NLP). The workflow can be seen in Fig. 2.1.

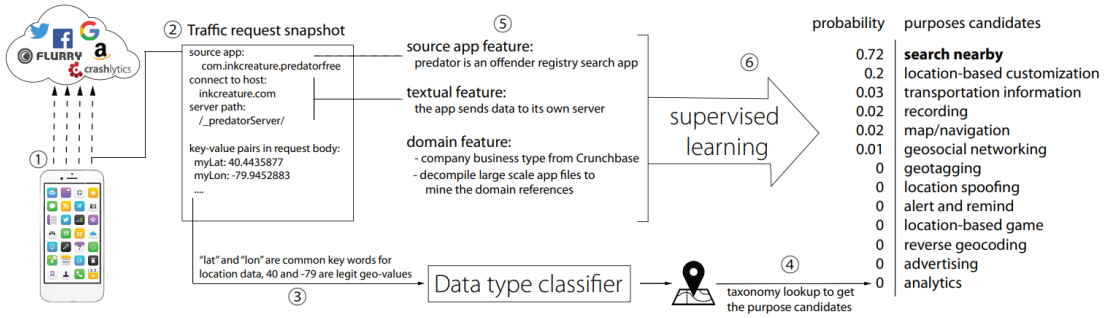


Figure 2.1: MobiPurpose Workflow - Extracted from [55]

In essence, Mobipurpose will run the application **(1)** and take snapshots of the requests being sent out of the device **(2)**. With the requests collected, the first problem to tackle is the **what** type of information is being sent out. The source of this information is the Key-Value Pairs (KVP) mentioned previously. Essentially, it will parse the body and queries of the HTTP request and retrieve all KVP it finds, e.g. “myLat: 40.7”. These key-value pairs are then sent to a Data Type classifier **(3)** with a two-step pipeline. First, it checks for special strings like IP, Email and Universally unique identifier (UUID), through the use of regular expressions. If the pair is not a special string, then it is parsed using a bag-of-words based on the English WordSegmentation model [82], which was then expanded to include common terms and abbreviations of the field such as “ad” for “advertisement”. Then, using a bootstrapping iteration with Bayesian classification, MobiPurpose classifies a KVP as their data type. With the data type defined, we can select the purpose candidates **(4)**, which are all the possible purposes from the purpose classification taxonomy, for that data type, as location can be “Search Nearby” but not “Authentication”. It then collects

several features to infer the purpose (the **why**) **(5)** for the data collection. These features, which can be seen in the Table. 2.3, are divided into 3 categories.

Group	Feature	Details
Embedded textual features	G1: *url path bag-of-words*	A bag-of-words representation of the URL path and sent data (160∼350 dimensions)
	G2: *package-endpoint similarity*	A 3 dimension vector, each value represents if the URL components share a common string with the package name.
	G3: *co-sent data types*	The length of k-v pairs and involved data types.
Source app features	G4: *app category*	The app category can be queried from Google Play using the package name (15 binary dimensions).
Domain features	G5: *owner business type*	The business categories from Crunchbase.
	G6: *domain occurrences in app corpus*	We first decompile the app file, then find the apps that contain the domain inside their source code, and count the app distribution across different app categories. We use the top app categories to represent the business type.

Table 2.3: MobiPurpose features for purpose classification - Extracted from [55]

First, there are embedded textual features, which analyze the URL to access the purpose, like the presence of the word “ad” in the URL, e.g. “ads.google.com”. The URL is also compared with the package of the app for a similarity check. This way, the tool can detect, or at least infer with higher probability, whether the endpoint belongs to a third party. Finally, they also analyze the size/length of the KVP, under the assumption that advertisement providers may sometimes require screen sizes, or analytic services may be more aggressive in their data collection. These features represent both the number of total key-value pairs as well as the number of different data types in the key-value pairs.

For the second group of features, MobiPurpose collects the app category, be that “Gaming”, ”Social Media”, etc, from the Google Play store. Finally, the third group will look for information about the service provider using both the WHOIS API and Crunchbase API, as well as analyzing the app’s source code, extracting domains, and looking for their presence in other apps. After doing this, they store the different app categories that reference this domain. Furthermore, if the domain is contacted by more than 10 apps across more than 5 categories, then the business type is defined as ”third-party library”; otherwise, the top 3 app categories collected represent the business type. To avoid overfitting of the model, after applying the bag-of-words model, the features with a frequency below 3% are dropped.

For the aforementioned purpose taxonomy, which can be found in Tables A.1, A.2, A.3 and A.4 in the Annex of this Thesis, the authors first defined a hierarchical-structured taxonomy on which to label the leaks. This taxonomy was developed based on an initial review of existing taxonomies, complemented by an in-house workshop where participants proposed categorizations and then engaged in group discussions to decide which

categories should be modified, removed, or retained.

With the foundation built and all data collected, Mobipurpose trains one model per data type, meaning that, for each KVP, where a data type was identified, a different classification model will be trained. This comes from the idea that a request can have multiple purposes, and for that end, some KVP might be associated with one of those purposes. In this situation, each KVP will have one purpose only, but the combination of all pairs will provide the set of purposes for the network request where these pairs reside. By separating this classification, they train a model for each data type, only for the purposes that data type can take. For example, as per Table A.4 Location can have purposes such as “Nearby Search”, “Geotagging” and “Alert and Remind” but not “Voice Authentication” and “Audio Streaming” which are exclusive to Microphone. MobiPurpose tested multiple models to classify the purpose; however, the one with the best performance was Maximum Entropy (ME) with an overall accuracy of 84% across all data types.

MobiPurpose achieves a respectable accuracy of 84% (macro-average) when labeling the permission purposes, and therefore presents a lot of potential for the purpose it was designed for. The foundation for network-focused analysis is based on the fact that, if a malicious developer intends to steal/misuse/sell the sensitive information, then this information will eventually need to leave the device. Setting aside overly engineered approaches, such as the use of covert channels, multi-app leaks, and other stealthy operations, most of the leaks will happen through the network channel [7, 8], and therefore, monitoring this exit path will provide very valuable information for the purpose inference. However, this approach is not without its drawbacks. The first and most obvious problem is the use of obfuscation within the request body. The obfuscation mentioned is not the result of the encryption used based on the HTTPS protocol, since a man-in-the-middle attack is used in order to strip the SSL/TLS layer of the request. The applications may employ their own obfuscation on the content before sending it to the network layer. If this happens, all the information inside the request becomes unusable, and therefore, no informed predictions can be made. Another potential problem is the use of uncommon identifiers for the information in the request body. Previously, it was given the example of ”myLat:12.01; myLon:-45.23” which clearly depicts location information, but if, for some reason, the developers went with random, meaningless terms such as ”hajsk:12.01; 8kjAN:-45.23” it would be considerably harder, not to say impossible, to identify the type of information. Note that in this case, the format of the data (coordinates) could be used to identify the data type; however, minor adjustments, such as geo-hashing or encoding

the coordinates, would be enough to prevent that.

2.2.4 FlowIntent

FlowIntent [40] aims to identify non-functional sensitive transmissions in mobile applications. To accomplish this, FlowIntent implements 2 modules, as shown in Fig. 2.2.

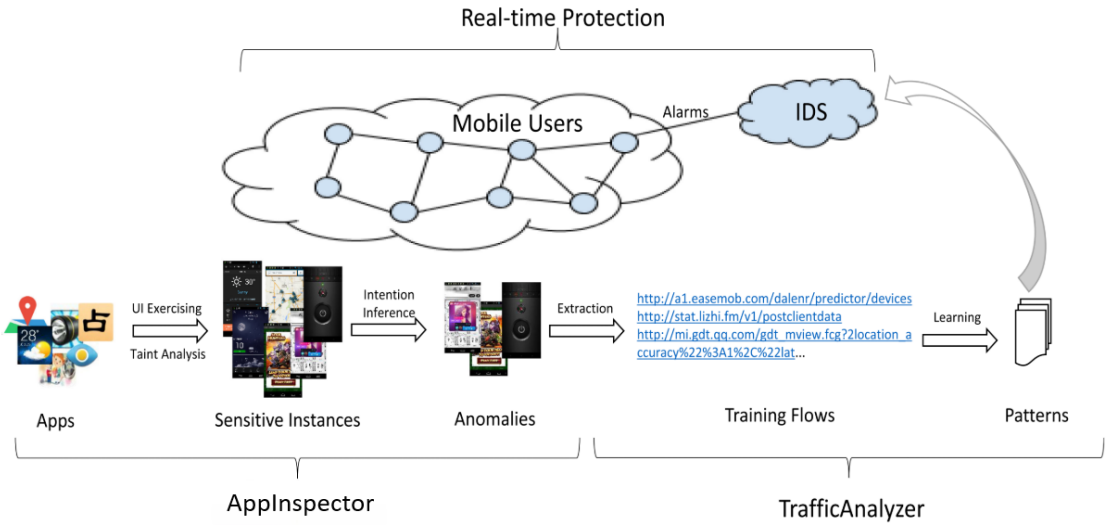


Figure 2.2: FlowIntent system architecture - Extracted from [40]

The first module, AppInspector, will run the application inside a sandbox. This sandbox will have TaintDroid [32], TCPdump[1], and COSMOS [39]. COSMOS, as mentioned previously, is a tool developed by the same authors that will analyze the application foreground information, such as app layout, widgets, and APIs used. When the app is placed inside this sandbox, COSMOS will trigger the sensitive activities which detected beforehand with FlowDroid [26], activities which it deemed sensitive due to a possible information leak, and once this activity is triggered, if taintdroid detects any leak then the sandbox will tcpdump all the requests for 60 seconds and COSMOS will take a screenshot of the app's current state. To process this information, COSMOS will leverage Optical Character Recognition (OCR) to extract the embedded text and store the information into an XML. This data will also be combined with the TCP packets stored by tcpdump, and together, it forms what the authors call "app context". To label the app context, the authors defined that, for a given resource r, the context is legitimate if there is at least one functionality that requires r to be sent out of the device by t, where t represents the sensitive transmission. For this, FlowIntent uses some different sources of information. First, since the name cannot always be used to understand the purpose of the app, take, for example, "Uber" vs "LocalWeather",

[1] https://www.tcpdump.org/

the app description from the app store is also added as another source of information. Furthermore, since sometimes even the app description can be more abstract to the app's inner workings, the authors further collect user interfaces that are especially useful when (1) the app has multiple functionalities that are valid under different contexts, and (2) some functionalities provided are not clearly mentioned in the app description. After labeling the app context, Flowintent transitions to its second module, TrafficAnalyzer. In this module, 2 processes are applied. The first one is a supervised ML model that summarizes the patterns for non-functional sensitive flows and functional flows. To find these patterns, they collect 2 types of features: Statistical features, where, for each HTTP flow with the same source IP, port, and destination IP, it retrieves the total number of TCP packets, uplink TCP packets, HTTP packets, packet size, uplink TCP packet size, downlink TCP packet size, and time interval between TCP packets. The second type of features are the lexical features that result from the analysis of the URL. Specifically, collect the binary feature for each token in the hostname and in the path URL, the length of the hostname and of the entire URL, and the number of dots in the URL. With these two categories, a supervised learning model such as logistic regression is used to construct a classifier that is able to identify the validity of sensitive transmissions.
The second process makes use of clustering to generate network-level signatures, i.e., network patterns representative of a given behavior. As an oversimplified example, the presence of "UNION", "SELECT", and "– ;" in network requests is a signature (pattern) of SQLi. The idea behind this is to create signatures that can later be used by a Network Intrusion Detection System (NIDS) for network-level detection.

FlowIntent achieves a respectable 94.1% F-Measure, where the lexical features provided a greater influence on the results when compared with the statistical features (93.1% vs 76.9%), highlighting the importance of textual/lexical-based features. However, this approach has its nuances; The first problem is that, by taking a screenshot of the app's current state, we are basing the analysis on the current user perspective. This is fine for things such as ads, which present a graphical component but for situations where the information is used in a non-perceptible way, for example, background contact syncing on WhatsApp, it may lead to some incorrect labeling of the flow. Furthermore, we also run into the risk of taking a screenshot of the application's current state before the changes occurred or after the valid context changed. The second potential problem is that, since COSMOS is used in order to trigger the sensitive flows, by launching the activities that it

deemed as containing the source of the flow, if not all activities were tagged with sensitive flows by the static taint analysis, then not all flows might be considered. For example, a flow in activity X that was not detected by FlowDroid, which is the static taint analysis tool used in COSMOS, but would be detected by TaintDroid. In this case, the sensitive flow would not be considered. Another issue revolves around gathering the network flows. As stated above, once taintdroid detects a flow to a network sink, the sandbox dumps all TCP packets sent during 60 seconds. This represents a problem since a lot of non-related packets, such as resource retrieval, data sync, configuration checking, preloading, and caching, will also be caught. These requests will inadvertently influence the quality of the features extracted by introducing noise. Finally, FlowIntent only analyzes foreground information, not considering any background practices/information. This means that use cases like background syncing will not be covered by FlowIntent, losing a significant amount of coverage.

2.2.5 Text mining approach by Wang et al. [70]

Wang et al. [70, 71] observe that, if an Application is not completely obfuscated, then textual identifiers can be extracted from custom code, specifically, class names, method names, and field names. For example, if keywords like “near” and “restaurant” appear together within a Location permission, this will usually indicate a “Search Nearby Places” purpose. This approach was built on top of their previous work Wang et al. [83], which determines where in the decompiled code sensitive permissions are used. Another important note is that this work focuses exclusively on Location and Contact permissions since, as the authors claim, these are the permissions users are most concerned about. In terms of architecture, the proposed technique can be divided into 2 different modules: static and dynamic. For the static module, as can be seen in Fig. 2.3, apps are first decompiled, and the sensitive segments of code, segments which contain sensitive APIs related to Location and Contacts, are identified. From here, they perform text mining, where identifiers and meaningful text features from the code are extracted, and static analysis to acquire the important/sensitive APIs, Intents, and Content Providers, as well as their frequency in the code. A full list of the features extracted can be seen in the Table. 2.4. After acquiring the information, namely the text-based ones, the TF-IDF [84] vector is calculated against a corpus of 2000 applications, and subsequently used as features. They then pass these features to a classification model, in this case a Support Vector Machine (SVM) with linear kernel and parameter C set as 1. It’s also relevant to mention

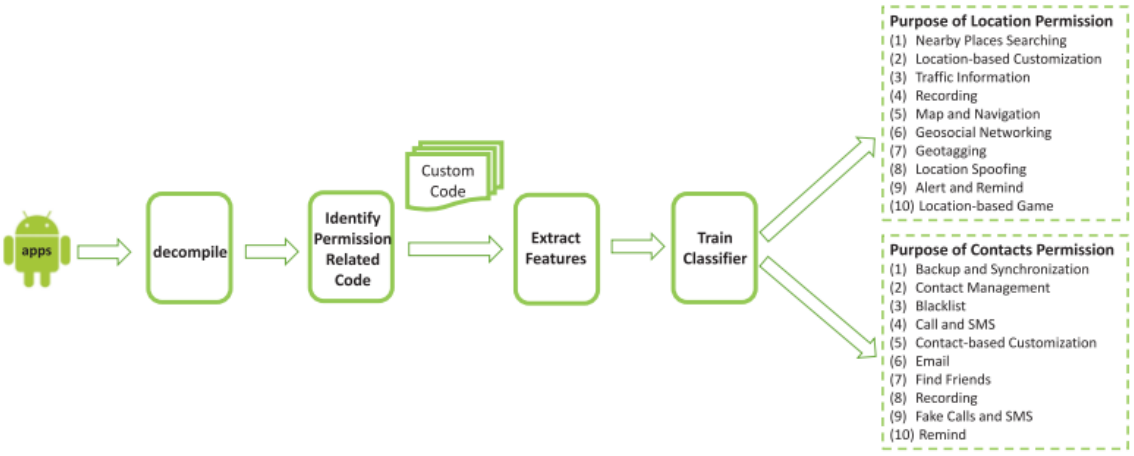


Figure 2.3: Text mining tool, Static Approach - Extracted from [70, 71]

Type	Kind	Features Description	Counts	Method
App Specific Features	Android API	Call frequency of each permission-related API	A 680 dimension vector, each value represents the number of occurrences of corresponding API.	Static Analysis
	Android Intent	Call frequency of each permission-related Intent	A 97 dimension vector, each value represents the number of occurrences of corresponding Intent	
	Content Provider	Call frequency of each permission-related Content Provider Uri	A 78 dimension vector, each value represents the number of occurrences of corresponding Content Provider	
Text-based Features	Package-level Features	Key words extracted from current package names	Calculate TF-IDF for all the key words, with each instance represented as a TF-IDF vector	Text Mining
	Class-level Features	Key words extracted from class and interface names		
	Method-level Features	Key words extracted from defined and used method and parameter names		
	Variable-level Features	Key words extracted from defined and used variable names		

Table 2.4: Text mining tool, extracted features - Extracted from [70, 71]

the granularity with which this approach was implemented. The authors state that classes present within the same directory/folder/package will usually have a cohesive, main objective. For example, classes inside the UI package will mostly deal with UI logic while the $repo$ package will deal with database control and operation. Due to this observation, all the features are collected from the same folder, excluding sub-folders, which are dealt with as individual packages themselves. To address the gaps in the static analysis, such as indirect sensitive data invocation (where one component of the app accesses sensitive data and stores it, and later other components retrieve that data from the first component), the researchers also designed a runtime module. This module takes into consideration the flow of the program, making use of a modified version of TaintDroid. The overall structure can be seen in Fig. 2.4. For the dynamic approach, an App Profile is created. This profile contains both the third-party libraries used in the application, information retrieved by running LibRadar [78] on the application beforehand, as well as textual features from the text mining process, which are shown in Table 2.4. Once a leak is detected, the sink

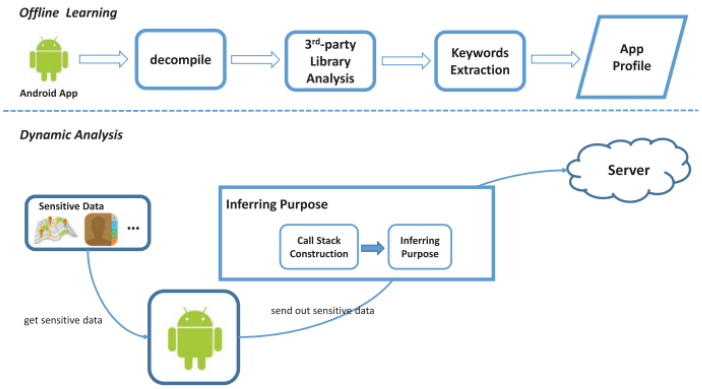


Figure 2.4: Text mining tool, dynamic approach - Extracted from [70, 71]

is analyzed and if it belongs to a known third-party library, then its purpose is labeled as belonging to the third party in question, such as AdMob and Firebase, which are previously labeled with a purpose. If the sink does not belong to a previously known third-party library, then the textual features of the classes to which the methods present on the leak belong are used to infer the purpose of the leak.

Wang et al. [70, 71] achieves a very good performance when classifying the purpose, with an accuracy of 94.73% for non-obfuscated applications and around 80% and 90% when including the obfuscated applications. However, this approach is not without its downfalls. Using textual features to infer the purpose of permissions has a lot of potential, given that this information can provide a better insight into the application's structure and the developer's intentions. However, there are a couple of issues with using such an approach. The first one is the use of obfuscation, which is a recurring problem as we also covered in MobiPurpose [55]. As stated by the authors, the percentage of apps that employ enough obfuscation to render this approach useless is relatively low, and therefore this technique can still be used for the majority of the applications. This is somehow supplemented by newer research [85–88] which claims that most of the obfuscation found in benign applications belongs to third party code, while the obfuscation applied to custom code is, in multiple occasions, "not on purpose" meaning that the obfuscation was not applied as recommended by the tools such as ProGuard [89] and R8 [90]. Nevertheless, as these tools become inherently more common, such as R8 being a default tool of Android Studio [91], the amount of obfuscation and its impact on the analysis will only grow. Furthermore, a problem that seems not to be addressed in this paper is the ProGuard/R8 feature of package flattening, which completely scrambles the package layout of the app, removing any

information about folder structure. Another, more critical flaw of such an approach is the possibility of an attacker manipulating the textual features to trick the module. Take, for example, a malware application that aims to track the user. To avoid suspicion from the analysis, the malware developer can name the methods and classes in a way that they present themselves as harmless/essential for the app's core functionality. The app can mask itself as a weather app and use names for methods such as "getWeather", "weatherResponseHandler", in an effort to make the analyzer believe that it is dealing with a weather retrieval instead of tracking. This problem is mentioned by the authors, and as they claim, it is not immediately clear how to solve it. Also, by using LibRadar to identify the use of third-party SDK, it provides a great insight into the purpose; however, solely classifying the purpose based on this information is a big assumption which can lead to false positives, as LibRadar will make mistakes and some SDK have multiple purposes. Finally, this approach relies on the use of TaintDroid [32], whose latest supported Android version is 4.3. This limits any application analyzed to be compatible with this Android version, which is unrealistic given that this version was released back in 2013.

2.3. Limitations

Current proposals provide strong inference capabilities for purpose identification in the Android ecosystem; however, all of them present their own limitations. In this section, those limitations will be covered; however, it's important to note that not all of these problems can be addressed within the scope of this work. Network-based approaches provide a good understanding of "who" will get the data, and therefore a strong basis for the identification of the purpose. However, these approaches are highly dependent on the ability to identify what type of information is being sent through the request. MobiPurpose tackles this problem by analyzing the request body and using NLP to look for keywords which could be used to identify which type(s) of sensitive data is being transmitted. This is a fair approach; however, it fails in situations where the request is obfuscated. Furthermore, in some instances, as aforementioned, the URL may not be enough to clearly identify the purpose. This can happen for multiple reasons, such as intentional obfuscation of the URL, ambiguous/general term usage for the construction of the URL, and usage of IP addresses without a registered domain associated. Take, for example, the URL *csi.gstatic.com*, a Google-controlled domain associated with the retrieval of static files, but also with advertisement/analytic services. A model with no further context would, most likely, fail to correctly identify the purpose of such requests. Let us say that Google

changes this URL to $cltInstr.statcontent.com$; Given the exact same model, it would be extremely hard to infer the purpose of analytic/advertisement tracking from this information. A possible workaround would be to introduce external information by building a corpus of domains and associating them with purposes. This works but is not scalable, since it requires an updated and reliable corpus available for querying.

Another potential point of failure is the possibility that the information is used against the model itself. In a more exoteric example, but still not a far-fetched hypothesis, we have intentionally naming the domain to make it look like a benign operation. Say, for example, that a developer wants to extract information for tracking, through a weather application; If the location information is sent to an URL like "weatherapp.solodeveloper.com/getWeather", but the information is actually being used for analytics, then it will be significantly harder, not to say impossible, for the model to label this leak as analytics. This problem arises due to the environment in which the prediction is made. The information we have access to make the prediction is almost entirely controlled by the developer itself and thus, being important to define the threat model for this proposal, as described in section 3.1.1, since a developer with bad intentions would use proxies to hide where the traffic is being sent, consider the scenario where all traffic is sent to one server, controlled by the developer, which is then redirected to the actual destination. For these reasons, additional context surrounding the request is needed, and in fact, some approaches attempt to do so. As covered before, FlowIntent [40] attempts to do this, but the design for context acquisition contains too many gaps in its methodology and thus fails to accurately retrieve this context. There are also other works which attempt to fetch this context such as Su et al. [49] by looking at who tried to initiate the request-whether it is a third-party SDK or not-which is a good starting point; however, it will fail in situations where it cannot distinguish between third-party SDK and application code, supplemented by the fact that this approach is more interested in measuring the tendencies of data collection from third parties, rather than the actual purpose for which data in the app are used. Furthermore, all network-based approaches found, which rely on Frida, either use it exclusively for SSL Pinning or, when the call stack is also retrieved, fail to deal with the thread's context switches. When a new thread is created and the execution is about to change to that new thread, the current state, i.e., the context, is stored in memory so that it can later return to where it was, and then switches to the new thread. This state includes the call stack, meaning that this information is essentially "invisible" from the perspective of the new thread, unless explicitly provided.

