World Academy of Science, Engineering and Technology
International Journal of Computer and Systems Engineering
Vol:17, No:3, 2023

Toward Understanding and Testing Deep Learning Information Flow in Deep Learning-Based Android Apps

Jie Zhang, Qianyu Guo, Tieyi Zhang, Zhiyong Feng, Xiaohong Li

Abstract—The widespread popularity of mobile devices and the development of artificial intelligence (AI) have led to the widespread adoption of deep learning (DL) in Android apps. Compared with traditional Android apps (traditional apps), deep learning based Android apps (DL-based apps) need to use more third-party application programming interfaces (APIs) to complete complex DL inference tasks. However, existing methods (e.g., FlowDroid) for detecting sensitive information leakage in Android apps cannot be directly used to detect DL-based apps as they are difficult to detect third-party APIs. To solve this problem, we design DLtrace, a new static information flow analysis tool that can effectively recognize third-party APIs. With our proposed trace and detection algorithms, DLtrace can also efficiently detect privacy leaks caused by sensitive APIs in DL-based apps. Additionally, we propose two formal definitions to deal with the common polymorphism and anonymous inner-class problems in the Android static analyzer. Using DLtrace, we summarize the non-sequential characteristics of DL inference tasks in DL-based apps and the specific functionalities provided by DL models for such apps. We conduct an empirical assessment with DLtrace on 208 popular DL-based apps in the wild and found that 26.0% of the apps suffered from sensitive information leakage. Furthermore, DLtrace outperformed FlowDroid in detecting and identifying third-party APIs. The experimental results demonstrate that DLtrace expands FlowDroid in understanding DL-based apps and detecting security issues therein.

Keywords—Mobile computing, deep learning apps, sensitive information, static analysis.

I. Introduction

DEEP Learning (DL) techniques have been widely applied into various applications, including speech recognition [1], images classification [2], and natural language processing [3]. With model compression technique [4] allowing DL models to complete offline inference tasks on mobile devices without degrading performance, a large number of deep learning based Android apps (DL-based apps) have emerged, which use DL techniques to complete a variety of complex tasks. These DL-based apps are usually applied in fields involving user-sensitive information, such as finance [5], traffic and healthcare [6]. With the widespread emergence of deep learning-based apps, user privacy and security concerns have become increasingly prominent. This also highlights the need to detect sensitive data leaks in DL-based apps. However, researches on DL-based apps only focus on the essential characteristics of DL-based apps [7], information flow analysis and automatic identification of model functionnalities in DL-based apps have not been investigated in depth.

The taint analysis is an information flow analysis technique that converts vulnerability detection problems into the problem of tracking the flow of user-sensitive data to malicious sink APIs. Therefore, taint analysis is effective to detect sensitive information leakage. The current detection techniques can be divided into dynamic and static detection technologies based on whether the program is running during detection. Compared to dynamic detection, static analysis ensures the scalability of profiling apps and traverses all possible execution paths. Automated static taint analysis tools for traditional apps (e.g., FlowDroid [8] which based on soot) is exhibit limitations, such as the inability to track faulty data leakages between third-party apis, performance, and efficiency. The newly proposed SEEKER [9] extended FlowDroid and defined sensor application programming interfaces (APIs) as sources to report privacy leaks, but it does not generalize sensitive data source APIs to general APIs used in DL-based apps, there is still a lack of DL oriented sensitive source and sink apis recognition technique in current research communities. To fill this research gap, in this study, we applied static taint analysis to the detection of DL-based apps.

The research community has demonstrated that numerous challenges are faced in the construction of a reliable and highly accurate static Android analyzer [10], DL-based apps are Android apps, therefore, considering the characteristics of Android [11], Java object-oriented characteristics must be taken into account when analyzing DL-based apps. In particular, the final state of the program can only be determined during the execution process for problems, such as dynamic code loading and polymorphism. Therefore, static analysis techniques often require auxiliary analysis techniques, such as alias analysis.