Returning to the problem of tool manipulation, this weakness is not exclusive to network requests but also to textual artifacts. In reality, the impact of this manipulation is even more significant for the latter. While redirecting all traffic to one single server is possible, it is not really a very practical solution, the usage of a single server refers to the scenario where the server is not only used for the core functionality, but also to redirect all third-party requests to hide the network traces, while changing the name of classes, methods and variables to make the behavior seem benign is inconvenient from a maintainability point-of-view, it is feasible. The developer only needs to change the names used for the data flows that it wants to hide (e.g., the flows for analytic behavior) to values contextually related. In reality, for most cases, this process does not even require manual work, given the existence and significant push for the adoption of code obfuscation tools. The authors of [70, 71] performed an analysis of the obfuscation situation in mobile applications, and they found that the number of applications that were completely obfuscated to the point that no useful information can be extracted is considerably low. However, this does not mean that the rest of the applications contain a significant amount of textual features such that all sensitive flows can be identified, but that they present some level of utility, in regard to preserving some of the context behind the application functionality. In 2018, a research paper that looked at the obfuscation trend in Google Play was published [85, 86], which found that around 25% of the applications were obfuscated. A more recent study, published in February 2025 [92], found that the obfuscation rate increased about 13% from 2016 to 2023, reaching a stable $50 - 55\%$ from 2016 to 2018, achieving a peak of 66% in 2023, showing that the trend indicates that the overall rate of obfuscation in Google Play will only increase. This paper also discusses the tendencies of each obfuscation technique, where Identifier Renaming (IRM), renaming of variables, classes, and methods, prevails with a rate of 99.62%. There are ways to de-obfuscate IRM, as analyzed in [93], but this falls outside the scope of this work.

Furthermore, the usage of data flow tracking tools, more specifically taint analysis tools, is extremely prevalent in this field. Considering that the goal of purpose inference is to understand what happens with the information collected, knowing the path the information takes is an exceptionally important piece of information to hold. By itself, data flow cannot be used to infer purpose; however, in combination with other techniques, such as the ones mentioned before, we can map sensitive data flows with actions performed by the application. One example of this is the use of TaintDroid [32] in Wang et al. [70, 71]

to retrieve the methods through which the sensitive data passes. This provides a really good understanding of what the application is doing with the information, but there are some considerable problems in its use. Most proposals that rely on dynamic taint analysis typically do so by using TaintDroid, or a modified version of TaintDroid to accommodate features needed by the authors. The major problem with TaintDroid entails its latest supported version, Android 4.3, which was released on July 24, 2013. For context, at the time of writing, the latest Android version is Android 16 Beta 4. To make the situation even worse, runtime permissions were introduced in Android 6.0 and, therefore, are not covered by TaintDroid. This means that by using TaintDroid, one is discarding the behavior associated with runtime permission requests, as well as making the support for newer apps extremely complicated. In addition, although more recent dynamic taint analysis tools (for Android) have been proposed, after an analysis of the state of the art, it was found that most tools were developed for more recent, but still relatively older, versions of Android, such as ViaLin. Vialin [37] was published in 2023, and runs on Android 8.0, released on August 21, 2017. However, even though Vialin is more recent than TaintDroid, Android 8.0 is no longer supported. Furthermore, from the 31st of August 2025 onward, newer apps in the Play Store must target at least Android 15, and existing apps Android 14, as mentioned on the official website [94]. There were, however, dynamic taint analysis tools proposed that do not rely on modifying the underlying OS, such as T-Recs [95]. This tool, in particular, modifies the application itself by adding logging instructions where it deems an important point in the program execution, and after execution, takes the logs generated and reconstructs the flows. In theory, this allows the analysis of any application regardless of the device, except for applications that implement integrity checks and certificates; however, after testing this tool with apps from the Play Store, F-Droid, and custom-made, none of the modified applications were able to execute without crashing, and thus, we discarded this tool. Another gap in the current state-of-the-art is the lack of a method to bridge a request with runtime information, in specific data flow data. Approaches like FlowIntent attempt this, but as stated previously, their proposed solution carries too many nuances and limitations.

In summary, while multiple different approaches and methodologies have been proposed for the challenge of purpose inference in mobile devices, all of them carry their own limitations and nuances. Coupled with the difficulty of inter-work comparison, it makes it extremely complicated to reach a definitive conclusion on which approach is the most reliable. Yet, among the different proposals, those whose focus lies on network analysis

or hooking tend to cover a critical aspect of this issue, which others have left comparatively unexplored: the **who** is getting the data. Nonetheless, these works either lack an appropriate context surrounding the data harvesting practice or the acquired context is tailored towards a different goal. Taken together, these considerations highlight the potential of the latter methodologies as means to overcoming the current limitations in purpose classification on mobile devices.

3. Proposed Methodology

3.1. Problem

To make it easier to understand our proposal, let us first understand the problem setting. Suppose that a user is using an application on his personal smartphone. This application was developed using both first-party and third-party code, meaning that the developers wrote their own code but also made use of available libraries to implement certain features/behaviors. This application will request access to sensitive information, to be used both by the first and third-party code; however, in most cases, this information will not stay in the device, but will instead be sent out to the services' endpoints, usually located in the Cloud or on a self-hosted server. In this scenario, the "person" collecting the information is not in the device itself but rather in the Cloud, and the only point of communication between the two is the application on the user's device. This defines our "adversary" as those controlling the behavior of said app, that is, the application developers and Third Party Libraries (TPL) developers, and we, as the ones analysing this data collection, can only control the information while it is within the domain of the device.

With our proposal, we want to identify the purpose behind that data transmission, providing users with the necessary context to understand why their data is being collected. This understanding is essential for users to make informed decisions about what information they are willing to share or withhold. In Android, this data collection is done, or at least allowed, through permission managers; however, these managers are not purpose-centric meaning that, unless the app's developers themselves, or the TPL developers, provide a clear explanation of the purpose behind the data collection, the only information the user gets is the fact that the data is being requested. This provides little to no control/awareness over the usage of the users' data since, in most cases, rejecting this permission request will also block the collection of that data for core/essential uses needed for the application to work, essentially locking the user behind a choice of "accept everything or nothing at all". This creates a situation where, by the permissions' requests being bundled up into one singular request, both the application and TPL developers do not need, or at least are not encouraged, to provide clear, unambiguous, and justifiable uses for the data collection. Thus, it is important to identify each of these uses individually to provide better visibility and thus, better control over the data collection practices done by the application.

With that in mind, in the remainder of this chapter, we will go over the methodological framework developed to acquire all the necessary data to construct and map context to network requests in an effort to perform purpose classification. First, we will delve into the Threat Model in Section 3.1.1, where we define and justify the baseline "adversary" which we consider in this work. Then in Section 3.2.1, we go, with significant detail, over the system architecture of LeakHound, our proposed tool, the design choices and technical considerations, and the process, step-by-step, on how we automatically retrieve all of this information to create our own dataset. Following in Section 3.2.2 we describe the data points collected, the features extracted from those data points, the reasons for which we chose these features, and the processes applied to those features. From here, we discuss the purpose classification model, where we go into the steps required to perform said classification in Section 3.2.3. Finally, in Section 3.3 we address all the challenges that we faced during the development of this tool. This final section provides not only a set of lessons learned for future research but also lays the foundation for the threats to validity discussed in Section 4.4.

3.1.1 Threat Model

As discussed previously, in this scenario the "adversary" are both the application developers, first-party code developers, and the TPL developers. With that in mid, we developed this approach assuming the developer to be "Honest but Curious", i.e. the developer will not actively try to evade detection and trick purpose inferring tools, but will sometimes attempt to collect more information than it needs/should by sending the information for other services/goals other than those which are defined/disclosed or expected by the user, usually with the intention of monetization of the data, such as targeted advertisement, analytics or other methods of further monetization beyond the core functionalities of the app. We consider this a fair base model since we believe that this constitutes the majority of the application marketplace, considering the most popular apps distributed across the major mobile application stores like Google's and Apple's app stores. Furthermore, the targeted applications will be mostly native Android applications. It is important to clarify due to the existence of apps which follow the MiniApp architecture [67] or frameworks like React Native [96]. These architectures/frameworks do not employ only native code, Java and Kotlin (which is later converted to Java Bytecode), but also JavaScript (and other Web technologies), which are usually a blind spot for most static analysis tools. Examples of

such applications are AliPay[1] and Facebook[2], respectively

However, different characteristics define this threat, such as who the developer is, how much knowledge they have, and their ultimate goal. Say, for example, a malware developer with vast expertise in Android exploitation (bypassing security and privacy mechanisms like isolation), covert data exfiltration, and detection evasion. In this situation, most approaches would be unable to detect the leak of information, not exclusively because all current proposals present weaknesses, but because the developer controls which information we, as the ones analysing the application, have access to. Take, for example, approaches based on network information; in these cases, the developer will obfuscate the body of the request, use IPv4/6 addresses instead of registered domains (i.e. instead of sending information to *facebook.com* it will send it to *57.144.180.146*), and employ proxy servers to send the traffic through a "middle man" to prevent exact tracking of where the information is ending up. If we use textual features, the developer will either remove them from the application (using nonsensical names, like obfuscation[3] does) or, in a worst-case scenario, take that to his advantage and use textual features to trick our tool into believing that the data is being used for a benign purpose. Going even further, the malicious application could even employ covert channels[4] to extract the information, delve into native code for data extraction (which poses a significant challenge for most analysis tools currently) or use other advanced techniques to avoid detection. Furthermore, application analysis, both for malware reverse engineering and for privacy analysis, was, is and will always be a "cat and mouse" game, where each side (the ones developing and the ones analyzing) constantly aims to overcome/patch protections/bypasses introduced by the other party, in turn making it conceivably impossible to introduce a fail-proof approach in privacy analysis. For those reasons, it is unrealistic to believe that any current proposal, including ours, will be able to accurately infer the purpose of data harvesting practices for all applications.

3.2. Proposed Solution

Although there are multiple possible paths for the purpose classification problem, each of them provides its own strengths and weaknesses; however, one path stands out among

[1] https://www.alipay.com/

[2] https://reactnative.dev/showcase

[3] Obfuscation commonly employs a strategy of changing most variables, classes, and constant names to nonsensical values such as "asd2as" to make it difficult to perform reverse engineering.

[4] Covert channels are a hidden method of communication mostly used to exfiltrate information secretly. One example is sending information within the headers of an HTTP request.

the others: network analysis. As covered in Chapter 2, network analysis will provide insight into *who* is getting the information. Furthermore, by monitoring this channel of communication, we can track most, if not all, information that leaves the device. This is especially important as we are mostly concerned about what leaves the user's domain and control. Furthermore, as highlighted by Falaki et al. [7], HTTP(S) is the most commonly used protocol in Android for communication, something which is also backed by Reardon et al. [8]. Apart from situations where the developer attempts to extract this information through less convenient and subvert means, such as covert channels, monitoring the network layer will allow us to have great visibility of all the information that is leaving the device, which is important as we are mostly interested in the information that leaves the user's domain and control. Furthermore, by revolving our monitoring and analysis around the network layer, we are also able to obtain a key piece of information: "who" is receiving the data. When trying to infer the purpose of the data collection, a lot of clues towards the "why" can be collected from other sources of information, as we were able to see from the methodological diversity of works reviewed in Section 2.2.1. However, something that we believe to be a key indicator of purpose classification, which was also proven by the results obtained from network-based approaches, both in general purpose classification like MobiPurpose [55] and more objective-specific works like Lalaine [53] and NoMoATS [47], is that knowing who is ultimately receiving/handling the information will provide a very important clue as to what the data is being collected for. Taking all of these points into consideration, we decided to focus our approach on network-based analysis, while trying to improve on the shortcomings that works that rely on this methodology typically present.

As discussed in Section 2.2.2, network-related tools provide a great deal of information for the classification of the purpose; however, in most cases, they lack contextual information surrounding the request made, fail to reliably acquire this context, or the context retrieved is not rich enough for general-purpose classification. In this work, we propose a new approach that attempts to close that gap by incorporating runtime/data flow information about the app together with app-wise profiling information, and mapping these findings to each network request in an effort to improve the contextual cues and therefore improve the capabilities of a network-focused classification approach. To accomplish this, we developed our own data acquisition tool, LeakHound[1], as will be discussed in Section 3.2.1. This tool collects all network requests and extracts information surrounding those

[1] https://github.com/LeoPer02/LeakHound

said requests, to construct our own dataset used to train a ML model, as will be discussed in Section 3.2.3, to perform the classification, as there is no available dataset with this information. Nevertheless, the main conceptual idea is to associate the complete call stack behind a network request, analyzing the textual features of the methods involved in the request execution, correlating this invocation to the access of sensitive information and the use of TPL, and complementing this with app-wise information such as the presence of trackers, data leaks, and number of sensitive data retrievals. As a proof-of-concept, our focus was primarily on the leakage of device information, geo-location, and contact data, as we believe these represent the majority of data leakages and the type of data users are most concerned about, a finding also reported in [70, 71]. Nevertheless, we believe that this strategy can be extended to the other types of sensitive data by extending the monitoring mechanisms to track those data types.

In the remainder of this section, we will cover the pipeline proposed to classify the purpose of the data collection in Android devices. First, we will delve into LeakHound, its design choices, and technical details/implementations. Following this, we describe the information used and the feature engineering and processing applied to it. From here, we describe how the purpose classification process will take place. Finally, we will cover all the technical challenges and limitations when implementing LeakHound and constructing our own dataset.

3.2.1 LeakHound

Evaluating our model will require a considerable amount of data, and therefore, not only will a decent amount of apps be required, but these apps should also be explored to a reasonable degree. There are multiple ways to go about this; for example, MobiPurpose [55] utilizes a VPN inside the device, stores the network requests in the local storage, and later retrieves everything. However, since in our approach we are not aiming solely for network information, a more specific data collection procedure is required.

Our tool, as shown in Fig. 3.1, has 2 phases: A dynamic phase, which will run the target application, monitor its network traffic, and hook API and threads to acquire runtime information to later associate with the network requests. And a static phase, which will run APK analysis on the application to retrieve information about data leaks and the use of TPL, while simultaneously querying a public database for the existence of trackers in the application. This information is then used to create our own dataset.

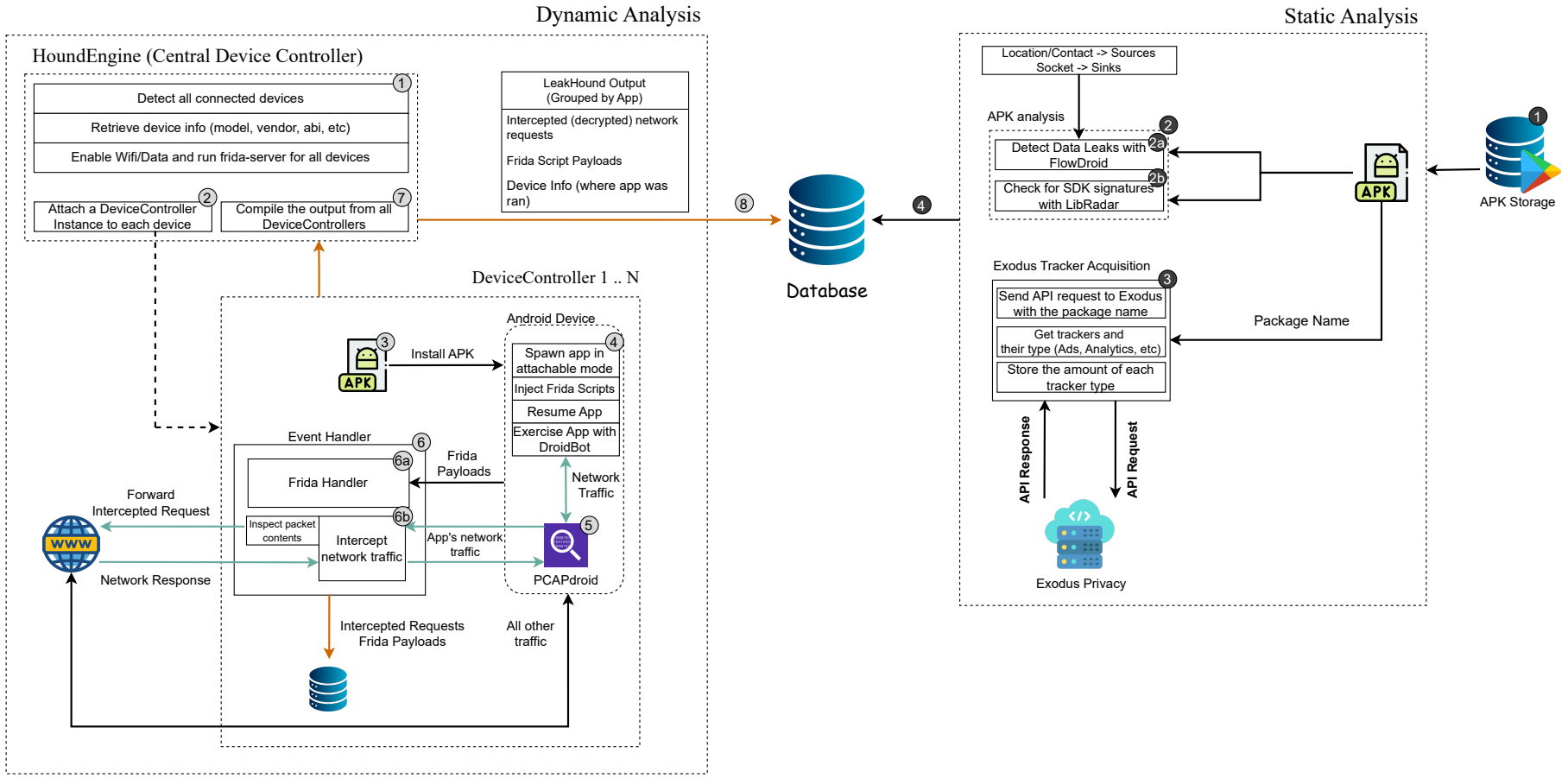


Figure 3.1: LeakHound system architecture diagram

To allow for the collection of a decently sized dataset, LeakHound allows multiple analyses to be performed in parallel. To accomplish this, the tool allows any number of devices at one time, the hard limit being the system's resources, since each connection will consume a considerable amount of resources, namely Random Access Memory (RAM), to be connected and used as the environment for the data collection. LeakHound expects a list of APK files to be analyzed, as well as configured devices detectable from the host using *adb*. The APK can be provided as a single file or, in the case where the application has a considerable size, in split APKs. The devices, however, need to be configured almost entirely by the user. There are a couple of reasons for this, first, if the user intends to use frida hooks, or strip the SSL/TLS layer from the requests, which involves installing a custom certificate in the device, then a rooted image will be required, which if done in an emulator could, in theory, be automated, or at least semi-automated, but for physical devices becomes more troublesome. Lastly, we do not know beforehand which version of Android and what vendor the user wishes to use, not to mention the possibility of using a custom ROM. For these reasons, the device will need to be manually configured by the user.

For the network trace collection, there are, broadly, two options for MitM attacks: a proxy or a VPN. Both of these work for the network trace retrieval, but from the two, we believe VPNs are the most suitable. This comes from the fact that most proxies cannot retrieve the information of **who** is sending a request. This may seem insignificant if we are only running one app at a time, but the underlying system itself makes network requests as well as some applications in the background, and to avoid introducing noise to the dataset, it is important to filter only the requests made by the application in question. Now, VPNs themselves do not provide this functionality by default, but with some work, it is possible to identify which app originated the request. For the VPN, we decided to use PCAPdroid [74], an application that serves as a VPN with specific filters for apps. We opted for this application since it provides a per-application filter and also the integration with the mitmproxy [61] library. It's important to note that, although a proxy is used, PCAPdroid emulates a VPN inside the device, and that's the way it retrieves the information; the SOCKS5 proxy is created only between PCAPdroid and our MitM server. Using these features together, we can, at runtime, redirect all the traffic to our system, filtering only the requests made by that specific application, and thus, collecting only the necessary information, while also stripping the SSL/TLS layer to access the request itself. Another major feature, especially for the integration with LeakHound, is the control

of the application over intents. PCAPdroid can be set up using intents; however, to avoid privacy/security problems, the app not only requires user consent to allow another app, which here is assumed to be the originator of the intent, to control it, but also employs checks, such as denying any remote SOCKS5 proxy server, unless specified by the user. To avoid incurring even more configuration work on the user, we modified PCAPdroid to remove every consent and security check, and thus, we can control it through intents with the adb command line.

We will now describe step-by-step how each phase in LeakHound works, explaining each step in detail with reference to the system architecture diagram in Fig. 3.1.

3.2.1.1 Static Analysis

From our list of APKs to analyze **(1)**, we take each APK and run it through two processes; The first, APK analysis **(2)**, will perform static code analysis upon the APK itself by running FlowDroid **(2a)** with our custom list of sources and sinks, which correspond to location and contacts APIs for sources and Java sockets as sinks. This will return a list of all the data leaks detected through taint analysis on the app. We also run LibRadar **(2b)** to detect TPL in the application and their name. This is important as TPL can be obfuscated within the application, and thus simply searching for their original name will not work. While this analysis is being done, we also query Exodus's database **(3)** for the presence of trackers in our app. This is done by sending an API request containing the package name of the application to the server, which will then return a list of trackers and their type, such as Advertisement, Analytics, etc. This database usually provides more hits than LibRadar, but as explained in section 3.2.2, it will be used for different purposes. For that end, we simply count the number of each tracker type detected on this application. When both of these processes finish, we store all the results in a database **(4)** to be later used along with the results from the dynamic analysis.

3.2.1.2 Dynamic Analysis

For the dynamic analysis phase, we need to acquire both network requests and runtime information in a way that we can later associate them. Since we're not retrieving the request from Frida itself, but rather through a MitM attack, we need to make sure we're able to map those 2 outputs. That is where LeakHound really comes into play; HoundEngine, the central device controller responsible for orchestrating all Device Controllers (DCs) under him, will detect how many devices are connected to the host **(1)** through adb and retrieve

the device information for each of those devices. This information is composed of things like hardware, model, vendor, Application Binary Interface (ABI), etc., and is used later in the dataset construction (Section 3.2.3) for the type inference to help us in understanding if device information is being leaked, both as hardcoded data and encoded variations of this information. After collecting the information, LeakHound makes sure to enable the WiFi or mobile data for all devices to allow network traffic to flow and once everything is set up and the applications are divided across all devices connected, it attaches a DeviceController instance to each of those devices **(2)**. A DeviceController is simply a class that will be responsible for handling all of the logic for each of the devices and their data collection, and thus, since the process is the same for all instances, we will describe the system assuming a single running instance. With the DeviceController attached and the device ready and running, we can proceed with the actual data acquisition. Using the *adb* command- line the DC will install the application inside the device **(3)** and spawn that application into an attachable mode[1], inject all the Frida scripts provided and finally resume the app **(4)**. This process allows us to inject the scripts as fast as possible to avoid losing API calls early on the application execution, something which can cause some problems as addressed in section 3.3. To trigger as many different requests within the app, we can either manually navigate the app, which the tool also allows, or we need a way to automatically traverse the activities. To achieve the latter, we utilized DroidBot [81], a UI exerciser that implements a smarter version of Monkey [77] to efficiently cover as much of the app as possible. This tool sometimes crashes the application while performing the exercise, and while it is capable of detecting that and restarting the app, that means that all our Frida scripts get lost (since these scripts are injected directly into the program's memory and not disk), and so does any information they were meant to retrieve. To solve that, we modified DroidBot to perform the spawning, injection, and resuming of the application, such that every time DroidBot restarts the app, it injects the scripts as well. With the scripts injected and the app being exercised, we need a way to capture all network traffic. As covered at the beginning of this section, we utilized a modified version of PCAPdroid **(5)** to retrieve all network requests and filter them per app. This means that all network traffic coming from the app will be tunneled through PCAPdroid's [74] VPN and redirected through our SOCKS5 proxy while any other request will be sent to the internet as usual, avoiding the problem of introducing noise from other applications. With these mechanisms in place, we need a way to handle all the data from the application. The DC

[1]Launching an app in attachable mode means allowing it to run normally and then using Frida to inject instrumentation into the already-running process, enabling dynamic analysis without restarting the application

will also run in parallel an Event Handler **(6)**, which will be responsible for getting all the information from the Frida scripts and network requests. The Frida scripts were built so that every time the given Hook is triggered, all the information is collected, parsed into a JSON, and then sent through the frida-server to our host **(6a)**, more specifically, our Frida Handler. This handler will be responsible for listening for all the data, filtering it based on where it's coming from (if it's a sensitive API call, Thread context collection, socket request, etc.), and finally storing it internally. Simultaneously, the Event Handler deploys an Interceptor **(6b)** which runs our MitM proxy. This proxy will be connected to PCAPdroid using the SOCKS5 protocol to intercept all the network traffic coming from the app. These requests are then stripped of the SSL/TLS layer and stored internally. The request is then forwarded to the actual server it was trying to contact, and the response, if there is one, is then forwarded back to PCAPdroid, which returns it to the application. Besides collecting the call stack and run time information with the Frida hooks, we also leverage the hooks to bypass one of the common protections against MitM attacks, SSL Pinning. By hooking the API used for SSL Pinning (when applications hardcode which certificates to trust), we can bypass that logic and essentially go around this check, making the application believe that it's communicating with a trusted endpoint. Furthermore, some applications also check for jailbroken devices and sandbox environments. For that end, we employ some hooks to attempt to bypass the commonly used checks for those situations. This is only possible since we do not rely only on MitM attacks, but also incorporate API Hooking with Frida. When the exercise finishes, something which is controlled through a timer, all of this information is then returned to the HoundEngine **(7)**, which compiles all of the information, from all DC, groups this information, and then sends it to a database **(8)**.

3.2.1.3 Call Stack

As mentioned initially, the reason we decided to build our own tool originated from our need for more than network traces' information. From the app, we are also interested in acquiring the call stacks of sources and sinks, as well as runtime information about the device, as explained in section 3.2.2; however, no dataset with this information is available. But before going further, it is important to have a basic understanding of what a call stack is. Although a simple concept for most readers with some basic level of programming understanding, there are some nuances that we need to consider. A call stack is, by simplification, a stack object, last in first out, which keeps track of the functions called which did not returned yet. This is important to understand because, once we

retrieve the call stack from the sink, just because a function is not present, it does not mean it was not called, since the call stack is not a history log. Another potential issue is the scope of the call stack. Say, for example, we want to send a request to a server that will take a considerable amount of time, say a database query. Doing that in the main thread can cause the application as a whole to slow down, hang, or even break. For those reasons, and others not mentioned, most, if not all, network requests are performed inside a thread. Although incorrect, for the sake of simplicity, let us assume that a thread is a new process. By launching a new thread for the network request, the entire call stack before thread execution is lost (from the point of view of the new thread), significantly impacting the ability to retrieve information about what happened before the sink was called. Given that we are only interested in network requests and, as mentioned previously, it is common practice to deal with them inside of threads, we need a way to reconstruct said call stack. Let us first understand how we collect the call stacks. First, we need to define at which points of execution we will retrieve the call stacks. The hooking location aligns with the reason why we are collecting them in the first place, as explained in section 3.2.2. Since we aim to get the context surrounding the execution of a request, we need the call stack at the moment that a request is performed. For that, we have 2 options: We either hook high-level libraries, which require an updated list of all libraries used and their APIs, something which is extremely hard to maintain, or we can look for a convergence point, somewhere where almost all libraries will reach, which are sockets. By hooking low-level sockets, we can retrieve the call stack for the majority of the requests, unless they implement their own socket logic, something extremely rare and unlikely, due to security and integrity concerns. But since we also want information about which data the app had access to, as well as the call stack for the sensitive data invocations, we also need to retrieve the call stack at all sensitive API, which in our case are all API for location and contacts retrieval. With these points defined, we now need to retrieve the entire call stack; For this, as mentioned previously, we used Frida [19], a tool which we detailed in section 2.1.2.2, that allows the injection of JavaScript code into a running app in such a way to modify/monitor/redirect methods. With Frida, we hook both the sources and the sinks API so that we can control the execution when that point in the program is reached. Once inside the hook, one possible solution would be to use Frida's $Backtracer$ class to retrieve the information; however, this will return the memory addresses of the functions. In theory, it is possible to grab the debugging symbols and map these addresses to their respective function name but after some testing, we were not able to consistently retrieve these

debug symbols from memory. To go around this, in our approach, we rely on Java's own logging and exception handling. Once we reach a hook, we trigger a $java.lang.Exception$ and log it using Java's built-in Logger. The actual function used for this can be seen in Listing 3.1, where $extractMethods()$ is a method used to parse the results into a more concise list, which can later be parsed into a JSON.

```
stackTraceHere() {
    var callStack = [];
    try {
        var triggeredException = Log.getStackTraceString(Exception.$new());
        callStack = extractMethods(triggeredException);
    } catch (error) {
        console.log("[Error] Failed to capture stack trace: " + error);
    }
    return callStack;
}
```

Listing 3.1: Call Stack Extraction

Whenever Java throws an exception, it prints to the console the entire call stack in an effort to help the developer understand what's really going on. By using this mechanism, we can force the method to return the call stack with the method names instead of the memory addresses. It's also important to note that these exceptions are also handled by the hook and, therefore, do not crash the application. In our work, we leverage Frida to recover the entire call stack behind the network request so that we can build the context behind the execution and extract textual tokens from that call stack. So, if when methods are run inside of a thread, all previous context is lost, then all previous contexts should be recorded to allow posterior reconstruction. The way we achieve this is by looking at the Threads themselves. By looking at the official Java documentation, we noticed that there are, at the time of writing, 8 different ways to initialize a Thread, i.e. 8 constructors. If, when a thread is created, all previous context is lost, then by storing the context immediately before a Thread is launched, we can later reconstruct these traces. For the actual Thread callstack retrieval we used the hook seen in Listing 3.2 which initializes the Thread, retrieves all of the information about the Thread, gets the callstack using the function shown in Listing 3.1 sends the data back to our Frida Handler in JSON format, and returns the Thread to the program. Keep in mind that this hook is for only one of the 8 possible constructors, but the only difference between this hook and the rest is the arguments passed to the constructor.

```
Thread.$init.overload('java.lang.ThreadGroup', 'java.lang.Runnable', 'java.lang
    .String', 'long').implementation = function(group, target, name, stackSize)
     {
    var thread = this.$init(group, target, name, stackSize);
    var threadName = this.getName();
    var threadId = this.getId();
    var threadGroup = this.getThreadGroup().getName();
    var callStack = stackTraceHere();

    // Try to extract the parent thread information, if one exists
    var ParentThreadName = "NONE";
    var ParentThreadId = -1;
    var ParentThreadGroup = "NONE";
    try {
        ParentThreadName = Thread.currentThread().getName();
        ParentThreadId = Thread.currentThread().getId();
        ParentThreadGroup = Thread.currentThread().getThreadGroup().getName();
    } catch (err) {}
    var threadData = { // Build the payload
        script: script_name,
        msg: {
            parent_thread_name: ParentThreadName,
            parent_thread_id: ParentThreadId,
            parent_thread_group: ParentThreadGroup,
            thread_name: threadName,
            thread_id: threadId,
            thread_group: threadGroup,
            callStack: callStack
        }
    };
    send(threadData); // Send the payload to the Frida Handler
    return thread;
};
```

Listing 3.2: Thread Context Retrieval

We also need to consider the usage of nested Threads, i.e., a Thread created inside of another Thread, which is a common occurrence, especially with network requests, where the developer calls a Thread to send the request, and the library itself opens another Thread. To accommodate this, when a thread is created, we also store the parent thread ID, name, and thread group, if one exists. Later, when a call stack from a source/sink is retrieved, and the tool detects that it's inside of a thread, these records are queried, and that way we are able to reconstruct the full call stack. This process can be seen in Fig. 3.2, although it should be mentioned that, for conciseness, the callstack and method names were truncated. In the example of Fig. 3.2, once we reach the socket or the *getLatitude* API (threads at the bottom) we get the current thread, 7 and 13 respectively, and we look up the parent thread, 4 for both; We keep doing this until we reach the top level thread with id 0 (in the example) and at that point we have all context switches and thus all that's left is to reconstruct the callstack by concatenating the callstack segments from each thread. Furthermore, by hooking the socket, we get access to information such as host, port, and timestamp used for the connection, and this allows us to effectively map a socket invocation to one or a group of network requests, thus bridging the two sources of data.

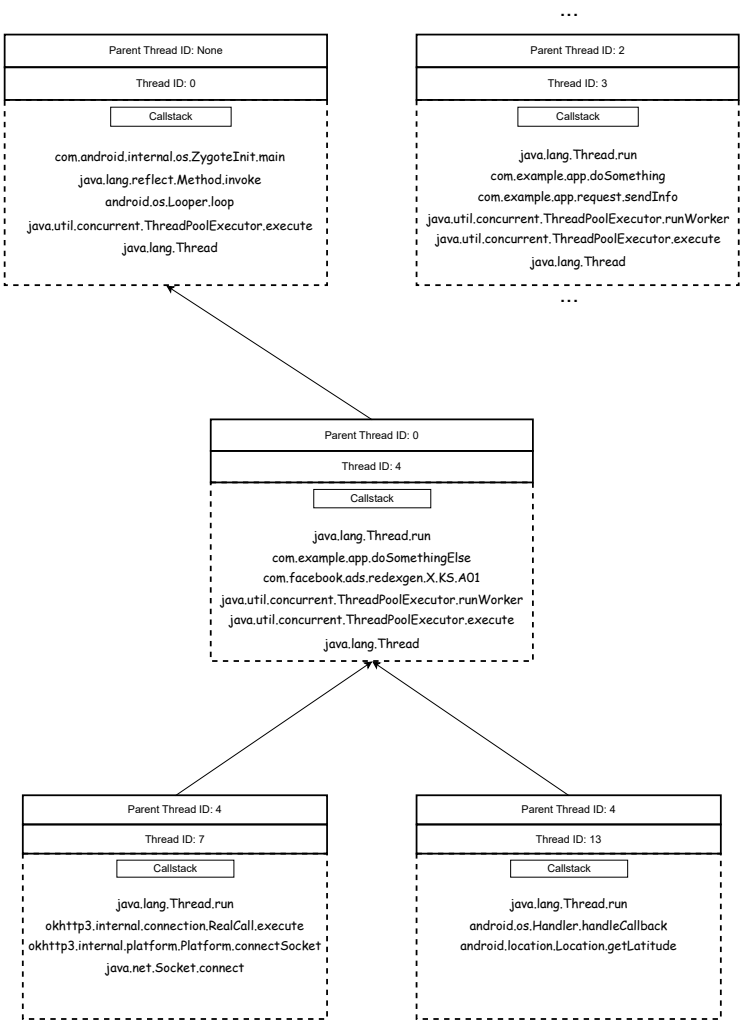


Figure 3.2: Callstack Reconstruction with truncated function names.

3.2.1.4 Runtime information

Another way to improve the detection of data leakage is to know exactly which data the application has access to; however, retrieving this information is not straightforward. Even more challenging is tracking this information as it flows through the application, the main goal of taint analysis tools. Implementing our own dynamic taint analysis is out of scope for our work, and thus, based on the reasons mentioned in section 2.3, we settled for a more simplistic approach revolving around tracking the information retrieval for dangerous permissions, from which we focused on Location and Contacts. To retrieve this information, we studied all the different ways one can request it on the Android platform. For example, if a developer wants to access the location, it can do so by calling $getLastKnownLocation()$ or $getLastLocation()$, whereas for contacts, one would make use of the $ContentResolver$

class. By developing Frida hooks for all the APIs that can be used to retrieve this information, we can store the actual device information retrieved from the device by the application, and thus, know exactly with which values we will be working. It is important to emphasize that the API hooks are, to the best of our knowledge, all the possible ways to retrieve the information.

3.2.2 Key Feature Selection

From the data retrieved with LeakHound, discussed in Section 3.2.1, we will now discuss which information we use, why, and what processing and feature engineering were applied to it. The final set of features is shown in Table 3.1.

Network Traffic		**Data Flow**		**Third Party Libraries (TPL)**	
Data Type	**Description**	**Data Type**	**Description**	**Data Type**	**Description**
Request Bag-of-words	Bag-of-words representation of the URL and key-value pairs	Callstack	Bag-of-words representation of the methods involved in the request	Exodus Profile	Total count of the different tracker types on the application
Package-endpoint similarity	A metric describing the similarity between the package name and endpoint	Callstack Distance	Metric describing the correlation between a sink and a source		
		Query Count	Total count of location and contacts queries from the application		
Co-sent data types	Tuple containing the total number of KVP and total number of different KVP types	FlowDroid Leak Match	Boolean indicator of the presence of a FlowDroid leak in the call stack	LibRadar Profile	Presence of a Third Party Libraries (TPL) on the request callstack
		FlowDroid Leak Count	Integer indicating the amount of data leaks detected on the app by FlowDroid		

Table 3.1: Summary of Features across Network Traffic, Data Flow, and Third Party Libraries (TPL)

3.2.2.1 Network Traffic

The URL of a network request is arguably the most important feature for the network analysis, as this will allow us to have an idea of “who” is getting the information, or at least

give us clues as to the type of service behind the data collection. However, the URL/endpoint is not the only data we are interested in when it comes to network requests. For this data type, we decided to follow the same logic as MobiPurpose however, since the only code provided by MobiPurpose is the one used to capture the network traffic, and not any of the feature engineering, we used our tool for the collection and performed the feature engineering at our own criteria. However, we only used the first 3 features. Although not the best performing combination in their work, the other features, as shown in Fig. 2.3 on section 2.2.3, rely on external information, particularly on a corpus of decompiled applications to check for domain occurrences. This feature proved to be a good improvement in their model; however, we believe that maintaining such a corpus up to date is too impractical, not scalable, and too time-consuming for our work, and thus it was discarded. Nevertheless, given that this feature is part of the baseline, if an improvement is seen for the baseline, that should reflect in our results as well, thus the comparison remains valid. The other two were also not used, given their dependency on Crunch Base, which in most cases fails to have the information and which is also paid for, and the app category, which proved to have almost no improvement in their model. For these reasons, for the network requests themselves, the three features we used can be seen in Table 3.1, in the “Network Traffic” column.

For reproducibility purposes, we will now discuss the feature engineering applied to the aforementioned features. For the network features only, which are common to both our work and MobiPurpose, we performed a similar feature engineering to their work to make the comparison fairer. However, not all steps from MobiPurpose could be reproduced as they depend on initial seed rules and thresholds which were not disclosed – these were recreated by us in a best effort manner using our data. Something worth remembering is that, for the dataset, each entry will be a Key-Value Pairs (KVP), instead of a full request. This is also relevant considering that each KVP can present its own purpose within a request, making this classification more specific and capable of identifying multi-purpose requests. For the URL bag-of-words, for each request, we gathered the URL and key-value pairs and extracted a bag-of-words from them. However, to avoid bias towards top-domains such as $googleads.g.doubleclick.net$ or $graph.facebook.com$, we defined a list of the top most common URLs in our dataset and clustered all requests for those domains by using regular expressions, such as

$$\hat{}https://graph\backslash.facebook\backslash.com/v\backslash d+.\backslash d+/app/model_asset.*$$

and then filtered out repeated KVP to avoid overfitting the model. As an example, for the previous regular expression, requests to URLs such as “https://graph.facebook.com/v1.2/app/model_asset_img” and “https://graph.facebook.com/v3.4/app/model_asset_disk” would be joined into a singular entry where any unique KVP would be kept, and all others would be dropped. For the token extraction, we used the wordsegment library [97] with the addition of stop words from the vocabulary domain, such as `com`, `http`, `keepalive`, etc. After a first round of word extraction, we sifted through the results to check if this extraction was done correctly. Looking at the results, we found some instances where the division was not done correctly, like in cases such as ”*googleads*”, which was not separated into ”*google*” and ”*ads*”, and thus, we also created some hard-coded rules for those cases we found; only five relevant cases were identified. This bag-of-words was later vectorized; We tried 3 different vectorizers: Count Vectorizer, TF-IDF Vectorizer, and Word2Vec. The last 2 are the most common options for NLP; however, we found out from our results that, given the domain of the vocabulary, a count vector was usually enough. This stems from the fact that in a URL or Key-Value Pairs (KVP), there is no real expectation of what other word should follow, since they are usually very short indicators and the document (or string) is very small. For those reasons, using Count vectorizer is usually more than enough, but, nonetheless, using TF-IDF or Word2Vec does provide a slight improvement over Count vectorizer, but at the cost of efficiency, which is not critical in our case due to the dataset size. For the package-endpoint similarity, the second feature, we initially followed MobiPurpose’s method of segmenting the endpoint address into 3 parts (subdomain, domain, path) and computing the longest common substrings between the package name and the three components separately. However, later, we also experimented with string similarity algorithms such as Levenshtein and Jaccard distances, which provided a slight benefit, while the results between the different algorithms were negligible, and therefore, we went with the latter. Finally, for the last feature, the Co-sent data type, we perform the same feature engineering process of counting how many key-value pairs were in that single request and the count of the different Key-Value Pairs (KVP) types in the request, which we classified with the type inference model that will be covered in Section 3.2.3. Nonetheless, a data type refers to what type of information the KVP holds such as location, contacts, among others.

3.2.2.2 Data Flow

The second type of data we used in our analysis is data flow. Inferring the purpose of data collection requires us to first understand when and where data leaves the device. The approaches that more heavily rely on this route usually do so by integrating static and/or dynamic taint analysis tools to retrieve the data. When it comes to dynamic taint analysis, we will run into multiple issues and be limited in our analysis, as discussed in 2.3, and thus, we decided to go through a different route. However, before delving into which tools were used and how they were used, let's first talk about what our goal is in data flow analysis. As mentioned previously, knowing who gets the data is an extremely important indicator of what is being done with the data; however, this is not always enough, especially when we have domains that are not very indicative of the service they implement, for example "googleads.google.com" compared with "graph.facebook.com". One way to fix this would be to have a corpus of endpoints and their apparent purpose/service/use; however, given the endless number of URLs, this task is impossible. To overcome this, we decided to focus on the information we can retrieve from the app itself, since this way, it will always be available without the need for external information. For a network request to be triggered, all the logic behind the request has to be processed. The process here is to look at what instructions/methods/functions led to the triggering of a network request, as that will be an indicator of their purpose. In this case, for each network request the app makes, the entire call stack at the point of execution will be collected and used to understand who/what was involved in initiating this request. Once the call stack for each request is acquired and mapped to the actual network request retrieved, explained in section 3.2.1, we retrieve textual tokens from the methods involved in the request. Similar to the bag-of-words used in the network features, we use wordsegment to extract words from the methods, remove some stop words related to the domain and remove entire methods if they are associated with framework operations (e.g. "*android.os*", "*android.app*" and "*java.lang.Thread*") as these will be extremely common but add little to no information about the purpose of the request. With this bag of words created, we vectorize it using the same vectorizer type as in the network features; however, this matrix will be separate from the network features, given that their vocabulary domain is different. Using the call stack, which leads to the triggering of the requests, already provides good information about what the application is trying to do with the data, but we found that it could also be used for another purpose. As stated, this problem is a matter of understanding where the information is going, and thus, we need to identify which information is being sent out. As will be covered in 3.2.3, we

deploy a type inference model to infer the data types present in the requests, associating a data type label to the relevant KVP extracted; however, we can enhance this with the callstack information as well. By extracting the call stack not only at the sink point (socket) but also at the source (location and contacts query), we can try to associate the two in an attempt to check if these two calls could be correlated. When we have all of these callstacks, we can, as shown in Figure 3.3, form a "graph" and check the branching point, i.e., the point at which the callstack splits and starts following different paths. Thus, this graph can be divided into 3 sections: the branching point, where the call stack diverges, the prefix, the paths taken after the branching point (it's important to note that the methods are added to the bottom), and the suffix, the path where the call stack is shared among all paths. This information can help us in the following way: The "deeper" the branching point appears in the call stack, the more specific the actions in the application will be. You can think of it as if the branching point happens on the OnCreate framework method, which is one of the first to be triggered in an activity, then one call stack can branch to GUI logic, while the other will branch for database operations. However, if this branching happens later in the program, it generally means we are within a sub-specific method.