Given the above challenges and the status of DL-based apps, we design and implement an automated information flow analysis prototype tool DLtrace. DLtrace is applied to detect polluted access from source to sink APIs during DL inference tasks based on an understanding of the information flow characteristics in DL-based apps. We use the global control flow graph (CFG) of a DL-based app as an interface for the information flow analysis based on AndroGuard. We collect 208 DL-based apps to evaluate the detection performance of DLtrace and FlowDroid. The experiments demonstrate that DLtrace is superior to FlowDroid in the detection of sensitive

Jie Zhang, Tieyi Zhang, Xiaohong Li* and Zhiyong Feng are with the College of Intelligence and Computing, Tianjin University, China (*e-mail: xiaohongli@tju.edu.cn).

Qianyu Guo is with Zhongguancun Laboratory, Beijing, P.R. China (e-mail: guoqy@zgclab.edu.cn).

data leakage in DL-based apps.

The main contributions of this paper are as follows:

- This is a study to provide an understanding of the DL inference regarding the information flow in DL-based apps. We propose that sets of APIs can verify the DL model functionality in DL-based apps. For simplicity, we refer to these APIs as "magic APIs".
- We recollect sensitive source and sink APIs to cover the DL inference stage.
- Owing to the limited ability of FlowDroid to identify and detect third-party APIs, our DLtrace tool significantly outperforms FlowDroid in detecting sensitive data leakage caused by third-party APIs that support DL techniques.

II. Related Work

This section discusses the related work from two aspects: 1) the taint analysis for traditional apps, and 2) the research for DL-based apps.

A. Taint Analysis for Traditional App

Taint analysis techniques can be combined with both dynamic and static methods. The dynamic analysis method detects the behavior and data in the apps real-timely [12]. TaintDroid [13] is a dynamic analysis method that tracks and marks multiple sensitive sources. Nevertheless, TaintDroid lacked support for new versions of apps based on Android Runtime (ART). The emergence of TaintART [14] has solved this problem. TaintART redesigned the dynamic information flow tracking method supporting ART.

Compared with the dynamic detection method, the static analysis can cover all the code and paths of the program without execution and detect the error code segment or malicious features. However, static analysis is often ineffective in detecting apps using dynamic code loading or fuzzy technologies such as encryption [15]. FlowDroid [8] is a domain-sensitive, object-sensitive, context-sensitive, flow-sensitive data flow analysis tool, but it cannot analyze the ICC.

DroidSafe [16] supported ICC and the object-sensitive, and created a virtual function for each component, but did not support flow-sensitive. A similar tool to DroidSafe is IccTA [17]. Amandroid [18] is also a static analysis platform, which can detect privacy data leakage, data injection, and API misuse. Amandroid implemented the alias analysis, but its comprehensive alias analysis is very costly.

Most static analyzers are based on the famous analysis framework soot [19] extension. Since the APIs support DL techniques are primarily the third-party APIs, soot cannot represent them as intermediate representation (IR), limiting the ability of detection tools to analyze the information flow in the DL inference process in DL-based apps.

B. Research for DL-Based App

With the great success of AI, Xu et al. [7] first studied the application of DL technique in smartphones, many mobile DL frameworks have been proposed [20]–[22]. Since integrating DL frameworks into apps is more efficient than integrating DL frameworks into the cloud and is not limited by network speed, more and more apps integrate DL techniques. In this situation, the security problems (e.g., evasion, poisoning, and stealing) of DL models in DL-based apps inevitably arise [23]–[28]. There also emerge some researches focusing on the model security in DL-based apps. For example, Li et al. [29] proposed a model watermarking method, and the watermark is injected and hidden in the DL model. At the same time, the watermark verification avoidance technique [30] also arises. Another concurrence work [31] focused on the robustness of the DL models in DL-based apps. By identifying highly similar pre-trained DL models from TensorFlow Hub, they carried out adversarial attacks on DL models. Sun [32] studied the risk and cost of DL model leakage, and found that there is a case that the DL-based apps will trigger the DL model download when running. There is a risk that the plaintext of models will be stolen in memory, even if the DL models are encrypted. The above work focus on the security of the DL model itself. Different from them, this work pays attention to the security risks (e.g., sensitive data leakage) in the full execution stack when DL-based apps perform DL inferences.