In the example from Listing 3.3, and its graph representation in Fig. 3.3, the branching happens in "loadDashboard" so it's fair to assume that any future methods will be used for the retrieval/processing of information for the user dashboard. Applying the same logic to the sinks and sources in our problem, the deeper the branching point, i.e. the longer the suffix, the more specific and correlated the two will be. However, we also need to consider the prefix; In the example of Listing 3.3, the suffix is the same, however, the prefix, i.e. , the paths after divergence, increased and became even more specific.

```
1 Path A: onUserPress → initializeSession → loadDashboard → retrieveUserLocation →
      getLocation

3 Path B: onUserPress → initializeSession → loadDashboard →
      retrieveTermsAndConditions → checkIfTermsCached → retrieveUserCountry →
      getLocation

5 Path C: onUserPress → initializeSession → loadDashboard → retrieveShops →
      sendGetRequest
```

Listing 3.3: Callstack Distance Example

The longer the prefix, the "further away" these methods will be from each other and thus, the more hyper-specific their purpose will be, and the less likely that they will be used together. In the previous example, we present 3 call stacks of equal suffix but different

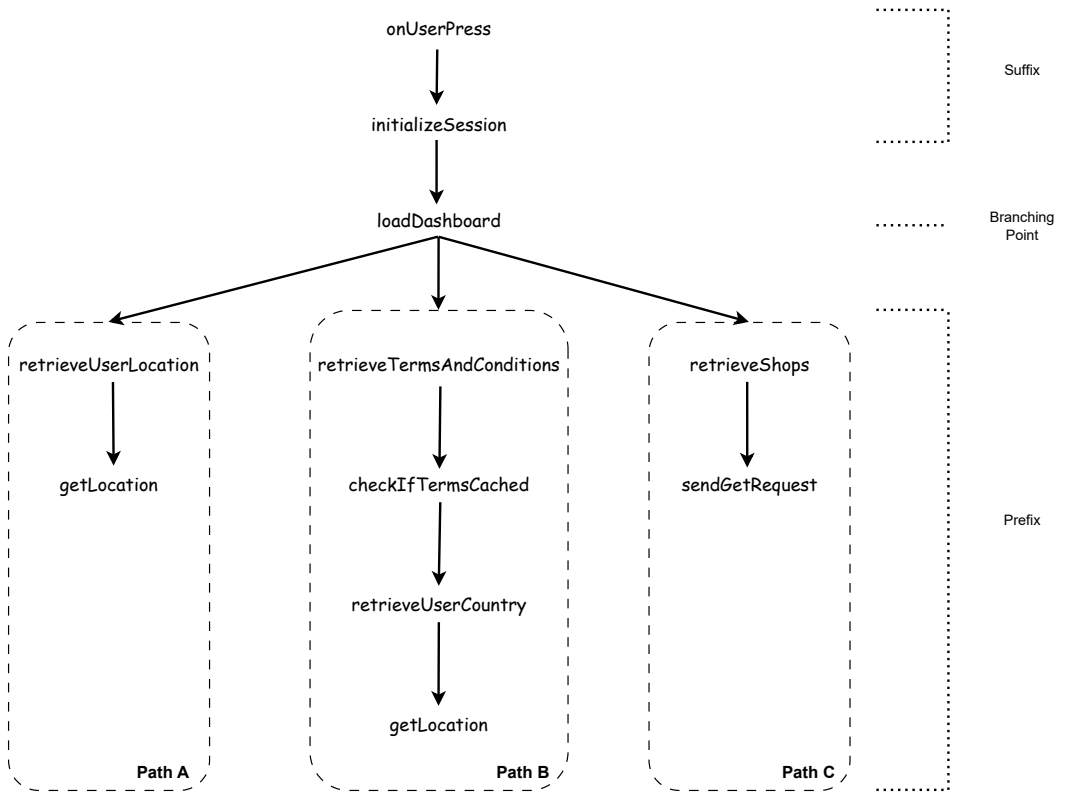


Figure 3.3: Callstack Distance Example Graph

prefixes. When looking at them, we believe that it is a fair assumption that Path A is more likely to be retrieving location information to be used in the request of Path C than it is for the location to come from Path B, which seems more likely to be retrieving the location information to request the correct legal document. This is because the longer the prefix, the more specific the goal of the call stack becomes. Nevertheless, it is important to clarify that this is in no way a foolproof method of identifying data flow correlation. An example of how bypassing such a check would be possible is to retrieve the location as soon as the application launches, store it somewhere, and later load it. Instead, this method will help trace the correlation of method invocation between sinks and sources when done “closely” together, boosting our confidence in claiming that a given type of sensitive data is being sent out by a request. Furthermore, by hooking the sources (location and contacts), we can keep a registry of the sensitive information that the application had access to, i.e., if the app requests the user’s location, the Frida hook will not only retrieve the call stack at that point but also retrieve a copy of the coordinates passed to the app, as well as the number of occurrences for those retrievals. This is what allows us to perform an initial check for hardcoded or encoded versions of this data in requests (such as geo-hashes or Base64 encoding) in the type inference model discussed in Section 3.2.3.

Finally, and as the last source in the data flow category, we have static taint analysis using FlowDroid. As mentioned multiple times throughout this work, taint analysis is usually in the spotlight in related works and for good reason. Tracking data flows in an application allows us to identify the leakage of data, making the whole process of classifying the purpose of data usage much easier. However, these tools come with their own sets of problems, as discussed in sections 2.1.2.1 and 2.3. This is one of the reasons why, in this paper, there is a bigger focus on using Frida for data flow information collection rather than Taint Analysis, as we also wanted to understand the reliability of some different tools when solving this issue, in an effort to provide alternatives to such tools. Nonetheless, we still believe that these tools can provide important data for this problem, and thus we decided to leverage FlowDroid [26] into 2 features. The first feature is a request-bound mapping between the call stack and FlowDroid leaks. After collecting all leaks from the app using FlowDroid, we check if the call stack of a request matches any of those leaks, in which case, we know that the request is associated with the given leak. To perform this mapping, we look for a non-continuous sub-sequence between the call stack and the leak. The reason for this choice is that the leak is not a call stack but rather an ordered sequence of instructions from a source to a sink; meanwhile, a call stack can be thought of as a record of all the functions called, which have not returned yet. For this reason, trying a one-to-one matching would not provide any results, and thus the mapping is performed by identifying a non-continuous sub-sequence within the leak corresponding to the call stack. An easier way of interpreting this is that for a call stack and a leak to be related, all methods from the call stack need to appear, in order, inside the leak, even if there are methods in between that do not appear in the call stack itself. This feature reinforces, or uncovers, the use of sensitive information on a request that would otherwise not be found, or reinforces our confidence in its presence. One of the main drawbacks in terms of data leakage monitoring for network-based approaches is the sole reliance on identifying this data through the request body, which, a lot of times, is obfuscated. By incorporating a static taint analysis tool and comparing it with the call stack, we increase our confidence when it comes to data leakage detection. The second feature extracted with FlowDroid is a simple app leak profile. This leak profile is nothing more than a count of how many leaks the application contains. In the end, we can have a more general data usage profile for the app, which will give us more insight into what the application is doing with the data.

3.2.2.3 Third Party Libraries

Finally, a last piece of information that provides great value in the classification of a request purpose is the presence and number of TPL in an app. We retrieve this information from two different sources. The first is Exodus Privacy [98], a public database of trackers for Android applications. From this database, we can query all the trackers present within a given application and get not only the names of these trackers but also their purpose, such as Advertisement, Analytics, etc. With this information, we can better paint a picture of the overall behavior of an application. We believe that it is a fair assumption that an application that carries more trackers, especially those for advertisement and analytics, is more likely to send requests for those purposes, since most of these interactions are usually implemented with the help of TPLs. With that idea in mind, for each application we analyze, we query this database and check the presence of the trackers. Since most of the information we get is the name, type, and metadata about the trackers, we decided to simply count the number of each type of tracker present in the application. Since there are a lot of different types, we focused on **Advertisement, Analytics, Crashalytics, Identification, Location, Profiling** and then a **Other**, which serves as a fallback for everything else. The second source for TPL is LibRadar [78]. LibRadar, in contrast to Exodus, is a tool that is run on the application instead of a database to be queried, and so, even if there's no record in Exodus for a given app, we might still be able to get some TPL information about the application from LibRadar. However, we don't just use LibRadar as a fallback from Exodus, especially because the number of apps not present in Exodus was not that significant. The reason why LibRadar is helpful is that it not only identifies the TPL, although with less accuracy and in less quantity compared to Exodus, but it also identifies the name of the package on the app under analysis. As mentioned in section 2.3, obfuscation in applications is increasingly more prevalent, and this obfuscation also applies to TPL inside the apps. For this reason, being able to identify the name that a TPL has in this instance of the application allows us to perform a more fine-grained and targeted classification. This comes in handy with the call stack retrieval mentioned previously, where, if we detect the presence of a TPL on the call stack of a network request, then the request is more likely, but not guaranteed, to serve the purpose of that TPL. So, after acquiring the TPL with LibRadar, we build a corpus mapping class name to TPL type, just like the ones defined previously for Exodus, and if the TPL is present in the call stack of a given request, we increment the type of feature present. We found that basing the entire decision of the detection of TPL, although promising for approaches

mainly focused on detecting A&T services, is too unreliable for general-purpose classification. An example of this can be seen in the Listing 3.4, where LibRadar identifies the package *com.google.android.gms.ads* inside of this app. With this information, after reconstructing the entire call stack of a request, we were able to identify the use of this package in the program. Note that the TPL name only covers part of the method, as LibRadar only identifies per root package name and not every single class within the TPL.

```
...
java.lang.Thread
com.google.android.gms.ads.internal.util.future.b.newThread
java.util.concurrent.ThreadPoolExecutor$Worker
java.util.concurrent.ThreadPoolExecutor.addWorker
java.util.concurrent.ThreadPoolExecutor.execute
com.google.android.gms.ads.internal.util.future.d.execute
com.google.android.gms.ads.nonagon.ad.activeview.b.a
com.google.android.gms.ads.internal.js.j.a
com.google.android.gms.ads.internal.webview.r.x
com.google.android.gms.ads.internal.webview.r.e
com.google.android.gms.ads.internal.webview.an.run
android.os.Handler.handleCallback
android.os.Handler.dispatchMessage
m.bbd.a
com.google.android.gms.ads.internal.util.f.a
m.bbd.dispatchMessage
android.os.Looper.loopOnce
android.os.Looper.loop
android.app.ActivityThread.main
java.lang.reflect.Method.invoke
com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run
com.android.internal.os.ZygoteInit.main
```

Listing 3.4: TPL indentification in callstack with LibRadar

To clarify, although these 2 sources retrieve a similar type of information, they work at different levels of profiling. Exodus paints a more general picture of the app behavior, while LibRadar [78] associates a request with a given TPL usage, thus completing each other.

3.2.3 Purpose Classification

With all the information retrieved, we can look into classifying the purpose of the data collection; however, we first need to understand which data types are being transmitted, as this will dictate the possible purposes. To make it easier to perform a comparison, we utilized MobiPurpose's purpose taxonomy, seen in Annex A, which is grouped by data type. This is done as different data types will have different purposes, such as location (in Table A.4), which will have "Map Navigation" while IDs (in Table A.1) will have "Authentication". Also, as requests can be multi-purpose, performing this classification at the KVP level will allow us to identify all different purposes for that request. Meaning that, although we

are performing the classification for each KVP, these will translate in the request, which contains these KVP, having multiple purposes.

In our approach, the data type will be classified through a 2-phase process: one based on pattern matching and the other which relies on a NLP model. For this, after collecting the network requests, we parse the request, extract all Key-Value Pairs (KVP) in the body, headers, and queries, and check for the presence of sensitive data. An example of a desensitized example, which sends out location information, can be seen in Annex A in Listing A.1. As for the possible different data types, since in each table we have multiple categories, we will be referring to them as "TABLE.TYPE". For example, for the location in the Sensor taxonomy Table A.4, we will refer to it as "SENSOR.LOCATION" and network in the Phone Table A.2, we will refer to it as "PHONE.NETWORK". Another 3 labels which are not shown in these tables are `non-privacy` ("NONPRIVACY.NONPRIVACY") which, in the taxonomy definition process, was introduced for cases where the likelihood of the predictions fell under a specific threshold in their model; `insufficient` ("INSUFFICIENT.INSUFFICIENT") was added for the manual classification process, where the participants had the option to choose this in the case they did not have enough information to classify, and finally `unknown` (with the same code as "unknow") which is chosen in the case none of the others apply.

For the first phase of the classification, we look for the presence of special strings (IPs, Emails, Advertisement IDs, among others), and if they match, we immediately classify them with the data type under which that regular expression falls. Furthermore, with the runtime information collected for location and contacts, as discussed in Section 3.2.1.4, we can look for hardcoded versions of the actual information collected. However, solely looking for hardcoded strings within the request will, in most cases, not be enough. Most of the time, the information format changes before being sent out of the device. To improve this, we look for encoded information in different formats to cover a wider range. For location, there are 2 common "encoding" processes used, GeoHasing and location truncation. The former revolves around encoding the geographic location into a string of characters and digits, while the latter removes decimal places from the coordinates to reduce their precision, making the location more general. In reality, truncation can also be used with geohashing. Since we have access to the actual values collected by the application, looking for the GeoHashing version is a matter of converting the value; however, with truncation, if we start considering all possible sizes, we might run into the issue

of wrongly classifying the information (47.2 could be both latitude or the price of something), and as such, these cases are left for the second phase. For emails, we mostly look for their presence, as a complete string, while for contacts, we take into consideration the presence of the country code, or lack thereof. Additionally, across all data types, we also look for Base64 encoding. It's important to mention that the idea behind looking for Base64 encoding is not to detect covert information leakage but rather to cover data transmission, which, for reliability purposes, uses Base64. The topic of covert channels and other advanced data extraction techniques [8] is beyond the scope of this work, as it requires extensive monitoring not only of the application itself but also of the underlying operating system.

For the second phase, we rely on an NLP model for text-based classification. When looking at the dataset, we noticed that there were mostly two scenarios: Either the data was completely obfuscated/random, in which case not even a human would be able to infer the type, or the information could be clearly identified by either the format or some keywords. To exemplify, most of the contact key-values would contain the words "username", "email" and "contact" and/or have as a value an email, which can be easily identified by the format. The device data type was also overwhelmingly dominated by the presence of words such as "sdk", "device", "version", among others, making them easier to be identified. As for the model itself, we tested multiple configurations from the spacy library[1], and in the end, we found that using a Convolutional Neural Network with Tok2Vec vectorization provided the best results. It's also important to note that this model was built on top of the English language, since most of our applications were in English. Nonetheless, this could be easily extended to support different languages.

With each KVP in the request classified, and all the features collected, we can now perform the purpose classification. The training process will be discussed in more detail in Section 4.3, but the fundamental idea is to employ a ML model, which in our case is XGBoost, to classify the purpose. This model will take all of the information acquired and processed as discussed previously, and use it to classify each individual Key-Value Pairs (KVP) for each request. Since each data type has its own possible purposes, in the end, we trained a model for each data type. This will allow each model to generalize only for the purposes for which those specific data types can be used, as well as allowing the model to become more specific, avoiding having one singular model classifying all possible purposes across

[1] https://spacy.io/

the entire purpose classification taxonomy, which enables an easier future expansion of said taxonomy.

3.3. Challenges

During the implementation of our framework, several limitations and challenges emerged that warrant attention in future work. A thorough discussion of these constraints is essential, as they may influence the outcomes presented in this study and provide valuable insight for future improvements.

One of the limitations is that this thesis was developed for English-based applications only. This is especially important given our use of textual features to infer the purpose, whose model was trained using the English language. This could be easily adapted to incorporate other languages; however, since our implementation serves as a simpler proof-of-concept for the overall idea, we only considered the English language (the most commonly used language in the Western market).

To obtain data leaks with static taint analysis in the applications, we used FlowDroid [26], which, to the best of our knowledge, is the most accurate and only continually maintained static taint analysis tool for Android. However, this is not without its problems, such as the app crashing in some applications or taking a significant amount of time even when using time limits (usually caused by the tool being stuck while running Soot [99], which is the foundation of FlowDroid) and consuming exorbitant amounts of resources, mainly RAM. These are known problems, and for that reason, we cannot assume we will always have information from FlowDroid, nor that FlowDroid will be able to detect leaks even if it can run.

Continuing on the topic of static taint analysis, two more problems arise. The first being the lack of an updated API-permission mapping. Most of the literature uses tools such as PScout [100] to achieve these mappings; however, as of the writing of this thesis, the last update to that tool was made in 2015. Initially we mapped the permissions by making use of multiple tools like PScout [100] and Axplorer [101], and if the API was not found we would use a similarity heuristic to try to infer the permission, i.e. based on the class name of this new API we would attempt to identify similarities with known APIs and assign the data source type that way. However, we later decided to create our own sources and sinks file for FlowDroid, and so, given that we were choosing our own sources, we knew which information was being retrieved. This is in no way an extensive mapping dataset

nor a significant contribution to the API-permission mapping problem; this is simply the approach we found most suitable for our work. The second problem is the increasing number of native React apps or apps that use WebView. This is a problem since these apps will utilize JavaScript in their logic, something that FlowDroid, to the best of our knowledge, does not track. This could be addressed by using JavaScript-specific analysis tools; however, since these do not represent the majority of the application marketplace, we decided to use only FlowDroid. Keep in mind that, even though FlowDroid does not track JavaScript, these apps still contain native code, and thus, we may still be able to detect some leaks.

Furthermore, while analyzing applications to generate our dataset, we ran into the same problem most dynamic/hybrid approaches face: app coverage. In an ideal scenario, an expert would manually traverse the app, using their technical and common knowledge to try to trigger every possible activity/state of the app. However, if we wish to cover a large set of applications, we need to automate this process, and thus most approaches, including ours, rely on automated UI exercisers. As mentioned previously, these tools, although useful, suffer from a significant lack of coverage, especially when it comes to login, QR codes, or the need for external information (such as a second factor of authentication). In addition to these common points of failure, we also identified other potential issues. Given that we cannot let the tool run for an indefinite amount of time, since we risk either taking too long to analyze a single application, making batch analysis impossible, or running into an infinite loop, a timeout needs to be set. Nevertheless, there are nuances/edge cases that also need to be considered. As an example, while creating our dataset using DroidBot, which we found significantly more reliable and stable than Monkey, we noticed that the tool would get stuck (doing a similar set of actions until the timeout was reached) in actions that did not translate into navigation or relevant network data collection. In our case, this happened often with filters in search activities, where, after loading all the items in a list, the tool would get stuck trying every possible combination of filters. This would not be that big of an issue in cases where the default behavior is to show nothing, and when adding new filters, it shows more information, working more as a sort of checkbox, as in that case, when selecting a new filter, more information would be loaded and more requests would be performed. This can be fixed by giving the tool a larger timeout, allowing it to exit this state, at the cost of a significantly more time-expensive analysis.

When developing LeakHound, multiple tools were considered and experimented with, to assess which ones are the most adequate; however, many of these tools, namely those

of the taint analysis category (both dynamic and static), are not production-ready tools. This is reflected in the lack of documentation, support, and troubleshooting guides. This then translates into significant amounts of time invested in either attempting to set up and run these tools, or time dedicated to adapting/fixing the tool to accommodate use-cases and fixing issues. As examples of these situations, for FlowDroid [26], a considerable amount of time was dedicated to understanding the best combination of search depth, timeout, engines, and backtrack algorithms, among other settings, that would be able to analyze the largest number of applications, while providing enough value. Time was also dedicated to attempting to modify the tool to resolve some of the issues/bugs that were preventing the analysis of certain applications. We also considered alternatives to FlowDroid, such as proguard-core [102]. We looked into leveraging this tool (the open-source version) to retrieve the leaks; however, since ProGuard-Core analyzes Java Bytecode in general, not just for Android, and Android has multiple entry points, we were not able to fully leverage this tool for the task. We also faced a roadblock when it came to dynamic taint analysis tools. At the beginning, we experimented with TaintDroid [32] to assess the type of results and value we could extract from the tool. However, the tool dates back to 2014, and the latest Android version it supports is Android 4.3; thus, we decided that it would provide little to no benefit to depend on it. Thus, we moved into looking for newer tools; We tried Vialin [37], however, with no success in making it work, even after multiple attempts, different Android versions and vendors, and with some guidance from the tool's authors. We then also looked into dynamic taint analysis tools, which do not modify the underlying operating system, such as T-Recs [95]; however, after multiple attempts at analyzing applications with these tools, most of them would crash immediately after starting, making the analysis impossible. In the end, the time devoted to these tools did not result in usable progress.

Finally, this tool was developed to gather information to generate a dataset that would be used to train a ML model. And the reason for the creation of our own dataset, as will be covered in more detail in Section 4.1, is that there is currently no available dataset that provides this information. MobiPurpose [55] made their dataset available; however, this dataset only provides the network requests and metadata about the app. For that reason, we created our tool to acquire the necessary information and create our own dataset. However, as will be covered in more detail in Section 4.4, this comes with its own set of problems. As discussed previously in this section, tools would break or fail to retrieve

information, some would take exorbitant amounts of time and/or resources, and thus render a batch analysis inefficient, the API-permission mappings had to be created manually, coverage of the applications with UI exercisers is limited, among other problems. To further complicate the issue, most network requests performed by an application do not contain sensitive information, which, when combined with the process of having only one person labeling each Key-Value Pairs (KVP) within every request manually, makes the creation of a decently sized dataset a challenge, thus limiting the final size of our dataset. This also affected the total number of apps analyzed, as for us to retrieve any significant amount of usable entries from the application, we need to sift through a significant amount of unusable requests, and all their Key-Value Pairs (KVP).

In this chapter, we covered the design of our tool, which retrieves network requests, as well as information surrounding these requests, in an effort to build a rich context used to classify the purpose of the data collection in Android applications, with a more general taxonomy, beyond just the network requests. To achieve this, we built a framework to retrieve all network requests, runtime information such as call stacks and sensitive data retrieved by the app, and the use and presence of TPL. As discussed in detail in Section 3.2.2, all these different types of data are then correlated and processed into a richer context, forming the dataset used to train our ML model. Nonetheless, this does not come without its own set of challenges, as previously addressed in this Section, namely, technical limitations related to the tools used and time restrictions associated with the manual labeling process of the dataset. These issues can and should be addressed in future work, particularly through dataset expansion by involving at least a second labeler in the process, as well as by overcoming the previously discussed technical challenges, which impact not only the quantity but also the potential quality of the collected data.

4. Evaluation

In this chapter, we will cover the entire process behind the evaluation of our model, how it competes against a similar approach, and analyze its results, as well as the process to obtain the dataset. For the reasons covered in Section 2.2.2, we chose to follow MobiPurpose's evaluation model structure and taxonomy, since that way we will be able to perform a direct comparison. However, MobiPurpose's analysis was limited to network traces and metadata about the application. Since our analysis makes use of runtime information, such as call stacks, and TPL we created our own dataset using our tool, LeakHound, something which will be discussed in section 4.1. Although not the only reason, by having our own, manually labeled dataset, we are able to better understand the performance of our model, as all data used to train it was reviewed and classified by a person. To summarize, due to limitations on inter-work comparison, our model will be compared against MobiPurpose's model with a manually labeled dataset to provide a better insight into how our model performs overall and in comparison with past work.

As for the metrics used to evaluate our models, we will be referring to Precision (P), Recall (R), and F1-Score, both macro and weighted. Precision measures how many of the predicted positive instances are actually correct, while recall measures how many of the true positive instances the model successfully identified. The F1-score combines both into a single value by balancing precision and recall. Since classification often involves multiple classes, the F1-score can be averaged in different ways. Macro F1 computes the F1 for each class and then averages them equally, giving the same weight to rare and frequent classes. Weighted F1, on the other hand, also averages the F1-scores but takes class size into account, so larger classes have more influence on the final value.

In the remainder of this chapter, we will first go over our dataset and why we created our own. Following that, we will discuss our type inference model and its performance. Next, we will discuss the training process of our purpose classification models and the results we obtained for each of them. Finally, we will analyze all the limitations and threats to validity present in this work.

4.1. Dataset

We created LeakHound specifically to generate the dataset on which to train our ML model. The reason why we created our own dataset is that no dataset with this information is available. MobiPurpose's dataset only provides the network requests and metadata about the app, as that is all they care about. For that reason, we decided to develop LeakHound, which collects all the network requests, the runtime information like call stacks, sensitive data collected, FlowDroid [26] leaks and TPL usage/presence, correlates the information into a rich context, which it then uses to build the dataset[1].

LeakHound was designed to analyse multiple applications at the same time to reduce the amount of time needed to collect enough entries. As for the applications chosen, we initially started collecting applications based on whether they were present in MobiPurpose's dataset, to make a closer comparison with their test environment; however, since the version of these applications were not provided, and a lot of these applications had since been updated/removed, we eventually moved into selecting the top applications based on their category/description depending on the label distribution at the time. So, for instance, if we noticed that the label "Map Navigation" did not have enough entries, we would look for top navigation-related applications from the Play Store to increase the chances of finding such requests, as there are only so many applications we can cover in the limited amount of time. In the end, and for reasons we will discuss in section 2.3, we analyzed a total of 43 applications, which can be seen in Table A.5

4.2. Type Inference

After training our model, as explained in Section 3.2.3, we obtained the results as seen in Fig. 4.1. The type distribution in our dataset can also be seen, in order of count, in Table 4.1. All data types have an F1-score of $\geq 95\%$, which initially might indicate data leakage; however, as discussed previously, the data type classification is either trivial or impractical, which explains our results. Nonetheless, we decided to run our model in MobiPurpose's dataset to validate the performance, and we noticed that some of the errors our model was making derived from a correct classification in cases where MobiPurpose's failed. These cases were most frequent for the data type "ID.GENERALID" with examples such as ***mopub_id = ifa 81...6452d*** and ***app_id = app363...7ab7***, and "PHONE.DEVICE" with cases like ***devicever = 5.13*** for which MobiPurpose's classification was "unknown".

[1] https://github.com/LeoPer02/LeakHound

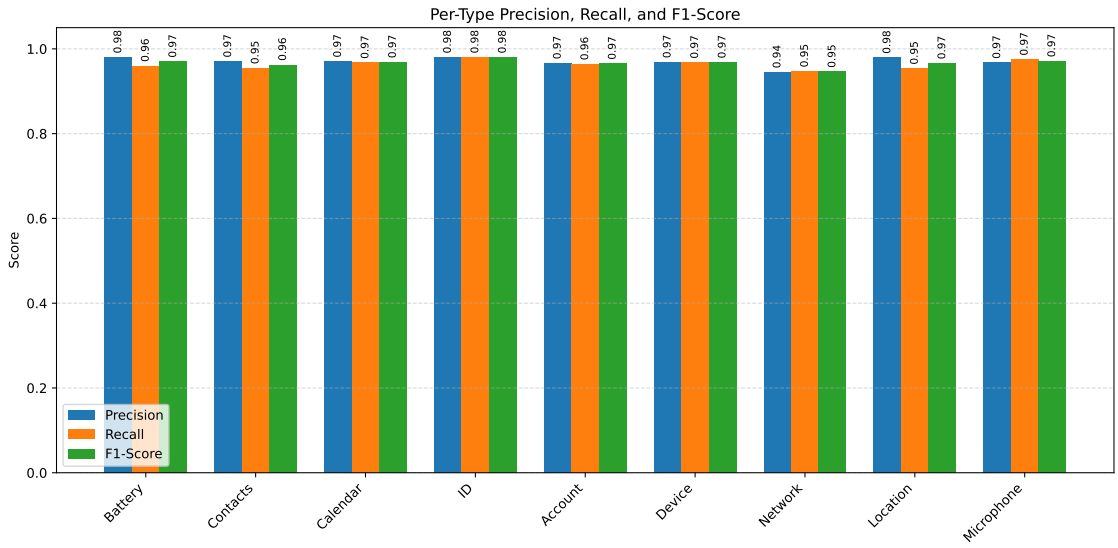


Figure 4.1: Type Inference Results

Another noteworthy result is the low frequency of “NONPRIVACY.NONPRIVACY”, which, after a review of the results, was seen to be caused by the model usually resorting to “unknown” or “INSUFFICIENT.INSUFFICIENT” instead.This is mostly caused by the fact that, “NONPRIVACY.NONPRIVACY” was created specifically for cases where the likelihood of the prediction fell under a specific threshold, something that does not apply to our model. Although not ideal, as will be discussed in section 4.3, these 3 data types will not be used for the model training process, and thus, this difference won’t affect our results.

Label	Count
ID.GENERALID	1282
PHONE.DEVICE	1253
unknown	1141
INSUFFICIENT.INSUFFICIENT	548
PHONE.NETWORK	420
PERSONAL.ACCOUNT	288
SENSOR.LOCATION	172
NONPRIVACY.NONPRIVACY	88
PERSONAL.CONTACTS	26
SENSOR.MICROPHONE	25
PHONE.BATTERY	24

Table 4.1: Label Frequency Distribution

4.3. Purpose Classification

For the purpose classification, as discussed in Section 3.2.3, we followed the methodology of MobiPurpose. Particularly, we deploy a different model for each data type, which was inferred previously, as described in section 4.2, and train each model for the labels/classes

associated with that data type. In the end, this will allow each request to have multiple purposes, but only those associated with the specific data type. For example, if we look at the “SENSOR.LOCATION” data type in Table A.4, it will have “Map Navigation” but not “Interface Customization” as “PERSONAL.ACCOUNT” in Table A.3 does, and vice-versa. However, notice that there are a total of 11 data types but only 5 models in use; There are two main reasons for this: first “unknown”, “INSUFFICIENT.INSUFFICIENT”, and “NONPRIVACY.NONPRIVACY” represents entries that are not privacy-sensitive or could not be identified, and thus there’s no reason to try to infer their purpose. The second reason is that, as seen in table 4.2, the rest of the missing labels have a very low frequency, or were not seen, making it extremely hard to train a model in or simply do not generate any network activity, as is the case with “PERSONAL.STORAGE” in Table A.3. Something worth mentioning is that, since we’re interested in Location and Contacts information, dropping the “PERSONAL.CONTACTS” data type would decrease our scope and make us lose some of the value from dynamic data tracking for the contacts information. For that reason, we merged “PERSONAL.CONTACTS” with “PERSONAL.ACCOUNT” as we identified that most of the contact information would be emails and phone numbers, information which is usually used for authentication and collected for advertisement and tracking. And as such, the contact information will be classified under the “PERSONAL.ACCOUNT” type. The final label distribution can be found in Table 4.2. We can see that the majority of the labels, $\sim 57\%$, are “Unclassified”. Moreover, the remaining labels are mostly allocated into the top 5 labels, which are not equally distributed across data types. As an example, around 56% of the “Ad Data” entries are for “PHONE.DEVICE”. This problem can also be seen in “Map Navigation” and “Nearby Search”, for which, although a lot of requests are detected, most of them are sent to the same service provider, to the same endpoint, changing only the arguments sent, which translates into very few diverse examples, given that we combine similar endpoints to avoid overfitting.

With these 5 data types ready, we tested different models to see which ones performed the best. In total, we tested Maximum Entropy, Support Vector Machines (SVM), C4.5, Random Forest, and XGBoost. Maximum Entropy was the model that achieved the best results in MobiPurpose; however, given that our model has a much higher dimension in terms of number of features, XGBoost provided the best results, as it can deal with complex non-linear relationships as well as deal with $null$ values, for situations where some information might not be accessible or retrievable. Before providing the information to the model, some more pre-processing is needed; Some of this process was already covered

Label	Count
Unclassified	1985
Analytic Data	358
Interface Customization	224
Ad Tracking	218
Ad Data	210
Analytic Tracking	173
Authentication	152
Network Optimization	70
Third Party Login	26
Anti Fraud	21
Location Based Customization	16
Nearby Search	18
Map Navigation	15
Signout Personalization	4

Table 4.2: Label Frequencies in Dataset

in section 3.2.2, namely the clustering and encoding performed. However, it is important to note that, to avoid dataset contamination, i.e., leaking test data into the training process, the dataset was divided into train, test, and val sets in a specific way. This division can not use a simple stratified random subsampling on the entries, since each entry is a Key-Value Pairs (KVP) and not a request. Using the stratified random subsampling would send entries from the same request, which, if we recall, contain request-wise features, to the test, train, and val sets, thus leaking data from the test to the train set. For that reason, the division had to be done request by request, while trying to make this division as balanced as possible in terms of label distribution, which is not always possible with a low count of labels. To tackle this issue, SMOTE [103] was used to deal with the class imbalance, which works by generating synthetic samples by selecting minority class instances, finding their nearest neighbors, and creating new data points through linear interpolation between the instance and its neighbors.

We then trained our model for every possible combination of features, and for each of these combinations, we performed hyperparameter tuning with Bayesian Optimization with 60 iterations each. Initially, this might seem unnecessary, however hyperparameters dictate the relation between the features and the labels; By changing the feature set, the model needs to learn a different relationship between features that best leads to the correct label, and thus, to achieve the best performance for each combination, we do the hyperparameter tuning process for all feature sets equally.

4.3.1 Analysis

In this section, we present the results for our models of purpose classification. For each model, we will consider the top 3 best-performing feature sets, explained in Table 3.1 with the feature-to-code mapping found in in Tables 4.3 and B.1 in this section and Annex B respectively, with the total number of entries for each label, where if a tie is seen, the combination with the least amount of features in the set will have priority, as it requires fewer data points to achieve a similar result. For example, if CQT (Callstack, QueryCount, and Trackers) and LD (Libradar and Callstack Distance) are "competing" for third place and both have the same F1-score, LD is preferred over CQT. Nonetheless, the complete evaluation across all 32 feature combinations (excluding the FlowDroid features for reasons which we will cover shortly), for all data types, is provided in Annex B (Tables B.2, B.3, B.4, B.5 and B.6). Furthermore, as mentioned previously, the feature-to-code mapping can be found in Tables 4.3 and B.1 in this section and Annex B, respectively.

Feature Code	Feature Name
MobiPurpose	MobiPurpose
Call stack	C
call stack Distance	D
Libradar	L
Query Count	Q
Trackers	T
Flowdroid Count	F_C
Flowdroid Match	F_M

Table 4.3: Mapping of feature codes to feature names

The top 3 best feature sets for "ID.GENERALID" by weighted f1 score are, as shown in Table 4.4, DTL, T, and QT, which outperform the base model (MobiPurpose) by 4.6, 4.2, and 3.4 percentage points, respectively. From the 3, DTL presents the best overall results; however, T provides a very close performance while only requiring a single feature, although this difference is more significant if we consider the macro F1-score instead, due to the class imbalance. Looking at all results, Table B.2, we can see that T is the strongest predictor (as it is present in all the best combinations) among all combinations, being also the best single feature. This is expected as most of the labels are usually associated with the use of TPL. What came as a surprise is the presence of D in the best combination; A plausible explanation is the correlation between the use of IDs and the collection

of location and/or contact information within these requests, specially in “Analytic Tracking” and/or “Ad Tracking”, although this hypothesis could not be confirmed as it requires reverse engineering the application to confirm that these data types are indeed present in the requests, for the cases where the NLP model was not able to identify them. “Ad Tracking”, “Analytic tracking”, and “Authentication” are consistently well recognized, but “Anti Fraud” provides the worst results due to the class imbalance (21 entries), while the “Signed-out Personalization” label was not included as we did not find enough examples to train the model on, due to the rarity of this purpose.

Feature Set	Ad Tracking (218)			Analytic Tracking (173)			Anti Fraud (21)			Authentication (152)			M-avg	W-avg
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	F1	F1
MobiPurpose	0.872	0.810	0.840	0.839	0.788	0.812	0.500	0.500	0.500	0.789	0.909	0.845	0.749	0.827
DTL	0.902	0.881	0.892	0.964	0.818	0.885	1.000	0.500	0.667	0.775	0.939	0.849	0.823	0.873
T	0.947	0.857	0.900	0.963	0.788	0.867	0.400	1.000	0.571	0.775	0.939	0.849	0.797	0.869
QT	0.971	0.810	0.883	0.964	0.818	0.885	0.400	1.000	0.571	0.738	0.939	0.827	0.792	0.861

Table 4.4: Top 3 Performance for `ID.GENERALID`

In table 4.5 we present the three best feature sets for “PHONE.DEVICE”, having the 3-feature combination CDL on the top, followed by the near-identical CQDL, and the two-feature pair CT, which provide an improvement of 4.9, 4.9, and 4.5 percentage points, w.r.t. the baseline. Notably, looking at Table B.2, although C appears in all three top sets, it is not the best single feature; T alone achieves a higher W-avg (0.870) than C alone (0.866), and the two-feature combination CT already provides a very strong result (0.876) even though it is slightly outperformed by the best 3-feature combo (0.880). This difference becomes even smaller if we consider the macro-avg F1 score, where there is only a 0.01 improvement. Examining pairwise behavior shows most two-feature combinations that include C do not beat C by itself (e.g., CD = 0.863; CL = 0.862), making the CT pair an important exception; moving to three feature types consistently yields further gains (several 3-feature sets exceed C’s 0.866), which implies complementary information emerges only when a third, orthogonal feature is added. Adding Q is not a reliable booster for the top configurations (it often produces negligible or inconsistent changes) as we can see in the top 2 feature combinations (CDQL $\approx$ CDL), while L typically supplies small incremental improvement when paired with C and D. Finally, the largest absolute improvements versus the base model occur on the smaller “Ad Data” split (“Ad Data” F1 increases by roughly $0.12 - 0.14$); overall the pattern favors a small number of high-signal, complementary feature types over simply aggregating many weaker features.

Looking at the results for “PHONE.NETWORK” in Table 4.6, you will notice 2 things. First

Feature Set	Ad Data (118)			Analytic Data (247)			Interface Customization (224)			M-avg	W-avg
	P	R	F1	P	R	F1	P	R	F1	F1	F1
MobiPurpose	0.616	0.833	0.709	0.930	0.856	0.892	0.860	0.784	0.821	0.807	0.831
CDL	0.754	0.907	0.824	0.935	0.928	0.932	0.891	0.804	0.845	0.867	0.880
CQDL	0.754	0.907	0.824	0.928	0.928	0.928	0.901	0.804	0.850	0.867	0.880
CT	0.758	0.926	0.833	0.942	0.904	0.922	0.874	0.814	0.843	0.866	0.876

Table 4.5: Top 3 Performance for `PHONE.DEVICE`

"Network switch notification" and "Geo localization" are missing and "Ad Data" and "'Network Optimization" were joined into a single feature. The former, just like "Signed-out Personalization" is due to examples of these purposes not being found when analyzing the applications; The latter comes from one of the downsides about having a single labeler for the dataset, as will be discussed in section 4.4, which is the lack of validation/second opinions on the classification. For this specific case, in most situations, it was hard to manually infer if the network information, usually about which type of network, WiFi or data, was being used to provide media with an appropriate resolution for the advertisement, or if it was simply being collected by the advertisement service for other undisclosed reasons. This created the situation where the model was consistently failing to differentiate the two labels, something we, as a singular labeler, also had trouble doing. Thus, we decided to join both labels to see if the issue was the features not providing enough context for the task, or the possible errors in the dataset. After joining both, we saw a significant increase in the classification score for this label, showing that the problem indeed stemmed from the labeling process. This, as will be later discussed, reinforces the importance of the expansion of the current dataset, preferably by the evaluation of multiple experts, as is usually done, to tackle these problems. Nonetheless, the top 3 best feature sets for "PHONE.NETWORK", as seen in Table 4.6, are L, T, TL, with improvements of 3, 1.5, and 1.5 percentage points, respectively. These results were expected since both labels (including the union) are mostly implemented through the use of TPL. However, among these 3, L outperforms the rest while only requiring a single feature. T alone has a slightly lower performance than L, and combining the two, although still presenting a good performance overall, provides no additional improvement over using only T, and a lower performance over using only L. These two features are highly related since TPL includes trackers. In fact, multiple other feature combinations achieve the same macro and weighted-avg F1 score as T, but these require a larger number of features to achieve. Thus, for "PHONE.NETWORK" the use of L, is not only enough but also the best performing feature "set" across the board.

Feature Set	Ad Data and Network Optimization (104)			Analytic Data (71)			M-avg	W-avg
	P	R	F1	P	R	F1	F1	F1
MobiPurpose	0.882	0.698	0.779	0.581	0.818	0.679	0.729	0.745
L	0.889	0.744	0.810	0.621	0.818	0.706	0.758	0.775
T	0.909	0.698	0.789	0.594	0.864	0.704	0.747	0.760
TL	0.909	0.698	0.789	0.594	0.864	0.704	0.747	0.760

Table 4.6: Top 3 Performance for PHONE.NETWORK

"SENSOR.LOCATION" is the data type with the least amount of entries per label. This is mostly due to this data type not being seen as often, across different applications, as other data types like IDs and device information, which is to be expected, and when seen, most requests with location data are sent for the same endpoints. If we look at the distribution, in Table 4.7 we will see that "Ad Data" contains the higher label count, double of the rest of the entries. This may seem unusual, especially considering that for applications which provide "Map Navigation", "Location-Based Customization" and "Nearby Search" services, the amount of requests is extremely high. However, when looking at the latter 3 purposes, most requests flow towards the same domains consistently across applications. For example, for "Map Navigation," a lot of requests seen were either for "earth-pa.googleapis.com" or "www.google.com/maps/", which skews the results. This is one of the reasons why we cluster results based on top domains, as these purposes generate considerable traffic for the same domain. Another important note is that, "Ad Data" has a very good score across most feature sets. This is mainly due to the advertisement endpoints seen, in the majority of the cases, being obvious by the URL alone. Nonetheless, two of the top three features for this label improved this classification significantly, showing its ability to extract extra context. If we look at Table B.4, we will also note that, among all features, D was able to achieve a perfect score for "Map Navigation", which shows the promising abilities of this feature do detect such purpose. The reason behind this performance is due to, for services like "Map Navigation", the location information needs to be constantly retrieved, which makes it impossible for the application to store it somewhere else and use it later, as at that point it will become outdated. That forces the application to essentially retrieve the information when it needs to send it, a situation in which D excels. Nonetheless, the improvement over the base model with the feature sets CQDT, CDT, and CTL for "SENSOR.LOCATION" is significant, both for macro and weighted f1-avg score ($1.6 - 6.1$ and $10.6 - 15.5$ percentage points, respectively); however, we need to consider that among all data types, "SENSOR.LOCATION" presents the lowest amount of entries.

Feature Set	Ad Data (31)			Analytic Data (15)			Location-based Customization (16)			Map Navigation (15)			Nearby Search (18)			M-avg	W-avg
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	F1	F1
MobiPurpose	0.857	1.000	0.923	0.429	1.000	0.600	0.833	0.625	0.714	0.800	1.000	0.889	1.000	0.200	0.333	0.792	0.701
CQDT	0.923	1.000	0.960	0.857	1.000	0.923	1.000	0.875	0.933	1.000	0.500	0.667	0.692	0.900	0.783	0.853	0.856
CDT	0.857	1.000	0.923	0.857	1.000	0.923	0.875	0.875	0.875	1.000	0.500	0.667	0.692	0.900	0.783	0.834	0.836
CTL	0.923	1.000	0.960	0.857	1.000	0.923	1.000	0.625	0.769	1.000	0.500	0.667	0.600	0.900	0.720	0.808	0.812

Table 4.7: Top 3 Performance for `SENSOR.LOCATION`

For “PERSONAL.ACCOUNT”, the results in Table 4.8, show that the top 3 best feature sets, CT, QT, and TL, all have the same performance (macro = 0.820 and weighted = 0.837), which provides an improvement of 13.1 percentage points. In reality, if we look at Table B.6, we will see that a lot more feature sets achieve the same score, and all of them, except CQD, utilize T to better discriminate the labels. This shows that T is a really good feature for “PERSONAL.ACCOUNT” as well. It’s also important to note that T by itself (macro = 0.719 and weighted = 0.732) underperforms when compared to its combination with other features, showing that although a very strong feature for this data type, it greatly benefits from the signals acquired from the other features, mainly C, Q, and L.

Feature Set	Ad Data (27)			Analytic Data (25)			Thirdparty Login (26)			M-avg	W-avg
	P	R	F1	P	R	F1	P	R	F1	F1	F1
MobiPurpose	0.400	0.857	0.545	0.900	0.562	0.692	0.900	0.750	0.818	0.685	0.706
CT	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
QT	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
TL	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837

Table 4.8: Top 3 Performance for `PERSONAL.ACCOUNT`

Now, if we recall Table 3.1, two features are missing from the previous analysis, which are Flowdroid Count (F_C) and Flowdroid Match (F_M). The reason why these 2 features were not included is mainly due to not having enough entries to train the model on. The reason as to why will be discussed in more detail in Section 4.4. Nonetheless, we decided to train the models with only the apps for which FlowDroid was able to get results, and attached the results on Annex B, in the Tables B.7, B.8, B.9 and B.10. However, instead of training for all the possible combinations we decided, due to the inherent limitations these 2 features contain and because doing so would increase the total feature combination, per data type, to 128, to add FlowDroid Count (F_C) and FlowDroid Match (F_M) on top of the previously best performing feature sets to see if there were any improvements.

For “ID.GENERALID”, Table B.7 we see an increase for QT and DTL when using both features, but F_M provided the bigger increase (0.024 and 0.011). Meanwhile, for T, both F_C and F_M, and their combination, worsen the classification. With “PHONE.DEVICE”, Table B.8 a similar behavior was seen for CQDL, where F_M provided a small improvement

of 0.007, while the other two feature sets, CDL and CT, only worsened with the addition of these features. For “SENSOR.LOCATION”, due to most applications that contained these traces not being able to be analyzed by FlowDroid, we did not have enough entries to train the model on, so no results were retrieved for this data type. For “PHONE.NETWORK”, Table B.9, only T saw an improvement in score with the two new features. F_C seemed to provide no improvement by it self while F_M provides an increase of 0.038. However, the combination of both of these features bumps that improvement up to 0.078, indicating that these two features provide complementary signals to each other. Finally, for “PERSONAL.ACCOUNT”, Table B.10, we see no improvement across all feature sets. This data type, apart from “SENSOR.LOCATION”, is the most affected by the number of available entries, which will also affect these results.

These results demonstrate the advantage of the data combination we proposed, acquired through the use of our tool, LeakHound, to produce a richer context around network requests. By producing this context, we were able to increase our ability to classify a general data collection purpose taxonomy for Android applications, which outperforms the state of the art. Furthermore, through this analysis, we are also able to identify which feature combinations perform the best for each data type analyzed, which provides the ideal and necessary data needed to classify each data type, thus reducing the complexity of each model. This complexity optimization facilitates the introduction of future context, which then allows the expansion of the purpose classification taxonomy.

4.4. Limitations

In this section, we will go over some of the potential issues that we identified in our work that might influence the validity of our results. This is an important topic to cover, as understanding these limitations will help future work in improving the classification model’s performance for the task of purpose classification.

As mentioned in the previous section, in the end, we did not have enough entries to train our model on F_C (FlowDroid Count) and F_M (FlowDroid Match). Initially, this may seem confusing, as we had enough entries to train the model on for the rest of the features; however, these two are entirely dependent on information retrieved from FlowDroid. As discussed previously in section 3.3, FlowDroid will often not work properly. In our case, only 37% of the apps had results from FlowDroid, which means that for the remaining 63% of the apps, FlowDroid either crashed during execution, reached the maximum amount of RAM provided (50 GB) and was unable to detect flows or reached the set timeout for

applications (2h) and with the information collected and processed within that window of time, was not able to find a leak. To make things worse, since FlowDroid is a static taint analysis tool, a lot of the findings will be False Positives, as FlowDroid will consider all paths, including those that in reality will not happen, such as trying to guess the order in which threads will execute, or dealing with dynamically loaded code. This resulted in a very small sample of entries that actually had some results. Initially, we attempted to train the model with the information we had, including entries for which the FlowDroid was unable to provide results, but we quickly reached the conclusion that the value of those results was too insignificant, as most entries would contain a null value for both features.