III. DLtrace

In this section, we propose a static detection tool called DLtrace for DL-based apps. Fig. 1 shows the overall working process of DLtrace, which mainly consists of two stages. The first stage carries out data processing, and the second stage completes the information flow analysis through two algorithms designed in DLtrace. At last, the output of the DLtrace is an information flow analysis report and a sensitive information leakage detection report.

A. Data Collection

This subsection starts with the dataset collecting methods, including DL-based apps collection and sensitive source and sink identification.

DL-based apps Collection: To study the quality of DL-based apps in the current market, we crawled and collected a host of popular Android apps for further research and testing. To that end, we took a market snapshot in March 2021. The preliminary indicator of the collected candidate Android apps is the number of downloads. We downloaded the top 500 Android apps in each category, and the latest versions of the Android apps were downloaded from Google Play directly. Therefore, we gathered a total of 19306 candidate Android apps. To filter out DL-based apps from the above 19,306 Android apps, we used the method proposed in previous work [7]: detecting feature binary files in apps. Since the DL technique applied on the mobile terminal relies on the underlying libraries, the apps integrated with the famous AI frameworks will have the corresponding feature binary files. By using the static filtering approach, we determined 208 DL-based apps, clustered into 29 blocks by category tags in Google Play, and the same category apps in the Section IV-B used as similar research targets.

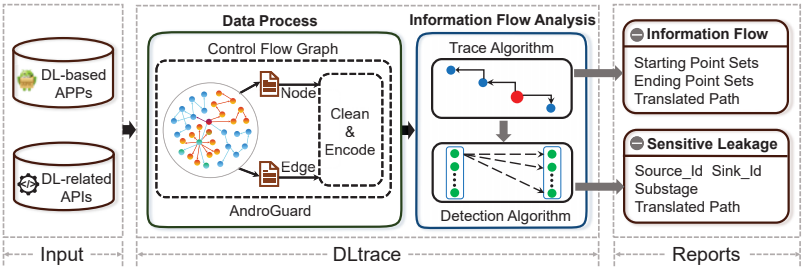


Fig. 1 Overview of DLtrace

Sensitive Source and Sink Identification: Sensitive sources and sinks need to be identified as DL-related API set to perform taint analysis. The sources refer to APIs related to DL inference tasks, such as APIs for initial loading data of user images into face detectors and APIs for constructing detectors, and sinks refer to APIs that perform dangerous operations, such as database storage, network sending data, and other APIs that abnormally transfer inference data and inference results. As there are no tools that can automate the identification of user-sensitive information APIs in the inference phase of DL tasks. This paper uses two ways to enrich the source and sink collections: Tracebacking the app information flow by DLtrace (the details of the trace Algorithm 1 are in the following section Section III-C); selecting from the Android developers' documentation. At last, we recollect 72 sources and 66 sinks, including first-party and third-party APIs. The selected set also includes the first-party APIs because of these APIs (e.g., APIs in the *android.Graphics.BitmapFactory* package process and pass image data before inference) are closely related to the DL inference processing.

B. Data Processing

The data processing phase focuses on extracting the CFG in the apps and pre-processing the CFG. To simplify, we conduct embedding for CFG to facilitate further information flow analysis, specifically cleaning and encoding to embedding. We still use the terminology of CFG in the following part. The CFG contains the raw data of nodes and calls. Data cleaning refers to processing the attribute information stored by each original node to ensure the readability of the output analysis report. Data encoding refers to all the nodes in the CFG, and encoded pairwise nodes also represent a call. The encoded node is remapped back before outputting the reports. In order to ensure the correctness of the experiment, CFG does not filter external node connections and multiple calls. Since the encoded nodes are unique, the parallel edges in the multigraph generated by the node calling relationship do not affect the experiment. Thus, the global CFG of an app is a directed multigraph. We define it in Definition 1.