One other limitation of this work stems from the data collection and acquisition process. The quality and completeness of the data used can significantly influence the reliability and how well the results can be generalized. In this work, several challenges were encountered during the data collection phase, especially when it came to runtime information acquisition. As covered in section 3.2.1.3, to acquire the runtime data, such as call stack, location, and content queries, etc., we used Frida. This created some shortcomings; Less frequently, some applications would detect the use of Frida Hooks and crash themselves or block any progress. However, the most considerable issue is the fact that Frida hooks take some time to attach, leading to gaps in app monitoring. This is even more relevant when we consider that sometimes DroidBot will crash the application unintentionally, and therefore, the hooks need to be reattached. This can make us lose some hook calls, which initially may not seem that critical; however, we need to consider the call stack reconstruction process. As explained in section 3.2.1.3, to reconstruct the call stack to a sink, we use Threads. We start at the last thread created, and start going back through the graph until we reach the main thread, at which point we connect the edges and form the entire call stack. Now, if one of these edges is missing due to gaps in the analysis, then all subsequent, from bottom up, calls are essentially lost, leading to an incomplete call stack. Removing all entries in which this happens would significantly reduce the number of entries we have to train our model, since we don't have that many to begin with, and given that we need to manually classify these labels, as there is no available dataset for our goal, doing this is not really a possibility. Instead, we relied on "partial" callstack reconstruction, which essentially is just reconstructing up until the last known thread. This will only be a problem when the amount of threads lost, in relation to the total amount of threads in the path, is very low, for example, if we only have the first Thread (and by first, we mean in the order of reconstruction, from bottom up, therefore

the last), in a lot of cases we will only get the information about framework methods used to perform the communication. We also run into a similar issue with FlowDroid and Exodus Privacy. FlowDroid, although one of the most commonly used Static Taint Analysis tools, still has a lot of shortcomings. One of its major problems is that sometimes it cannot detect paths between sources and sinks, due to hitting resource limits, although we can see the data in the requests, as well as crashing for some applications, leaving us with an incomplete dataset when it comes to that information. The lack of flow detection can, in some cases, be associated with a lack of RAM. We also looked at alternatives for FlowDroid such as Proguard [89], however, proguard-core, the open-source version, was created with Java bytecode in mind and not necessarily the Android environment, and thus it has some trouble adapting to having multiple entry points in an application. For that reason, we decided to stay with FlowDroid. Still in the realm of static analysis, Exodus Privacy, although less common, will sometimes be missing information on some applications. This is less important as we only noticed about 4-5 applications missing, but it can influence results.

Finally, the last, and arguably most significant, threat to validity in this work is the dataset size. As mentioned previously, there is no available dataset containing the information that we need, and mapping the information back to an existing dataset is unreliable at best. For that reason, we had to create our own dataset, but that comes with its share of problems. Since only one person is acquiring and classifying entries, it means that not only can the results be skewed/biased, but also the number of entries ultimately acquired will be very low. Coupled with that, a lot of the entries will contain no personal/dangerous/sensitive information, something we expected, and thus, will be of no use for the models. Furthermore, from these collected labels, some behaviors will be more frequent than others. To worsen the distribution of labels, since clustering of similar endpoints was applied to avoid overfitting the model towards domains that will appear more than others, a lot of different entries are collapsed into one, something which heavily affected “Map Navigation” and “Nearby Search”. For that reason, only a small percentage of the actual dataset will be used by the model, increasing the need for a considerably bigger dataset. A smaller dataset will not allow a model to generalize as well when compared with a more robust one. This is a significant issue in any problem that relies on machine learning, and we are no exception; and for that reason, it’s necessary to consider this when looking at the results.

5. Conclusions and Future Work

Most Android users provide access to sensitive information to applications, without completely understanding the purpose behind this data collection. This is mostly due to a lack of a purpose-centric permission manager in Android, since application, and TPL developers are not forced, nor incentivized, to provide clear and unambiguous reasoning behind the data collection. This is then combined with the fact that permission requests are performed singularly and centrally, meaning that the user will face a situation of "accept everything or nothing at all". This highlights the need to introduce purpose inference mechanisms for Android applications to better equip users with the knowledge to make privacy-preserving decisions.

When it comes to purpose classification in Android, most works that focus on network analysis either collect only the network information without further context or, when they do collect it, they do it with a more tailored and hyper-specific goal, such as privacy policy validation or ATS detection. Therefore, the context surrounding network requests for a general-purpose classification remains a relatively under-explored topic. For that reason, in this work, we focused on acquiring multiple sources of context to understand which data points, across the different data types, better help in this purpose classification task.

To acquire this information, we developed LeakHound, a tool designed to collect network requests and information surrounding said requests in an effort to map and correlate them to provide a richer context used to classify the purpose of the data transmission. Our tool is divided into two phases: dynamic and static. In the dynamic phase, LeakHound will intercept all requests made by the application and, by employing a Man-in-the-Middle (MitM) and DroidBot [81] setup, retrieve the contents of said requests. Together with this information, LeakHound, by leveraging Frida [19], will retrieve all call stacks that lead to the execution of the network request, along with the call stacks that originated a retrieval of location and contact information, and reconstruct the entire call stack across different threads by hooking the threads involved in the process. With these call stacks, LeakHound will correlate network requests with the retrieval of sensitive information by looking at the "distance" between the two executions, that is, how correlated a network request and the retrieval are. Finally, LeakHound will also keep a record of all sensitive information retrieved during the application's execution, to search for that information and its encoded version in the network requests, as well as the number of times that sensitive

data was retrieved by the application. In the static phase, LeakHound will run the applications through an APK analysis with LibRadar [78] to extract the TPL and their name within this instance of the application (which can change due to obfuscation), as well as running FlowDroid [26] to detect any data leaks with taint analysis. At the same time, it will query a public tracker database, Exodus Privacy [98], for possible trackers within this application. All this information is then used to train a ML model, XGBoost, to classify the purpose behind data collection in Android applications. Due to the lack of an available dataset and the technical difficulties in gathering this data, we make publicly available[1] both our tool LeakHound and our dataset.

In summary, focusing on a compact set of discriminative context features substantially improves purpose inference across multiple data types. For that, we consider the data types "ID.GENERALID", "PHONE.DEVICE", "PHONE.NETWORK", "PERSONAL.ACCOUNT", "SENSOR.LOCATION". We used these data types since, as covered in Section 4.3, these were the ones we were able to find enough examples to train our model on. Nonetheless, the purposes that these data types can be classified with can be seen in the Tables A.1, A.2 (for "PHONE.DEVICE" and "PHONE.NETWORK"), A.3, and A.4, respectively. For "ID.GENERALID", the DTL (Call stack Distance, Trackers and Libradar) combination gives the largest increase over the MobiPurpose baseline (4.6 percentage points weighted F1), with single-feature T (Trackers) nearly matching performance (4.2 points) but showing larger gaps on macro F1 due to class imbalance. For "PHONE.DEVICE", three-feature sets CDL (Call Stack, Call Stack Distance, and LibRadar) and CQDL (Call Stack, Query Count, Call Stack Distance, and LibRadar) achieve the best weighted F1 (0.880), which provides an improvement of 4.9 percentage points while the compact pair CT (Call Stack and Trackers) attains nearly identical performance (0.876), indicating that complementary orthogonal features are especially valuable. "PHONE.NETWORK" provides the smallest improvement of all data types with 3 percentage points' improvement over MobiPurpose with the L (LibRadar) feature, with T (Trackers) close behind. "SENSOR.LOCATION" shows the largest proportional gains versus baseline (macro F1 increases 6.1 and weighted 15.5 percentage points), though results must be interpreted cautiously because this data type has the fewest examples and is dominated by a few domains. For "PERSONAL.ACCOUNT" several feature sets tie at macro 0.820 / weighted 0.837, highlighting T's (Trackers) central role but also the benefit of combining it with C (Call Stack), Q (Query Count), or L (LibRadar). This is also shown by the increase of 13.1 percentage points for

[1] https://github.com/LeoPer02/LeakHound

all three best features, CT (Call Stack and Trackers), QT (Query Count and Trackers), and TL (Trackers and Libradar). Finally, although promising in theory, the FlowDroid features, F_C (FlowDroid Count) and F_M (FlowDroid Match), suffer from a lack of entries due to issues presented by the use of FlowDroid, especially for smaller datasets. From these results, the only pattern we were able to discern is that when the improvements happen, it is mostly because of F_M (FlowDroid Match), except for "PHONE.NETWORK" on which its combination with F_C (FlowDroid Count) is better than just F_M (FlowDroid Match). This could indicate value to be extracted from the comparison of call stacks with FlowDroid leaks; however, this conclusion is restricted due to the limited size of the dataset for these features in particular. Overall, this provides an average improvement of weighted f1-avg of 8.22 percentage points over MobiPurpose.

Altogether, these results highlight the importance of combining complementary features to improve the context surrounding network requests in the task of purpose classification on a more generic taxonomy, therefore empowering classification performance, especially for underrepresented or ambiguous classes. In particular, the use of textual features from the call stack (C) which lead to the network request, the correlation between the invocation of a network request and the retrieval of sensitive information (D), app-wise information such as number of sensitive data retrieval occurrences (Q), data leaks (F_C) and trackers (T), as well as the matching of data leaks (F_M) and TPL (L) with the call stack. Moreover, carefully designed feature sets are crucial for achieving balanced and generalizable results. Looking at the results, we can see the importance of per-data-type context selection, as different features, as well as their combinations, benefit the data types differently. In particular, "ID.GENERALID" benefits the most from the combination DTL, "PHONE.DEVICE" from CDL, "PHONE.NETWORK" from L, "SENSOR.LOCATION" CQDT and "PERSONAL.ACCOUNT" CT. This per-data type selection is an important consideration, as expanding the purpose classification taxonomy will inevitably require the addition of new features, which would gradually increase the model's complexity. Through this study, we were able to identify which features are the backbone for each of the data types, therefore reducing that added complexity. Nonetheless, some potential improvements, which are outlined in Section 5.1, could be pursued in future work.

5.1. Future Work

With this work, we aimed at increasing the context surrounding each network request in an effort to increase the classification capability of purpose-inferencing models. Although

we achieve relevant results, taking into account the limitations covered in Sections 3.3 and 4.4, there are still improvements that could be made to increase the accuracy and new fronts that could not be explored in this work.

One of the issues regarding our work is the time to attach from Frida, which sometimes leads to the loss of data. This could be addressed by the incorporation of Frida Gadgets into the application itself in an effort to reduce the time taken for API hooking and thus increase the coverage of our approach. Another point of improvement would be in data type identification; Currently, we utilize an NLP model to infer the data type present within a network request body. This model attempts to identify information related mostly to the user device and some personal data as well. Further improvement could be made by the integration of PII detection, such as in the work from Kohli et al. [50], in the body of the request, to further increase the detection of data leakage within the requests. We also think it would prove beneficial to look at the data leakage association in this work; We currently propose a mechanism to detect correlation between requests and the socket call, as well as mapping static taint analysis leaks to the socket invocation. However, there are ways to bypass this if the developer intends to. For that end, investing in supplementary mechanisms for data correlation would increase our coverage in data leakage detection, and thus improve the inference capabilities of our model. In this work, we focused the analysis on location and contact information, and thus, there are still different types of information to cover. As mentioned already, looking into PII would prove beneficial, although it would require an expansion of the taxonomy used, and beyond that, it would also be worth looking into extending this analysis for the other dangerous permissions. Location and contacts are by far the ones most users are concerned about, as covered in [70, 71]; however, access to the microphone, camera, SMS, etc., poses significant threats to the user's privacy. Yet, these permissions are not as straightforward to monitor as the other ones, since usually they are not a simple value stored in a variable. For the microphone and camera, the collection can happen in a stream of information or by storing it in files and sending it later. For SMS, these usually appear as interceptors or listeners, which handle the SMS receivers through broadcasts, making it more difficult to trace.

Moreover, the entirety of the purpose classification taxonomy was not covered. The reason, as discussed previously, is due to the imbalance of frequency for these purposes. However, another issue arises from the incorporation of further purposes into the classification model. Each purpose, excluding those that do not generate any network activity, such as "Access photo album" in "PERSONAL.STORAGE" in Table A.3, will have its own

unique context. For example, looking at table A.4 “Voice Authentication” will require the transmission of audio to an external service, which will validate it and return an authentication token. However, by simply looking at the information sent but not the received, one might classify this request as “Audio streaming” as the response context will be missing. For that reason, to expand the current set of labels to include more purposes, we will require an assessment, at an individual level, to understand what extra context is needed for the classification to be reliable.

From the deployment perspective, our current implementation serves well for research purposes but lacks the refinement to be deployed in a production scenario. One of the possible improvements in this area would be the implementation of privacy-preserving network analysis tools, opposed to our approach of mitmproxy [61] with PCAPdroid [74]. An example of such a tool is discussed in Djob and Bromberg [72]. If possible, it would also be worth looking into migrating this analysis into the device itself, so that this analysis can be applied to apps running on the user’s device, while the users utilize the app daily. This change would require refinement not only on the resource consumption front, but also the integration of mechanisms as previously proposed to better safeguard user privacy when it comes to network analysis. Nevertheless, moving this analysis into a consumer device could help trace an application profile giving users a better understanding of what exactly is being done with their data, as well as bypassing one of the major problems on dynamic analysis, as discussed in Section 2.1.2.1 (on the dynamic taint analysis section), coverage; Having the user navigate the application as he normally does, provides a better coverage or, in the cases where this coverage is limited, it covers all requests generated during the user’s usage of the application, which is enough. These aspects were, however, beyond the scope of this thesis.

Furthermore, as covered previously in section 2.2.2, comparisons with other works are quite limited in the field of purpose classification. In most cases, to be able to perform a comparison, one must not only reconstruct the entire pipeline of previous works, which is not always made publicly available, but also modify the purpose classification taxonomy used in either their own work or the one they are comparing with, which could pose some problems when making statements about said comparisons. To validate our results, our evaluation model was designed as similar to Mobipurpose [55] as possible, so that in the end we could compare both approaches. Nonetheless, we do feel that further comparisons would benefit our work in terms of validating/invalidating our findings, as well as providing further insight into what methodologies are better suited to which type

of purpose.

Lastly, which is also related to some of the points raised in this section, a major improvement would be the expansion of the dataset. As also covered in section 2.2.2, since there are no available datasets that could be used in this work, a new one had to be created and manually labeled. This significantly constrains the number of entries that can be acquired within a reasonable timeline. To further complicate this issue, most network requests will contain no sensitive data and therefore will not be used for training. The labels will be heavily imbalanced across different data types, FlowDroid will fail to analyze most applications and be unable to detect flows due to RAM limitations or over-flag flows, which in reality do not happen. This significantly reduces the number of usable entries in the dataset that could be used for the model training, and as such, further work should be allocated towards expanding this dataset so that every label is sufficiently represented. It's also important for multiple experts to be involved in this classification, not only because it allows them to cover more ground, but also because it can introduce a validation process to avoid situations where labels cannot be distinguished, as was the case with "PHONE.NETWORK" or the introduction of consistent errors by the misclassifications of one expert. This improvement of the dataset would also allow the expansion of the taxonomy, both by covering the purposes which we did not detect, but also allow the inclusion of PII detection and their associated purposes.

References

[1] X. Liu, Y. Leng, W. Yang, W. Wang, C. Zhai, and T. Xie, "A large-scale empirical study on android runtime-permission rationale messages," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2018, pp. 137–146. [Cited on page 1.]

[2] A. Girish, J. Reardon, J. Tapiador, S. Matic, and N. Vallina□Rodriguez, "Your signal, their data: An empirical privacy analysis of wireless-scanning sdks in android," *Proceedings on Privacy Enhancing Technologies (PoPETs)*, vol. 2025, no. 3, pp. 361–381, 2025. [Online]. Available: https://petsymposium.org/popets/2025/popets-2025-0103.php [Cited on page 1.]

[3] R. Mendes, A. Brandão, J. P. Vilela, and A. R. Beresford, "Effect of user expectation on mobile app privacy: a field study," in *2022 IEEE international conference on pervasive computing and communications (PerCom)*. IEEE, 2022, pp. 207–214. [Cited on pages 1 and 5.]

[4] B. Liu, M. S. Andersen, F. Schaub, H. Almuhimedi, S. A. Zhang, N. Sadeh, Y. Agarwal, and A. Acquisti, "Follow my recommendations: A personalized privacy assistant for mobile app permissions," in *Twelfth symposium on usable privacy and security (SOUPS 2016)*, 2016, pp. 27–41. [Cited on page 1.]

[5] D. Smullen, Y. Feng, S. A. Zhang, and N. Sadeh, "The best of both worlds: Mitigating trade-offs between accuracy and user burden in capturing mobile app privacy preferences," *Proceedings on Privacy Enhancing Technologies*, 2020. [Cited on page 2.]

[6] "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)," Official Journal of the European Union, L 119, 4 May 2016, pp. 1–88, 2016, accessed: 2025-08-21. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2016/679/oj [Cited on page 2.]

[7] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin, "A first look at traffic on smartphones," in *Proceedings of the 10th ACM SIGCOMM*

Conference on Internet Measurement, ser. IMC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 281–287. [Online]. Available: https://doi.org/10.1145/1879141.1879176 [Cited on pages 3, 11, 25, and 39.]

[8] J. Reardon, Á. Feal, P. Wijesekera, A. E. B. On, N. Vallina-Rodriguez, and S. Egelman, "50 ways to leak your data: An exploration of apps' circumvention of the android permissions system," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 603–620. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/reardon [Cited on pages 3, 25, 39, and 60.]

[9] N. P. Borges Jr., "Data flow oriented ui testing: exploiting data flows and ui elements to test android applications," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 432–435. [Online]. Available: https://doi.org/10.1145/3092703.3098234 [Cited on page 7.]

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 258–261.

[11] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 641–660.

[12] P. Carter, C. Mulliner, M. Lindorfer, W. Robertson, and E. Kirda, "Curiousdroid: automated user interface interaction for android application analysis sandboxes," in *Financial Cryptography and Data Security: 20th International Conference, FC 2016, Christ Church, Barbados, February 22–26, 2016, Revised Selected Papers 20*. Springer, 2017, pp. 231–249.

[13] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, 2014, pp. 204–217.

[14] A. Machiry, R. Tahiliani, and M. Naik, “Dynodroid: An input generation system for android apps,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224–234.

[15] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, “Guided, stochastic model-based gui testing of android apps,” in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 245–256.

[16] M. Y. Wong and D. Lie, “Intellidroid: a targeted input generator for the dynamic analysis of android malware.” in *NDSS*, vol. 16, no. 2016, 2016, pp. 21–24.

[17] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, “Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications,” in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, 2012, pp. 93–104. [Cited on page 7.]

[18] 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. [Cited on page 8.]

[19] “Frida - dynamic instrumentation toolkit,” https://frida.re/, accessed: 2025-04-15. [Cited on pages 9, 12, 13, 14, 21, 46, and 78.]

[20] “Xposed/lsposed framework,” https://github.com/LSPosed/LSPosed, accessed: 2025-04-15. [Cited on page 9.]

[21] “Detours - microsoft research,” https://github.com/microsoft/Detours, accessed: 2025-04-15. [Cited on page 9.]

[22] “Intel pin - a dynamic binary instrumentation tool,” https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html, accessed: 2025-04-15. [Cited on page 9.]

[23] Guardsquare, “Dexguard - android app security and obfuscation,” https://www.guardsquare.com/dexguard, accessed: 2025-04-15. [Cited on page 9.]

[24] Appdome Inc., “Appdome - mobile app security and integration platform,” https://www.appdome.com/, accessed: 2025-04-15. [Cited on page 9.]

[25] INKA Entworks Inc., “Appsealing - mobile app protection,” https://www.appsealing.com/, accessed: 2025-04-15. [Cited on page 9.]

[26] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM sigplan notices*, vol. 49, no. 6, pp. 259–269, 2014. [Cited on pages 10, 21, 26, 56, 61, 63, 66, and 79.]

[27] S. Arzt, "Static data flow analysis for android applications," Ph.D. dissertation, Technische Universität Darmstadt, Darmstadt, 2017. [Online]. Available: http://tuprints.ulb.tu-darmstadt.de/5937/ [Cited on page 10.]

[28] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?" in *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 331–341. [Cited on page 10.]

[29] 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 *Network and Distributed System Security (NDSS) Symposium*, vol. 15, no. 201, 2015, p. 110. [Cited on page 10.]

[30] 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 *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 280–291. [Cited on page 10.]

[31] 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. [Cited on page 10.]

[32] 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 Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 1–29, 2014. [Cited on pages 10, 16, 21, 26, 31, 33, and 63.]

[33] O. Schranz, "Artist: A novel instrumentation framework for reversing and analyzing android apps and the middleware," in *Black Hat USA*, 2018, conference presentation, Las Vegas, NV, USA. [Online]. Available: https://i.blackhat.com/us-18/Thu-August-9/us-18-Schranz-ARTist-A-Novel-Instrumentation-Framework-

for-Reversing-and-Analyzing-Android-Apps-and-the-Middleware-wp.pdf [Cited on page 11.]

[34] M. Sun, T. Wei, and J. C. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 331–342. [Cited on page 11.]

[35] L. K. Yan and H. Yin, "{DroidScope}: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis," in *21st USENIX security symposium (USENIX security 12)*, 2012, pp. 569–584. [Cited on page 11.]

[36] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors." in *Ndss*, 2015, pp. 1–15. [Cited on page 11.]

[37] K. Ahmed, Y. Wang, M. Lis, and J. Rubin, "Vialin: Path-aware dynamic taint analysis for android," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1598–1610. [Cited on pages 11, 34, and 63.]

[38] S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao, Z. Liu, F. Xu *et al.*, "Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2421–2436. [Cited on page 11.]

[39] H. Fu, Z. Zheng, S. Zhu, and P. Mohapatra, "Keeping context in mind: Automating mobile app access control with user interface inspection," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 2089–2097. [Cited on pages 11, 21, and 26.]

[40] H. Fu, P. Hu, Z. Zheng, A. K. Das, P. H. Pathak, T. Gu, S. Zhu, and P. Mohapatra, "Towards automatic detection of nonfunctional sensitive transmissions in mobile applications," *IEEE Transactions on Mobile Computing*, vol. 20, no. 10, pp. 3066–3080, 2020. [Cited on pages vii, 11, 20, 21, 26, and 32.]

[41] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-Resilient Privacy Leak Detection for Mobile Apps Through

Differential Analysis," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017. [Cited on page 12.]

[42] X. Wang, A. Continella, Y. Yang, Y. He, and S. Zhu, "Leakdoctor: Toward automatically diagnosing privacy leaks in mobile applications," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 3, no. 1, pp. 1–25, 2019. [Cited on pages 12, 16, and 21.]

[43] A. Shuba, A. Markopoulou, and Z. Shafiq, "Nomoads: Effective and efficient cross-app mobile ad-blocking," *Proceedings on Privacy Enhancing Technologies*, 2018. [Cited on pages 12, 17, 20, and 21.]

[44] EasyList Team, "easylist: Adblock filter lists for blocking ads and tracking," https://github.com/easylist/easylist, 2025, accessed: 2025-06-29. [Cited on pages 12 and 21.]

[45] Eyeo GmbH, "Adblock plus - surf the web without annoying ads," https://adblockplus.org/, 2025, accessed: 2025-06-27. [Cited on pages 12 and 21.]

[46] A. Shuba, A. Le, E. Alimpertis, M. Gjoka, and A. Markopoulou, "Antmonitor: A system for on-device mobile network monitoring and its applications," *arXiv preprint arXiv:1611.04268*, 2016. [Cited on pages 12, 14, and 21.]