Definition 1 (CFG): A CFG is a directed multigraph $CFG = (N, C)$, where N is the set of nodes and C is the set of calls.

Definition 2 (Node): In a CFG, a node is defined as a four-tuple of attributes $N = \langle p, n, m, r \rangle$, where p is the package name, n is the node name, m is the parameter list, and r is the return value (which can be empty) of the N. The DL-based app developers often obfuscate the components of the four-tuple, and the four attributes all have a probability of being obfuscated.

Algorithm 1: Tracing algorithm for information flow

Input : A=<T,P,S>: Extracting DL-based app Information
Output: R: Analysis Report

```
T := tracePoints(A);
P := nodeId(A);
S := edgeId(A);
for l ← 0 to Len(S) do
    W := addEdge(S[l]);
for i ← 0 to Len(T) do
    for p ← 0 to Len(P) do
        if shortestPath(P[p],T[i],W) then
            begin[m] ← P[p];
            m := m + 1;
                    ▷ (T[i],P[q],W) collecting ending points
            p := p + 1;
    for x ← 0 to m do
        simplePath(begin[x], T[i], W);
        translatePath(simplePath);
        x := x + 1;
    m ← 0;
    i := i + 1;
return R;
```

C. Information Flow Analysis

To perform information flow analysis, we design two algorithms with different focuses. The first algorithm can perform single-point up and down backtracking to generate information flow paths, which is named trace algorithm. Based on the trace algorithm and relying on the idea of taint analysis, the second algorithm determines sensitive information leakage during the data flow in the information flow, designated as detection algorithm. Trace algorithm is executed firstly to study the data processing process of the DL inference phase.

Algorithm 1 is a single-point tracing algorithm designed to describe the complete information flow of the DL inference stage in DL-based apps. Specifically, for a node N_P, this node may lie in one substage of the the DL inference tasks such as: data preprocessing, inference detector constructing or inference result processing. And in the case of having only one node (especially an input node or an output node) of a substage in the information flow, DLtrace takes N_P as the center to detect longer reachable paths $(N_{max\_b}, \ldots, N_P, \ldots, N_{max\_e})$. The algorithm uses A (the information set extracted from

Algorithm 2: Detection algorithm for information leakage path

Input : $D = \langle I, P, S \rangle$: Extracting DL-based app Information
Output: R: Analysis Report

1 $I := detectPoints(D)$;
2 $P := nodeId(D)$;
3 $S := edgeId(D)$;
4 **for** $i \leftarrow 0$ **to** $Len(I)$ **do**
5 **if** $getType(I[i]) == START$ **then**
6 $start[t] \leftarrow I[i]$;
7 $t := t + 1$;
8 **else**
9 $end[p] \leftarrow I[i]$;
10 $p := p + 1$;
11 **for** $j \leftarrow 0$ **to** $Len(S)$ **do**
12 $W := addEdge(S[j])$;
13 **for** $m \leftarrow 0$ **to** t **do**
14 **for** $n \leftarrow 0$ **to** p **do**
15 **if** $hasPath(start[m], end[n], W)$ **then**
16 $simplePath(start[m], end[n])$;
17 $translatePath(simplePath)$;
18 **return** R;

the app) as input, in the initial state, data processing should determine the T, P and S values in A (P refers to the encoding set of the N, S refers to the encoding set of the C, T is the encoding set of the nodes $\{N_1, \ldots, N_n\}$ that will be generated their information flow path), and the output is a report of the information flow derived with the nodes in T. In Algorithm 1, first the global CFG is recovered using the set S before the information flow analysis, adding the edge information to graph W through the traversal of S. Then the algorithm determines the starting and ending point sets of the shortest paths with N_i where $i \in (1 \ldots n)$, the idea of shortest path which is often used in graph algorithm is used to solve the starting point set and ending point set. To calculate the upstream derived paths with the starting set and the downstream derived paths with the ending set, the reachable paths derived from nodes adopts the idea of simple path solution in graph algorithm. Then it removes the duplicated subpaths and integrates the encoded paths, and uses the simplePath function to represent (Line 14). Finally, the encoded paths are translated and output to the information flow analysis report R. Line 8 of the algorithm is the starting point set judgment formula of N_i, and the formula to collect ending point set is $(T[i], P[q], W)$.