[47] A. Shuba and A. Markopoulou, "Nomoats: Towards automatic detection of mobile tracking," *Proceedings on Privacy Enhancing Technologies*, 2020. [Cited on pages 12, 17, 20, 21, and 39.]

[48] H. Wang, Z. Liu, J. Liang, N. Vallina-Rodriguez, Y. Guo, L. Li, J. Tapiador, J. Cao, and G. Xu, "Beyond google play: A large-scale comparative study of chinese android app markets," in *Proceedings of the Internet Measurement Conference 2018*, 2018, pp. 293–307. [Cited on pages 12 and 21.]

[49] C. Su, K. Sakurai, and F. Liu, *Science of Cyber Security: 4th International Conference, SciSec 2022, Matsue, Japan, August 10–12, 2022, Revised Selected Papers*. Springer Nature, 2022, vol. 13580. [Cited on pages 12, 17, 20, 21, and 32.]

[50] R. Kohli, S. Gupta, and M. S. Gaur, "End-to-end triplet loss based fine-tuning for network embedding in effective pii detection," *arXiv preprint arXiv:2502.09002*, 2025. [Cited on pages 13, 18, 20, 21, and 81.]

[51] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, 2019, pp. 4171–4186. [Cited on pages 13 and 21.]

[52] H. Tang, Y. Cui, J. Wu, X. Yang, and Z. Yang, “Trigger relationship aware mobile traffic classification,” in *Proceedings of the International Symposium on Quality of Service*, 2019, pp. 1–10. [Cited on pages 13 and 21.]

[53] Y. Xiao, Z. Li, Y. Qin, X. Bai, J. Guan, X. Liao, and L. Xing, “Lalaine: Measuring and characterizing {Non-Compliance} of apple privacy labels,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1091–1108. [Cited on pages 13, 16, 17, 18, 20, 21, and 39.]

[54] D. Bui, Y. Yao, K. G. Shin, J.-M. Choi, and J. Shin, “Consistency analysis of data-usage purposes in mobile apps,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2824–2843. [Cited on pages 13, 17, 18, 20, and 21.]

[55] H. Jin, M. Liu, K. Dodhia, Y. Li, G. Srivastava, M. Fredrikson, Y. Agarwal, and J. I. Hong, “Why are they collecting my data? inferring the purposes of network traffic in mobile apps,” *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 2, no. 4, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3287051 [Cited on pages vii, 13, 17, 18, 20, 21, 23, 24, 30, 39, 40, 63, 82, 96, 97, 98, and 99.]

[56] B. Andow, S. Y. Mahmud, J. Whitaker, W. Enck, B. Reaves, K. Singh, and S. Egelman, “Actions speak louder than words:{Entity-Sensitive} privacy policy and data flow analysis with {PoliCheck},” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 985–1002. [Cited on pages 13, 14, and 21.]

[57] B. Andow, S. Y. Mahmud, W. Wang, J. Whitaker, W. Enck, B. Reaves, K. Singh, and T. Xie, “{PolicyLint}: investigating internal privacy policy contradictions on google play,” in *28th USENIX security symposium (USENIX security 19)*, 2019, pp. 585–602. [Cited on pages 13, 14, and 21.]

[58] H. Harkous, K. Fawaz, R. Lebret, F. Schaub, K. G. Shin, and K. Aberer, “Polisis: Automated analysis and presentation of privacy policies using deep learning,” in *27th*

USENIX Security Symposium (USENIX Security 18), 2018, pp. 531–548. [Cited on pages 13, 14, and 21.]

[59] R. Trimananda, H. Le, H. Cui, J. T. Ho, A. Shuba, and A. Markopoulou, "{OVRseen}: Auditing network traffic and privacy policies in oculus {VR}," in *31st USENIX security symposium (USENIX security 22)*, 2022, pp. 3789–3806. [Cited on pages 13, 17, 18, 20, and 21.]

[60] H. Liu, "Safeguarding user security and privacy: seeing through communications and obfuscating network activities," 2023. [Cited on pages 14, 17, 20, and 21.]

[61] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors, "mitmproxy: A free and open source interactive HTTPS proxy," 2010–, [Version 12.0]. [Online]. Available: https://mitmproxy.org/ [Cited on pages 14, 21, 42, and 82.]

[62] H. Liu, P. Patras, and D. J. Leith, "Android mobile os snooping by samsung, xiaomi, huawei and realme handsets," *Trinity College Dublin, Tech. Report*, 2021. [Cited on pages 14, 17, 20, and 21.]

[63] ElderDrivers, "Edxposed framework," https://github.com/ElderDrivers/EdXposed, 2019, [Online; accessed 16-October-2024]. [Cited on pages 14 and 21.]

[64] RikkaApps, "Riru," https://github.com/RikkaApps/Riru, 2025, accessed: 2025-06-19. [Cited on pages 14 and 21.]

[65] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1354–1365. [Online]. Available: https://doi.org/10.1145/2660267.2660287 [Cited on page 14.]

[66] J. Feichtner and S. Gruber, "Understanding privacy awareness in android app descriptions using deep learning," in *Proceedings of the tenth ACM conference on data and application security and privacy*, 2020, pp. 203–214. [Cited on page 14.]

[67] MiniApps Working Group. (2022, Jul.) Miniapp standardization white paper, version 2. World Wide Web Consortium (W3C). [Online]. Available: https://www.w3.org/TR/mini-app-white-paper/ [Cited on pages 15 and 37.]

[68] W. Chao, Z. Yue, and L. Zhiqiang, “Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis.” ICSE. [Cited on page 15.]

[69] S. Wang, Y. Li, K. Wang, Y. Liu, H. Li, Y. Liu, and H. Wang, “Miniscope: Automated ui exploration and privacy inconsistency detection of miniapps via two-phase iterative hybrid analysis,” *ACM Transactions on Software Engineering and Methodology*, 2024. [Cited on page 15.]

[70] H. Wang, J. Hong, and Y. Guo, “Using text mining to infer the purpose of permission use in mobile apps,” in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ser. UbiComp ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1107–1118. [Online]. Available: https://doi.org/10.1145/2750858.2805833 [Cited on pages v, vii, 16, 20, 21, 22, 28, 29, 30, 33, 40, and 81.]

[71] H. Wang, Y. Li, Y. Guo, Y. Agarwal, and J. I. Hong, “Understanding the purpose of permission use in mobile apps,” *ACM Transactions on Information Systems (TOIS)*, vol. 35, no. 4, pp. 1–40, 2017. [Cited on pages vii, 16, 17, 20, 21, 22, 28, 29, 30, 33, 40, and 81.]

[72] D. B. T. M. Djob and Y.-D. Bromberg, “Secure access to network data for mobile network traffic analysis applications,” in *55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2025)*, 2025. [Cited on pages 16 and 82.]

[73] O. Figueira, R. Trimananda, A. Markopoulou, and S. Jordan, “Diffaudit: Auditing privacy practices of online services for children and adolescents,” in *Proceedings of the 2024 ACM on Internet Measurement Conference*, 2024, pp. 488–504. [Cited on pages 17, 20, and 21.]

[74] Emanuele, “Pcapdroid: No-root network monitor, firewall and pcap dumper for android,” 2025, accessed: April 3, 2025. [Online]. Available: https://github.com/emanuele-f/PCAPdroid [Cited on pages 21, 42, 44, and 82.]

[75] G. Combs and the Wireshark Team, “Wireshark: Network protocol analyzer,” Software, 1998, available at https://www.wireshark.org/. Original author: Gerald Combs; first released in 1998. [Cited on page 21.]

[76] O. Lhoták and L. Hendren, “Scaling java points-to analysis using s park,” in *Compiler Construction: 12th International Conference, CC 2003 Held as Part of the Joint*

European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings 12. Springer, 2003, pp. 153–169. [Cited on page 21.]

[77] Android Developers, "Monkeyrunner," https://developer.android.com/studio/test/monkeyrunner, 2023, last updated April 12, 2023, accessed June 24, 2025. [Cited on pages 21 and 44.]

[78] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 653–656. [Online]. Available: https://doi.org/10.1145/2889160.2889178 [Cited on pages 21, 29, 57, 58, and 79.]

[79] M. Team, "Nosmoke: A cross-platform ui crawler for automated smoke testing," GitHub repository, 2022, https://github.com/macacajs/NoSmoke (archived August 25, 2022). [Cited on page 21.]

[80] E. Lawrence, "Fiddler: a web debugging proxy tool," Telerik (Progress) software, 2003, available at: https://www.telerik.com/fiddler. [Cited on page 21.]

[81] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 23–26. [Cited on pages 21, 44, and 78.]

[82] P. Norvig, "Natural language corpus data," *Beautiful data*, pp. 219–242, 2009. [Cited on page 23.]

[83] H. Wang, J. Hong, and Y. Guo, "Using text mining to infer the purpose of permission use in mobile apps," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ser. UbiComp '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1107–1118. [Online]. Available: https://doi.org/10.1145/2750858.2805833 [Cited on page 28.]

[84] K. Spärck Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, vol. 60, no. 5, pp. 493–502, 2004. [Cited on page 28.]

[85] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding android obfuscation techniques: A large-scale investigation in the wild," in *Security and privacy in communication networks: 14th international conference, secureComm 2018, Singapore, Singapore, August 8-10, 2018, proceedings, part i*. Springer, 2018, pp. 172–192. [Cited on pages 30 and 33.]

[86] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl, "A large scale investigation of obfuscation use in google play," in *Proceedings of the 34th annual computer security applications conference*, 2018, pp. 222–235. [Cited on page 33.]

[87] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting android reuse studies in the context of code obfuscation and library usages," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 242–251. [Online]. Available: https://doi.org/10.1145/2597073.2597109

[88] U. Kargén, N. Mauthe, and N. Shahmehri, "Characterizing the use of code obfuscation in malicious and benign android apps," in *Proceedings of the 18th International Conference on Availability, Reliability and Security*, 2023, pp. 1–12. [Cited on page 30.]

[89] E. Lafortune, *ProGuard*, 2002, accessed: 2025-01-28. [Online]. Available: https://www.guardsquare.com/proguard [Cited on pages 30 and 77.]

[90] Google, *R8 Optimizer and Obfuscator*, 2018, accessed: 2025-01-28. [Online]. Available: https://developer.android.com/studio/build/shrink-code [Cited on page 30.]

[91] ——, *Android Studio*, 2013, accessed: 2025-01-28. [Online]. Available: https://developer.android.com/studio [Cited on page 30.]

[92] A. Niroshan, S. Seneviratne, and A. Seneviratne, "An empirical study of code obfuscation practices in the google play store," *arXiv preprint arXiv:2502.04636*, 2025. [Cited on page 33.]

[93] Y. Isobe and H. Tamada, "Are identifier renaming methods secure?" in *2018 19th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2018, pp. 322–328. [Cited on page 33.]

[94] "Target API level requirements for Google Play apps - Play Console Help — support.google.com," https://support.google.com/googleplay/android-developer/answer/11926878, [Accessed 10-03-2025]. [Cited on page 34.]

[95] H. Inayoshi, S. Kakei, and S. Saito, "Execution recording and reconstruction for detecting information flows in android apps," *IEEE Access*, vol. 11, pp. 10 730–10 750, 2023. [Cited on pages 34 and 63.]

[96] "React native: Learn once, write anywhere," https://reactnative.dev/, Meta Platforms, Inc., 2025, accessed: 2025-09-21. [Cited on page 37.]

[97] G. Jenks, "wordsegment: English word segmentation (python package)," https://pypi.org/project/wordsegment/, 2018, version 1.3.1, Apache 2.0 License. [Cited on page 52.]

[98] Exodus Privacy, "Exodus," https://reports.exodus-privacy.eu.org/en/, accessed: 2025-06-01. [Cited on pages 57 and 79.]

[99] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, vol. 15, no. 35, 2011. [Cited on page 61.]

[100] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 627–638. [Cited on page 61.]

[101] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the android application framework:{Re-Visiting} android permission specification analysis," in *25th USENIX security symposium (USENIX security 16)*, 2016, pp. 1101–1118. [Cited on page 61.]

[102] Guardsquare, "proguard-core: Library to read, write, analyze, and process java bytecode," 2025, accessed: 2025-09-24. [Cited on page 63.]

[103] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002. [Cited on page 69.]

A. Purpose Classification Taxonomy

Data types (what)	Purposes (why)	Example usages
Instance / hardware / advertising ID	Tracking for advertising	Support ad targeting / evaluation
	Tracking for data analytics	Avoid redundant device counting in marketing
	Signed-out personalization	Personalize news for sign-out users
	Anti-fraud	Enforce free content / advertisement limits
	Authentication	Relogin a user with a cookie

Table A.1: Purpose taxonomy for Phone ID - Extracted from [55]

Data types (what)	Purposes (why)	Example usages
Battery/charging	Battery-based event trigger	Show charging/low battery notifications
	Power management	Adapt the phone settings to save battery
	Data collection for analytics	Analyze the battery assumption of apps
Device Info	Interface customization	Customize the interface based on the resolution
	Data collection for ad	Collect data for ad personalization
	Data collection for analytics	Collect data for marketing analysis
Network	Network switch notification	Show wifi/lte switch notification
	Network optimization	Download low resolution images when on LTE
	Geo localization	Use IP to infer the geo location
	Data collection for ad	Collect data for ad personalization
	Data collection for analytics	Collect data for marketing analysis
Running state	Cross-app communication	Support cross-app communication
	Task management	Detect foreground tasks
Notification	Interface customization	Customize lock screen notifications
	Interruption management	Delay notification to manage interruption

Table A.2: Purpose Taxonomy for Phone Status - Extracted from [55]

Data types (what)	Purposes (why)	Example usages
Contact list	Backup and Synchronization	Backup contacts to the server
	Contact management	Merge duplicate contacts
	Blacklist	Block unwanted calls
	Call and SMS	Make VoIP/Wifi calls using Internet
	Contact-based customization	Add contacts to the personalized dictionary of input typing apps
	Email	Send Email to contacts
	Find friends	Find common friends who use the same service
	Record	Display call history
	Fake calls and SMS	Select a contact to fake calls
Calendar	Context prediction	Predict if the user is commute to workplace
	Schedule	Manage schedule conflicts
	Alarm	Notify calendar events ahead
SMS	Send messages	Send messages through apps
	Organize messages	Cluster/delete/re-rank SMS messages
	Extract message content	Extract the verification code in SMS
	Block messages	Block unwanted SMS
	Schedule messages	Delay the messaging sending
	Backup/sync messages	Backup messages to the server
Storage	Access photo album	Modify or upload photos
	Manage downloaded files	Save photos to local storage
	App data persistent storage	Save configuration files
Account/user profile	Third-party login	Login through Facebook/Google accounts
	Data collection for analytics	Collect data for marketing analysis
	Data collection for ad	Collect data for ad personalization

Table A.3: Purpose Taxonomy for Personal Data - Extracted from [55]

Data types (what)	Purposes (why)	Example usages
Camera	Flashlight	Turn on/off flashlight
	Video streaming	Stream the video capture
	Code scanning	Scan QR code
	Document scanning	Scan document
	Augment reality	Capture videos
	Text recognition	Recognize text in live capture
	Photo taking	Take photos
Location	Nearby Search	Search nearby POIs/real estates
	Location-based Customization	Fetch weather information
	Query Transportation Information	Estimate the trip time through Uber API
	Recording	Track the running velocity
	Map and Navigation	Find the user location in Map apps
	Geosocial Networking	Find nearby users in the social network
	Geotagging	Tag photos with locations
	Location Spoofing	Set up fake GPS locations
	Alert and Remind	Remind location-based tasks
	Location-based game	Play games require users' physical location
	Reverse geocoding	Use the GPS coords to find the real-world address.
	Data collection for analytics	Collect data for marketing analysis
	Data collection for ad	Collect data for ad personalization
Microphone	Voice Authentication	Authenticate users using voices
	Audio streaming	Make VOIP phone calls
	Voice control	Use voice to send the command
	Speech recognition	Turn the speech audio into text
	Audio recording	Record voice messages
	Acoustic event detection	Sense users' health using microphone
	Acoustic communication	Decode audios to receive messages
	Music	Record songs
Accelerometer/ gyroscope/ magnetometer	Step-counter	Count users' steps
	Game input controller	Detect device movements to control game input
	Map/Navigation/Compass	Use dead reckoning to improve localization
Proximity sensor	Speaker/display activation	Turn off screen if the phone is near the users' ear

Table A.4: Purpose taxonomy for Sensor - Extracted from [55]

```
    "request": {
                "method": "GET",
                "url": "https://146.75.90.132/backend-amgp-lmfr/v3/stores/api/v2/stores?
    lat=12.345&lon=54.321&page=1&countryISO=PT",
                "real_url": "https://api.lm.app/backend-amgp-lmfr/v3/stores/api/v2/stores?
    lat=12.345&lon=54.321&page=1&countryISO=PT",
                "text": "",
                "port": 443,
                "source_port": -1,
                "cookies": {
                    "datadome": "XXXXXX",
                    "Max-Age": "31536000",
                    "Domain": ".lm.app",
                    "Path": "/",
                    "Secure": "",
                    "SameSite": "Lax"
                },
                "path": "/backend-amgp-lmfr/v3/stores/api/v2/stores?lat=12.345&lon=54.321&
    page=1&countryISO=PT",
                "authority": "api.lm.app",
                "host": "146.75.90.132",
                "urlencoded_form": {},
                "first_line_format": "absolute",
                "host_header": "api.lm.app",
                "http_version": "HTTP/2.0",
                "query": {
                    "lat": "12.345",
                    "lon": "54.321",
                    "page": "1",
                    "countryISO": "PT"
                },
                "multipart_form": {},
                "path_components": [
                    "backend-amgp-lmfr",
                    "v3",
                    "stores",
                    "api",
                    "v2",
                    "stores"
                ],
                "timestamp_end": 1747217201.3071487,
                "timestamp_start": 1747217201.3071487,
                "scheme": "https",
                "trailers": {},
                "headers": {
                    "from-feature": "DEFAULT",
                    "user-agent": "Leroy Merlin/8.41.3 (prod-release; build: 8410300;
    Android: 12-API31) OkHttp/4.12.0",
                    "x-bu-code": "LMPT",
                    "accept-language": "en-US",
                    "x-customer-type": "",
                    "x-devicemodel": "XXXX",
                    "operating-system": "android",
                    "application-version": "8.41.3",
                    "application-id": "com.adeo.android.app",
                    "accept": "application/json",
                    "x-clientapikey": "XXXX",
                    "cookie": "datadome=XXXX; Max-Age=31536000; Domain=.lm.app; Path=/;
    Secure; SameSite=Lax",
                    "accept-encoding": "gzip"
                }
            },
```


Listing A.1: Example of Network Request - Desensitized

Application Package Names
air.co.id.netmediatama.NetMediatama
co.triller.droid
com.LTGExamPracticePlatform.Prep4SAT
com.accuweather.android
com.adeo.android.app
com.aliendev.khmersmartkeyboard
com.amazon.mShop.android.shopping
com.android.chrome
com.app.pepperfry
com.carbuddy.app
com.emojifamily.emoji.keyboard
com.g8n8.pregnancytracker
com.gomtv.gomaudio
com.google.android.apps.maps
com.google.android.googlequicksearchbox
com.google.android.youtube
com.google.earth
com.handmark.expressweather
com.khmercreator.khmermusicbox
com.lafourchette.lafourchette
com.mapquest.android.ace
com.microsoft.translator
com.mventus.selfcare.activity
com.perm.kate.new.6
com.podcast.podcasts
com.polyglotmobile.vkontakte
com.radio.fmradio
com.radiolight.bresil
com.radiolight.etatsunis
com.radiolight.france
com.radiolight.russie
com.ria.auto
com.safakge.radyokulesi
com.spotify.music
com.ssstudio.anatomy
com.ubercab.eats
com.waze
com.weather.Weather
com.yelp.android
com.zavariseapps.bibliadamulhermp3
masih.vahida.and.saz.arabi.demo
me.sorteie
mobi.lockdown.weather

Table A.5: List of Application Package Names

B. Evaluation Tables

Feature Code	Feature Name
MobiPurpose	MobiPurpose
Callstack	C
callstack Distance	D
Libradar	L
Query Count	Q
Trackers	T
Flowdroid Count	F_C
Flowdroid Match	F_M

Table B.1: Mapping of feature codes to feature names

Feature Set	Ad Tracking (218)			Analytic Tracking (173)			Anti Fraud (21)			Authentication (152)			M-avg	W-avg
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	F1	F1
MobiPurpose	0.872	0.810	0.840	0.839	0.788	0.812	0.500	0.500	0.500	0.789	0.909	0.845	0.749	0.827
C	0.850	0.810	0.829	0.844	0.818	0.831	1.000	0.500	0.667	0.811	0.909	0.857	0.796	0.835
D	0.857	0.857	0.857	0.931	0.818	0.871	1.000	0.500	0.667	0.763	0.879	0.817	0.803	0.846
DL	0.895	0.810	0.850	0.871	0.818	0.844	1.000	0.500	0.667	0.750	0.909	0.822	0.796	0.836
DT	0.971	0.786	0.868	0.963	0.788	0.867	0.400	1.000	0.571	0.705	0.939	0.805	0.778	0.844
DTL	0.902	0.881	0.892	0.964	0.818	0.885	1.000	0.500	0.667	0.775	0.939	0.849	0.823	0.873
CD	0.868	0.786	0.825	0.875	0.848	0.862	1.000	0.500	0.667	0.769	0.909	0.833	0.797	0.836
CDL	0.943	0.786	0.857	0.871	0.818	0.844	0.400	1.000	0.571	0.769	0.909	0.833	0.776	0.841
CDT	0.872	0.810	0.840	0.848	0.848	0.848	1.000	0.500	0.667	0.784	0.879	0.829	0.796	0.836
CDTL	0.854	0.833	0.843	0.900	0.818	0.857	1.000	0.500	0.667	0.763	0.879	0.817	0.796	0.836
CL	0.854	0.833	0.843	0.848	0.848	0.848	0.500	0.500	0.500	0.853	0.879	0.866	0.764	0.845
CQ	0.854	0.833	0.843	0.818	0.818	0.818	1.000	0.500	0.667	0.800	0.848	0.824	0.788	0.827
CQD	0.921	0.833	0.875	0.900	0.818	0.857	0.400	1.000	0.571	0.757	0.848	0.800	0.776	0.842
CQDL	0.850	0.810	0.829	0.848	0.848	0.848	1.000	0.500	0.667	0.833	0.909	0.870	0.803	0.844
CQDT	0.921	0.833	0.875	0.844	0.818	0.831	0.400	1.000	0.571	0.829	0.879	0.853	0.783	0.850
CQDTL	0.943	0.786	0.857	0.812	0.788	0.800	0.400	1.000	0.571	0.763	0.879	0.817	0.761	0.823
CQL	0.872	0.810	0.840	0.871	0.818	0.844	1.000	0.500	0.667	0.769	0.909	0.833	0.796	0.836
CQT	0.854	0.833	0.843	0.871	0.818	0.844	1.000	0.500	0.667	0.784	0.879	0.829	0.796	0.836
CQTL	0.917	0.786	0.846	0.818	0.818	0.818	0.400	1.000	0.571	0.778	0.848	0.812	0.762	0.822
CT	0.919	0.810	0.861	0.818	0.818	0.818	0.400	1.000	0.571	0.800	0.848	0.824	0.768	0.832
CTL	0.860	0.881	0.871	0.897	0.788	0.839	1.000	0.500	0.667	0.784	0.879	0.829	0.801	0.845
L	0.857	0.857	0.857	0.931	0.818	0.871	1.000	0.500	0.667	0.763	0.879	0.817	0.803	0.846
Q	0.897	0.833	0.864	0.897	0.788	0.839	0.400	1.000	0.571	0.784	0.879	0.829	0.776	0.841
QD	0.895	0.810	0.850	0.867	0.788	0.825	0.400	1.000	0.571	0.784	0.879	0.829	0.769	0.831
QDL	0.857	0.857	0.857	0.929	0.788	0.852	1.000	0.500	0.667	0.744	0.879	0.806	0.795	0.837
QDT	0.897	0.833	0.864	0.964	0.818	0.885	1.000	0.500	0.667	0.738	0.939	0.827	0.811	0.856
QDTL	0.971	0.810	0.883	0.964	0.818	0.885	0.400	1.000	0.571	0.738	0.939	0.827	0.792	0.861
QL	0.833	0.833	0.833	0.929	0.788	0.852	1.000	0.500	0.667	0.744	0.879	0.806	0.790	0.828
QT	0.971	0.810	0.883	0.964	0.818	0.885	0.400	1.000	0.571	0.738	0.939	0.827	0.792	0.861
QTL	0.878	0.857	0.867	0.963	0.788	0.867	1.000	0.500	0.667	0.756	0.939	0.838	0.810	0.855
T	0.947	0.857	0.900	0.963	0.788	0.867	0.400	1.000	0.571	0.775	0.939	0.849	0.797	0.869
TL	0.900	0.857	0.878	0.963	0.788	0.867	1.000	0.500	0.667	0.738	0.939	0.827	0.810	0.855