We design Algorithm 2 that directly determines a valid information flow path between pairwise points. Algorithm 2 will be used to perform sensitive information leakage detection. The input of the algorithm is the information extracted from the app, and the output is the result of determining the existence of paths between the test pairwise sets and all valid paths. In the input, we use D to denote the set of information extracted from the app, S is the encoding set of the C, and P is the encoding set of the N. I is the encoding set of the nodes to be predicted $\{N_1, \ldots, N_n\}$, and we need to assign a $type\_label(type\_lable \subseteq \{START, END\})$ to each prediction node. First, we cluster the set of test points by type labels through Line 5 of the algorithm. This step is to classify the input set of test points I. The following algorithm is to determine all connections from the START (as the source point) to END (as the sink point). Then the global CFG is recovered. The data connectivity between all pairwise points is determined by Line 15. In general, the algorithm detects the reachability between $start[m]$ and $end[n]$ in the W, if there exists at least one sequence of nodes from node $start[m]$ to $end[n]$, $start[m] = v_0, \ldots, v_n = end[n]$ where $v_k \in N\,(0 \leq k \leq n \wedge d\,(start[m], end[n]) > 0)$ and d represents the length of the directed path in CFG, then the simplePath between $start[m]$ to $end[n]$ are translated Line 17, the output of Algorithm 2 is the sensitive information leakage detection report R.

IV. Evaluation and Results

DLtrace is a static tracing tool for information flow based on taint analysis technique programmed with python. To demonstrate the effectiveness and efficiency of DLtrace, we design experiments to answer the following 3 research questions:

- **RQ1**: How useful and effective is DLtrace in helping understand the DL information flow within DL-based apps?
- **RQ2**: How useful is DLtrace in helping detect the privacy leakage issues in DL-based apps?
- **RQ3**: What kind of factor affect the performance of DLtrace?

A. Experimental Setups

All experiments were carried out on a PC with an Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz, 16.0 GB RAM, Microsoft Windows 11, PyCharm Community Edition 2021.2.1.Ink and python 3.9 Interpreter.

B. Model Functionality Identification with DLtrace (RQ1)

In DL-based apps, DL models are only a part of the functionalities of the apps. The core functionality of the apps is supported through the DL inference results. The reasoning results of DL models collaborate and exchange information with other transaction processing modules. In other words, there are functional differences between the tasks of the DL models and other transaction processing modules in the DL-based apps. At present, most of the DL models are embedded in the DL-based apps in the form of binary, and it is difficult to directly obtain the DL models in the apps. Although the existing research on the stealing DL models, we find that the structure of the models can be obtained intuitively after stealing, but the real function of the models still needs the cooperation of expert knowledge, and different parameters configuration and input also make the DL models play different functionalities. Therefore, without obtaining the models in DL-based apps, verifying what services the DL models provide for DL-based apps through the information

TABLE I
THE RESULTS OF MODEL FUNCTIONALITY IDENTIFY

Category	#Fun	Identification		Category	#Fun	Identification	
		Fields	DLtrace			Fields	DLtrace
SHOPPING	5	47.1%	94.1%	TOOLS	10	72.2%	66.7%
FINANCE	3	11.8%	94.1%	SOCIAL	6	20.8%	91.7%
PHOTOGRAPH	2	61.3%	74.2%	MAPS	10	46.2%	77.0%
COMMUNICATION	4	73.3%	86.7%	TRAVEL	9	42.9%	92.9%
MEDICAL	1	100%	100%	HEALTH	2	0.0%	100%

[1] Category refers to the category tags in Google Play; #Fun refers to the number of the detected DL model functionalities in this category based on DLtrace and magic APIs; Identification is the identification rate of DL model functionality.