Table B.2: Performance for `ID.GENERALID`

Feature Set	Ad Data (118)			Analytic Data (247)			Interface Customization (224)			M-avg	W-avg
	P	R	F1	P	R	F1	P	R	F1	F1	F1
MobiPurpose	0.616	0.833	0.709	0.930	0.856	0.892	0.860	0.784	0.821	0.807	0.831
C	0.731	0.907	0.810	0.926	0.896	0.911	0.882	0.804	0.841	0.854	0.866
D	0.718	0.944	0.816	0.939	0.864	0.900	0.863	0.804	0.832	0.849	0.859
DL	0.729	0.944	0.823	0.940	0.880	0.909	0.872	0.804	0.837	0.856	0.866
DT	0.735	0.926	0.820	0.957	0.880	0.917	0.857	0.824	0.840	0.859	0.870
DTL	0.731	0.907	0.810	0.941	0.896	0.918	0.874	0.814	0.843	0.857	0.870
CD	0.731	0.907	0.810	0.933	0.888	0.910	0.863	0.804	0.832	0.851	0.863
CDL	0.754	0.907	0.824	0.935	0.928	0.932	0.891	0.804	0.845	0.867	0.880
CDT	0.758	0.926	0.833	0.941	0.896	0.918	0.865	0.814	0.838	0.863	0.873
CDTL	0.781	0.926	0.847	0.927	0.920	0.924	0.871	0.794	0.831	0.867	0.875
CL	0.742	0.907	0.817	0.926	0.904	0.915	0.860	0.784	0.821	0.851	0.862
CQ	0.758	0.926	0.833	0.917	0.888	0.902	0.840	0.775	0.806	0.847	0.854
CQD	0.731	0.907	0.810	0.934	0.912	0.923	0.891	0.804	0.845	0.859	0.873
CQDL	0.754	0.907	0.824	0.928	0.928	0.928	0.901	0.804	0.850	0.867	0.880
CQDT	0.754	0.907	0.824	0.942	0.904	0.922	0.865	0.814	0.838	0.861	0.873
CQDTL	0.769	0.926	0.840	0.927	0.920	0.924	0.880	0.794	0.835	0.866	0.875
CQL	0.742	0.907	0.817	0.925	0.888	0.906	0.863	0.804	0.832	0.852	0.862
CQT	0.769	0.926	0.840	0.935	0.920	0.927	0.871	0.794	0.831	0.866	0.876
CQTL	0.754	0.907	0.824	0.942	0.904	0.922	0.865	0.814	0.838	0.861	0.873
CT	0.758	0.926	0.833	0.942	0.904	0.922	0.874	0.814	0.843	0.866	0.876
CTL	0.758	0.926	0.833	0.933	0.896	0.914	0.863	0.804	0.832	0.860	0.869
L	0.721	0.907	0.803	0.925	0.888	0.906	0.871	0.794	0.831	0.847	0.859
Q	0.718	0.944	0.816	0.940	0.872	0.905	0.872	0.804	0.837	0.852	0.863
QD	0.725	0.926	0.813	0.933	0.888	0.910	0.871	0.794	0.831	0.851	0.863
QDL	0.718	0.944	0.816	0.940	0.880	0.909	0.882	0.804	0.841	0.855	0.866
QDT	0.727	0.889	0.800	0.932	0.880	0.905	0.856	0.814	0.834	0.847	0.859
QDTL	0.766	0.907	0.831	0.933	0.896	0.914	0.856	0.814	0.834	0.860	0.869
QL	0.725	0.926	0.813	0.933	0.888	0.910	0.871	0.794	0.831	0.851	0.863
QT	0.746	0.926	0.826	0.933	0.888	0.910	0.874	0.814	0.843	0.860	0.869
QTL	0.769	0.926	0.840	0.941	0.896	0.918	0.856	0.814	0.834	0.864	0.873
T	0.725	0.926	0.813	0.949	0.888	0.917	0.874	0.814	0.843	0.858	0.870
TL	0.725	0.926	0.813	0.940	0.880	0.909	0.874	0.814	0.843	0.855	0.867

Table B.3: Performance for `PHONE.DEVICE`

Feature Set	Ad Data (31)			Analytic Data (15)			Location-based Customization (16)			Map Navigation (15)			Nearby Search (18)			M-avg	W-avg
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	F1	F1
MobiPurpose	0.857	1.000	0.923	0.429	1.000	0.600	0.833	0.625	0.714	0.800	1.000	0.889	1.000	0.200	0.333	0.792	0.701
C	0.545	1.000	0.706	0.800	0.667	0.727	1.000	0.625	0.769	1.000	0.500	0.667	0.250	0.200	0.222	0.618	0.603
D	0.857	1.000	0.923	0.429	1.000	0.600	0.875	0.875	0.875	1.000	1.000	1.000	1.000	0.200	0.333	0.746	0.750
DL	0.857	1.000	0.923	0.667	1.000	0.800	0.750	0.375	0.500	0.471	1.000	0.640	1.000	0.200	0.333	0.639	0.644
DT	0.857	1.000	0.923	0.800	0.667	0.727	0.833	0.625	0.714	0.364	0.500	0.421	0.200	0.200	0.200	0.597	0.603
DTL	0.857	1.000	0.923	0.857	1.000	0.923	0.833	0.625	0.714	0.364	0.500	0.421	0.250	0.200	0.222	0.641	0.634
CD	0.545	1.000	0.706	0.857	1.000	0.923	1.000	0.625	0.769	1.000	0.500	0.667	0.333	0.200	0.250	0.663	0.636
CDL	0.857	1.000	0.923	0.857	1.000	0.923	0.833	0.625	0.714	1.000	0.500	0.667	0.692	0.900	0.783	0.802	0.807
CDT	0.857	1.000	0.923	0.857	1.000	0.923	0.875	0.875	0.875	1.000	0.500	0.667	0.692	0.900	0.783	0.834	0.836
CDTL	0.857	1.000	0.923	0.857	1.000	0.923	0.833	0.625	0.714	1.000	0.500	0.667	0.600	0.900	0.720	0.789	0.792
CL	0.667	1.000	0.800	0.857	1.000	0.923	0.833	0.625	0.714	1.000	0.500	0.667	1.000	0.900	0.947	0.810	0.810
CQ	0.632	1.000	0.774	0.800	0.667	0.727	1.000	0.625	0.769	1.000	0.500	0.667	0.818	0.900	0.857	0.759	0.766
CQD	0.600	1.000	0.750	0.800	0.667	0.727	1.000	0.625	0.769	1.000	0.500	0.667	0.200	0.200	0.200	0.623	0.610
CQDL	0.800	1.000	0.889	0.857	1.000	0.923	1.000	0.625	0.769	1.000	0.500	0.667	0.692	0.900	0.783	0.806	0.807
CQDT	0.923	1.000	0.960	0.857	1.000	0.923	1.000	0.875	0.933	1.000	0.500	0.667	0.692	0.900	0.783	0.853	0.856
CQDTL	0.923	1.000	0.960	0.857	1.000	0.923	1.000	0.625	0.769	1.000	0.500	0.667	0.600	0.900	0.720	0.808	0.812
CQL	0.462	1.000	0.632	0.857	1.000	0.923	1.000	0.625	0.769	1.000	0.500	0.667	1.000	0.200	0.333	0.665	0.635
CQT	0.800	1.000	0.889	0.857	1.000	0.923	1.000	0.625	0.769	1.000	0.500	0.667	0.692	0.900	0.783	0.806	0.807
CQTL	0.667	1.000	0.800	0.857	1.000	0.923	0.833	0.625	0.714	1.000	0.500	0.667	1.000	0.900	0.947	0.810	0.810
CT	0.857	1.000	0.923	0.857	1.000	0.923	0.833	0.625	0.714	1.000	0.500	0.667	0.600	0.900	0.720	0.789	0.792
CTL	0.923	1.000	0.960	0.857	1.000	0.923	1.000	0.625	0.769	1.000	0.500	0.667	0.600	0.900	0.720	0.808	0.812
L	0.857	1.000	0.923	0.857	1.000	0.923	0.833	0.625	0.714	1.000	0.500	0.667	0.692	0.900	0.783	0.802	0.807
Q	0.857	1.000	0.923	0.800	0.667	0.727	0.833	0.625	0.714	0.471	1.000	0.640	0.500	0.200	0.286	0.658	0.662
QD	0.857	1.000	0.923	0.857	1.000	0.923	0.875	0.875	0.875	0.533	1.000	0.696	1.000	0.200	0.333	0.750	0.739
QDL	0.857	1.000	0.923	0.857	1.000	0.923	0.875	0.875	0.875	0.533	1.000	0.696	1.000	0.200	0.333	0.750	0.739
QDT	0.857	1.000	0.923	0.857	1.000	0.923	0.833	0.625	0.714	0.364	0.500	0.421	0.250	0.200	0.222	0.641	0.634
QDTL	0.857	1.000	0.923	0.857	1.000	0.923	0.833	0.625	0.714	0.364	0.500	0.421	0.250	0.200	0.222	0.641	0.634
QL	0.857	1.000	0.923	0.333	1.000	0.500	0.875	0.875	0.875	1.000	0.500	0.667	1.000	0.200	0.333	0.660	0.676
QT	0.857	1.000	0.923	0.857	1.000	0.923	0.833	0.625	0.714	0.364	0.500	0.421	0.250	0.200	0.222	0.641	0.634
QTL	0.857	1.000	0.923	0.857	1.000	0.923	0.833	0.625	0.714	0.364	0.500	0.421	0.250	0.200	0.222	0.641	0.634
T	0.857	1.000	0.923	0.857	1.000	0.923	0.833	0.625	0.714	0.364	0.500	0.421	0.250	0.200	0.222	0.641	0.634
TL	0.857	1.000	0.923	0.857	1.000	0.923	0.833	0.625	0.714	0.364	0.500	0.421	0.250	0.200	0.222	0.641	0.634

Table B.4: Performance for `SENSOR.LOCATION`

Feature Set	Ad Data and Network Optimization (104)			Analytic Data (71)			M-avg	W-avg
	P	R	F1	P	R	F1	F1	F1
MobiPurpose	0.882	0.698	0.779	0.581	0.818	0.679	0.729	0.745
C	0.879	0.674	0.763	0.562	0.818	0.667	0.715	0.730
D	0.882	0.698	0.779	0.581	0.818	0.679	0.729	0.745
DL	0.882	0.698	0.779	0.581	0.818	0.679	0.729	0.745
DT	0.906	0.674	0.773	0.576	0.864	0.691	0.732	0.745
DTL	0.909	0.698	0.789	0.594	0.864	0.704	0.747	0.760
CD	0.879	0.674	0.763	0.562	0.818	0.667	0.715	0.730
CDL	0.879	0.674	0.763	0.562	0.818	0.667	0.715	0.730
CDT	0.909	0.698	0.789	0.594	0.864	0.704	0.747	0.760
CDTL	0.906	0.674	0.773	0.576	0.864	0.691	0.732	0.745
CL	0.865	0.744	0.800	0.607	0.773	0.680	0.740	0.759
CQ	0.857	0.698	0.769	0.567	0.773	0.654	0.712	0.730
CQD	0.882	0.698	0.779	0.581	0.818	0.679	0.729	0.745
CQDL	0.903	0.651	0.757	0.559	0.864	0.679	0.718	0.730
CQDT	0.882	0.698	0.779	0.581	0.818	0.679	0.729	0.745
CQDTL	0.882	0.698	0.779	0.581	0.818	0.679	0.729	0.745
CQL	0.857	0.698	0.769	0.567	0.773	0.654	0.712	0.730
CQT	0.882	0.698	0.779	0.581	0.818	0.679	0.729	0.745
CQTL	0.882	0.698	0.779	0.581	0.818	0.679	0.729	0.745
CT	0.906	0.674	0.773	0.576	0.864	0.691	0.732	0.745
CTL	0.906	0.674	0.773	0.576	0.864	0.691	0.732	0.745
L	0.889	0.744	0.810	0.621	0.818	0.706	0.758	0.775
Q	0.882	0.698	0.779	0.581	0.818	0.679	0.729	0.745
QD	0.882	0.698	0.779	0.581	0.818	0.679	0.729	0.745
QDL	0.865	0.744	0.800	0.607	0.773	0.680	0.740	0.759
QDT	0.909	0.698	0.789	0.594	0.864	0.704	0.747	0.760
QDTL	0.909	0.698	0.789	0.594	0.864	0.704	0.747	0.760
QL	0.886	0.721	0.795	0.600	0.818	0.692	0.744	0.760
QT	0.909	0.698	0.789	0.594	0.864	0.704	0.747	0.760
QTL	0.909	0.698	0.789	0.594	0.864	0.704	0.747	0.760
T	0.909	0.698	0.789	0.594	0.864	0.704	0.747	0.760
TL	0.909	0.698	0.789	0.594	0.864	0.704	0.747	0.760

Table B.5: Performance for `PHONE.NETWORK`

Feature Set	Ad Data (27)			Analytic Data (25)			Thirdparty Login (26)			M-avg	W-avg
	P	R	F1	P	R	F1	P	R	F1	F1	F1
MobiPurpose	0.400	0.857	0.545	0.900	0.562	0.692	0.900	0.750	0.818	0.685	0.706
C	0.438	1.000	0.609	0.900	0.562	0.692	1.000	0.750	0.857	0.719	0.732
D	0.462	0.857	0.600	0.917	0.688	0.786	0.900	0.750	0.818	0.735	0.760
DL	0.462	0.857	0.600	0.917	0.688	0.786	0.900	0.750	0.818	0.735	0.760
DT	0.545	0.857	0.667	0.929	0.812	0.867	0.900	0.750	0.818	0.784	0.810
DTL	0.500	1.000	0.667	0.917	0.688	0.786	1.000	0.750	0.857	0.770	0.786
CD	0.500	1.000	0.667	0.917	0.688	0.786	1.000	0.750	0.857	0.770	0.786
CDL	0.500	1.000	0.667	0.917	0.688	0.786	1.000	0.750	0.857	0.770	0.786
CDT	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
CDTL	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
CL	0.462	0.857	0.600	0.917	0.688	0.786	0.900	0.750	0.818	0.735	0.760
CQ	0.500	1.000	0.667	0.917	0.688	0.786	1.000	0.750	0.857	0.770	0.786
CQD	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
CQDL	0.500	1.000	0.667	0.917	0.688	0.786	1.000	0.750	0.857	0.770	0.786
CQDT	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
CQDTL	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
CQL	0.500	1.000	0.667	0.900	0.562	0.692	0.818	0.750	0.783	0.714	0.718
CQT	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
CQTL	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
CT	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
CTL	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
L	0.462	0.857	0.600	0.917	0.688	0.786	0.900	0.750	0.818	0.735	0.760
Q	0.400	0.857	0.545	0.900	0.562	0.692	0.900	0.750	0.818	0.685	0.706
QD	0.462	0.857	0.600	0.917	0.688	0.786	0.900	0.750	0.818	0.735	0.760
QDL	0.400	0.857	0.545	0.900	0.562	0.692	0.900	0.750	0.818	0.685	0.706
QDT	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
QDTL	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
QL	0.462	0.857	0.600	0.917	0.688	0.786	0.900	0.750	0.818	0.735	0.760
QT	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837
QTL	0.538	1.000	0.700	0.923	0.750	0.828	1.000	0.750	0.857	0.795	0.812
T	0.438	1.000	0.609	0.900	0.562	0.692	1.000	0.750	0.857	0.719	0.732
TL	0.583	1.000	0.737	0.929	0.812	0.867	1.000	0.750	0.857	0.820	0.837

Table B.6: Performance for `PERSONAL.ACCOUNT`

Feature Set	Ad Tracking (74)			Analytic Tracking (54)			Authentication (35)			M-avg	W-avg
	P	R	F1	P	R	F1	P	R	F1	F1	F1
QT	0.838	0.795	0.816	0.686	0.828	0.750	0.857	0.667	0.750	0.772	0.780
QTF_C	0.933	0.718	0.812	0.667	0.966	0.789	0.857	0.667	0.750	0.783	0.791
QTF_M	0.938	0.769	0.845	0.675	0.931	0.783	0.857	0.667	0.750	0.793	0.804
QTF_MF_C	0.933	0.718	0.812	0.667	0.966	0.789	0.857	0.667	0.750	0.783	0.791
DTL	0.882	0.769	0.822	0.703	0.897	0.788	0.800	0.667	0.727	0.779	0.791
DTLF_C	0.806	0.744	0.773	0.667	0.828	0.738	0.857	0.667	0.750	0.754	0.757
DTLF_M	0.886	0.795	0.838	0.722	0.897	0.800	0.800	0.667	0.727	0.788	0.802
DTLF_MF_C	0.838	0.795	0.816	0.714	0.862	0.781	0.857	0.667	0.750	0.782	0.790
T	0.886	0.795	0.838	0.711	0.931	0.806	0.923	0.667	0.774	0.806	0.814
TF_C	0.833	0.769	0.800	0.667	0.828	0.738	0.857	0.667	0.750	0.763	0.769
TF_M	0.838	0.795	0.816	0.714	0.862	0.781	0.857	0.667	0.750	0.782	0.790
TF_MF_C	0.882	0.769	0.822	0.730	0.931	0.818	0.800	0.667	0.727	0.789	0.801

Table B.7: Performance for `ID.GENERAL` with FlowDroid

Feature Set	Ad Data (70)			Analytic Data (121)			Interface Customization (37)			M-avg	W-avg
	P	R	F1	P	R	F1	P	R	F1	F1	F1
CQDL	0.815	0.667	0.733	0.939	0.969	0.954	0.500	0.600	0.545	0.744	0.822
CQDLF_C	0.815	0.667	0.733	0.938	0.953	0.946	0.480	0.600	0.533	0.737	0.815
CQDLF_M	0.771	0.818	0.794	0.937	0.922	0.929	0.579	0.550	0.564	0.762	0.829
CQDLF_MF_C	0.815	0.667	0.733	0.938	0.953	0.946	0.480	0.600	0.533	0.737	0.815
CDL	0.818	0.750	0.783	0.893	1.000	0.943	0.500	0.333	0.400	0.709	0.823
CDLF_C	0.818	0.750	0.783	0.893	1.000	0.943	0.500	0.333	0.400	0.709	0.823
CDLF_M	0.750	0.750	0.750	0.857	0.960	0.906	0.667	0.333	0.444	0.700	0.798
CDLF_MF_C	0.750	0.750	0.750	0.857	0.960	0.906	0.667	0.333	0.444	0.700	0.798
CT	0.818	0.818	0.818	0.939	0.969	0.954	0.611	0.550	0.579	0.784	0.851
CTF_C	0.794	0.818	0.806	0.925	0.969	0.947	0.625	0.500	0.556	0.769	0.840
CTF_M	0.786	0.667	0.721	0.939	0.969	0.954	0.522	0.600	0.558	0.744	0.821
CTF_MF_C	0.786	0.667	0.721	0.938	0.953	0.946	0.500	0.600	0.545	0.738	0.814

Table B.8: Performance for `PHONE.DEVICE` with FlowDroid

Feature Set	Ad Data and Network Optimization (51)			Analytic Data (12)			M-avg	W-avg
	P	R	F1	P	R	F1	F1	F1
L	0.889	0.889	0.889	0.600	0.600	0.600	0.744	0.826
LF_C	0.889	0.889	0.889	0.600	0.600	0.600	0.744	0.826
LF_M	0.889	0.889	0.889	0.600	0.600	0.600	0.744	0.826
LF_MF_C	0.889	0.889	0.889	0.600	0.600	0.600	0.744	0.826
TL	0.889	0.889	0.889	0.600	0.600	0.600	0.744	0.826
TLF_C	0.889	0.889	0.889	0.600	0.600	0.600	0.744	0.826
TLF_M	0.889	0.889	0.889	0.600	0.600	0.600	0.744	0.826
TLF_MF_C	0.889	0.889	0.889	0.600	0.600	0.600	0.744	0.826
T	0.889	0.889	0.889	0.600	0.600	0.600	0.744	0.826
TF_C	0.889	0.889	0.889	0.600	0.600	0.600	0.744	0.826
TF_M	0.895	0.944	0.919	0.750	0.600	0.667	0.793	0.864
TF_MF_C	0.900	1.000	0.947	1.000	0.600	0.750	0.849	0.904

Table B.9: Performance for `PHONE.NETWORK` with FlowDroid

Feature Set	Ad Data (12)			Analytic Data (8)			M-avg	W-avg
	P	R	F1	P	R	F1	F1	F1
QT	0.833	1.000	0.909	1.000	0.667	0.800	0.855	0.868
QTF_C	0.833	1.000	0.909	1.000	0.667	0.800	0.855	0.868
QTF_M	0.833	1.000	0.909	1.000	0.667	0.800	0.855	0.868
QT$F_M F_C$	0.833	1.000	0.909	1.000	0.667	0.800	0.855	0.868
CT	0.833	1.000	0.909	1.000	0.750	0.857	0.883	0.886
CTF_C	0.833	1.000	0.909	1.000	0.750	0.857	0.883	0.886
CTF_M	0.833	1.000	0.909	1.000	0.750	0.857	0.883	0.886
CT$F_M F_C$	0.833	1.000	0.909	1.000	0.750	0.857	0.883	0.886
TL	0.833	1.000	0.909	1.000	0.667	0.800	0.855	0.868
TLF_C	0.833	1.000	0.909	1.000	0.667	0.800	0.855	0.868
TLF_M	0.833	1.000	0.909	1.000	0.667	0.800	0.855	0.868
TL$F_M F_C$	0.833	1.000	0.909	1.000	0.667	0.800	0.855	0.868

Table B.10: Performance for `PERSONAL.ACCOUNT` with FlowDroid