TABLE II
MAGIC APIS IN THE CATEGORY OF FINANCE DL-BASED APPS

Model Functionality	Magic API	Subtask
Face Detection	confirmed $upload^*$ fromByteArray	Data Processing
	getFaceDetector	Detector Constructing
	detectInImage	Inference Results Processing
Barcode Detection	invoke initializeBarcodeDetector	Data Processing
	getVisionBarcodeDetector	Detector Constructing
	detectInImage	Inference Results Processing
Text Recognition	onPictureCaptured DocumentDetectionFrame getMRZDetector	Data Processing
	getOnDeviceTextRecognizer getPersistenceKey FirebaseVisionTextRecognizer get&put	Detector Constructing
	processImage checkArgument	Inference Results Processing

[1] $*$ represents the similar APIs, such as *uploadDocumentForValidation*, *uploadSelfie*, *uploadSelfieForValidation* and *uploadLivePhoto*.

flow is challenging. In order to identify the functionalities of the DL models, we set up the experiment, which process and results are as follows.

Firstly, exerting statistical experiments to collect magic APIs. The task of DL models to complete offline inference tasks is that in the process of interaction between DL models and DL-based apps, DL-based apps transmit the data (e.g. a picture data) to DL models for inference, and the inference results of the DL models are fed back to DL-based apps. When analyzing the information flow of DL models inference process in various DL-based apps, we find that DL-based apps with similar functionalities have similar information flow. Based on this finding, we summarize the APIs from the information flow, take the high frequency APIs as the magic APIs that can help identify the functionalities of the DL models, and establish the mapping relationship between the functionalities of the models and the magic APIs. In order to display magic APIs, according to the data processing in the information flow, we divide the information flow of the DL models inference task whole process into three parts: preparing data, constructing inference detector, and processing inference results. In order to make our statistical magic APIs locate in the inference stage of DL-based app, we start from meta APIs in each stage. We collect some meta APIs that support DL inference tasks to form a fundamental corpus. These APIs are defined in ML suites provided by official websites [33]–[35] with open-source documents of DL techniques. DLtrace uses Algorithm 1 to extract the information flow through these meta APIs in DL-based apps automatically, the APIs related to the DL model inference tasks are filtered from the information flow of the meta APIs, and added to the meta APIs. The meta APIs are continuously enriched in the iterative process. From the corpus, we perform the further statistical screening manually, the high-frequency APIs in the corpus are called magic APIs that can help identify DL model functionalities.

Meanwhile, we further conduct comparative experiments with the previous method proposed by Xu et al. [7] they manually determine the functionality of the DL model based on the official description of the released apps. According to their method, we count the description fields of apps about DL techniques in Google Play. The two experimental results of this part are shown as follows.

By analyzing 208 DL-based apps, we can determine 15 categories functionalitise of 15 DL models, and summarize that the total number of magic apis is 173. We find that the category of *TOOLS* DL-based apps contains such speech recognition, speech synthesis, and machine translation DL models; in the category of *SHOPPING* apps contains image recognition, and intelligent recommendation models; the image segmentation and other DL models are contained in *COMMUNICATION* apps; *MAP_AND_NAVIGATION* apps include object recognition and estimated time of arrival models; the models functionalities verified in other categories of DL-based apps are shown in Table I (with limited space, we only show 10 categories). Taking the *FINANCE* DL-based apps as an example, during the information flow analysis, we extract APIs that appear frequency more than 15% in the DL inference stage, and these APIs are not confused as partial magic APIs shown in Table II.

We compare the above two ways to identify the functionalities of DL models, and take *FINANCE* apps as an example. Table III shows the statistics: the *FINANCE* category includes 17 DL-based apps, 88.2% (15/17) of which have no fields about DL techniques in official descriptions. With the help of magic APIs, in DLtrace identified results, there are 8 apps using face detection models, 13 apps using the barcode detection, and 6 apps using the text recognition models. Among them, *lguplus.smartotp* app has been seriously confused in the information flow of the DL inference stage, therefore DLtrace cannot judge. Compared with the mean value that identifying the DL model functionality with official text representation, the ability of DLtrace to determine the *FINANCE* apps (at least one DL model is used) is 94.1% (16/17). The rate of the two approaches for identifying DL model functionalities in other categories of DL-based apps is shown in Table I.

Answer to RQ1: The collected magic APIs provide a basis for identifying the functionality of DL models in DL-based apps, and prove the usefulness of DLtrace in understanding the DL information flow during inference. Currently, determining the model functionality by the functional descriptions in the app market (i.e., Google Play) is still unsatisfactory, because

TABLE III
THE RESULTS OF DETERMINING MODEL FUNCTIONALITIES COMPARISON BETWEEN DLTRACE AND TEXT REPRESENTATION

DL-based Apps	DLtrace Analysis	Description Fields
atws.app *shinhan.foreignerbank.app* *cz.rb.app.smartphonebanking*⋆ *ch.sympany.clientporta*⋆ *paywaywallet* *wit.android.bcpBankingApp*⋆ *io.cex.app.prod*	barcode detection	No description about DL
fi.danskebank.mobilepay *dk.danskebank.mobilepay*⋆ *worldremit.android*	face detection	No description about DL
bitx.android.wallet *ripio.android* *medan.app.android* *mmoney.wallet*	face detection barcode detection text recognition	No description about DL
android.barclaysmobilebanking	barcode detection	Biological feature detection
lguplus.smartotp	serious confusion	Facial certification services
wooribank.smart.mwib⋆	barcode detection text recognition	No description about DL

[1] ⋆ indicates that in the DL-based app uses the second party APIs to complete the DL inference task, it means that these APPs do not use the third-party APIs that provided by Google and other companies to complete the DL inference task, but use the self-defined APIs.

the description of non-core functions is generally omitted in official app descriptions. By performing the information analysis in the apps, DLtrace is more effective in identifying the model functionalities.

From the results of RQ1, it can be observed that DL techniques in mobile devices are applied to safety- or security-critical fields, such as financial, and even medical care. These fields usually involve much privacy-sensitive information. Therefore, we must pay attention to detecting the leakage of sensitive information in DL-based apps, particularly during the DL inferencing.

C. Detection of Privacy Leakage with DLtrace (RQ2)

In this section, we utilize the idea of taint analysis to perform sensitive information leakage detection in 208 DL-based apps. Similar to FlowDroid [8], DLtrace is also designed as a general method. Configuring the DL-related API set that manipulates user-sensitive information to DLtrace and FlowDroid. The same DL-related API set will limit DLtrace and FlowDroid to the same test range.

DLtrace extracts all the source and sink APIs used in each DL-based app as the test point set I, calling the detection algorithm to validate. The details of the compared experimental results are below.

In Fig. 2, the leaked app refers to the DL-based app that is detected to have at least one pair leak. DLtrace can detect data leakage in 54 apps, such as *mmoney.wallet* app is detected with 11 pairwise sensitive data leakage. FlowDroid detects 39 apps are leaked apps, the largest number of detected sensitive data leakage pairs is 12 in an app. Simultaneously, in Fig. 5(b), DLtrace and FlowDroid both detect leaks in 9 apps, and 18 pairs of leaks be detected together.

The Leak Number is the total leakage pairs, DLtrace in 208 apps report a gross of 161 sensitive data leak routes, and FlowDroid detects 105 leakage paths. FlowDroid shows poor detection performance due to its limited ability to identify

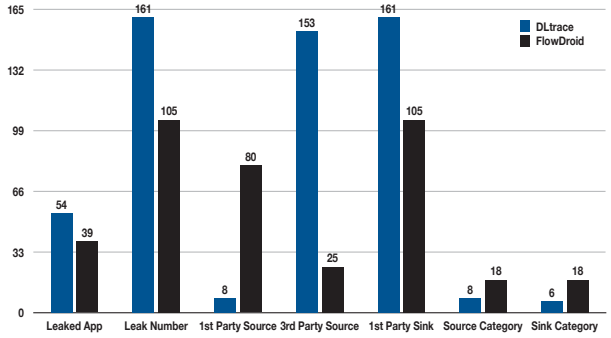


Fig. 2 The details of the detecting sensitive data leakage

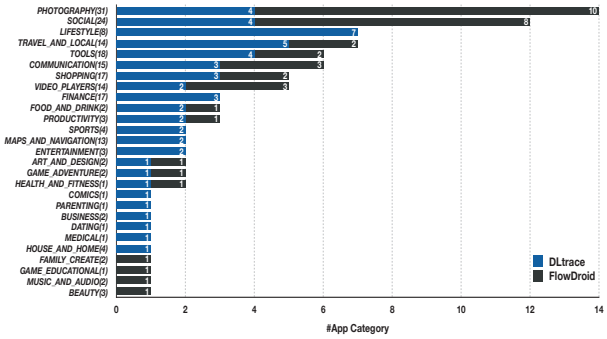


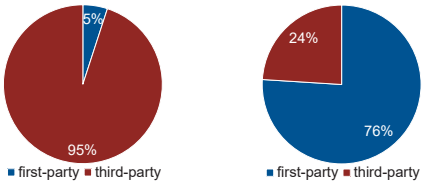
Fig. 3 Number of the categories in DL-based apps

the third-party APIs. This is because we use Androguard to extract the global CFG of DL-based apps, and retain all APIs in the apps for further analysis. FlowDroid is based on soot expansion, the ability of soot to transfer the intermediate representation (IR) of the third-party APIs is limited, which limits the scope of third-party apis that FlowDroid can analyze.

As demonstrated by Fig. 3, our DL-based apps dataset can be divided into 29 categories, DLtrace detects leaks in 23 categories, FlowDroid covers 16 categories of apps. For example, there are 17 apps in the category of *SHOPPING*, DLtrace detects leakage in three of them, FlowDroid detects leakage in two apps. DLtrace and FlowDroid do not report leak in the categories *GAME_SPORTS* and *LIBRARIES*, but this can not explain that these two categories of DL-based apps are the most secure, because in our dataset only collected one app respectively.

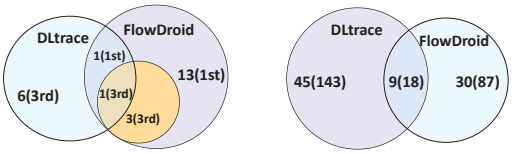
Details of the proportion of source APIs in the sensitive data leakage paths, the DLtrace test results shows that the number of first-party API sources (1st Party Source) among the tested leaked paths is 8, as we can see from Fig. 4(a) accounting for 5.0%, and the total number of third-party sources (3rd Party Source) APIs is 153, accounting for 95.0%. In contrast, 80 first-party sources are detected by FlowDroid, and 25 third-party sources, account for 76.0% (80/105) and 24.0% (25/105) respectively, as shown in Fig. 4(b).

The categories of source and sink in the test results, DLtrace detects eight categories of API sources and six categories of hazard sinks. FlowDroid reports that there are 18 categories of danger sources and sinks. In Fig. 5(a), the categories of sources identified by DLtrace alone have 6 in 74 leaked pairs.



(a) DLtrace (b) FlowDroid

Fig. 4 The category distribution of api-source detected by DLtrace and FlowDroid



(a) Source categories (b) Leaked app and number

Fig. 5 The details of source APIs, leaked app and number

FlowDroid can detect 13 categories of sources separately and belong to the first party. The leaks detected by both include two source APIs. One sink API identified by DLtrace alone exists in 3 leaked pairs, the API *getContentCharSet*. It is interesting that FlowDroid cannot detect the leak caused by this first-party API. Note that, there are 32 DL-based apps that FlowDroid cannot analyze, DLtrace detects three apps that suffered from data leakage.

The third-party sensitive sources detected separately by DLtrace such *createFileFromURI* and *launchCamera*. In these APIs, regarding the *RequestManger* class has the interface to set the image resource, the image data can be a local connection, a URL, or a Drawable resource id. In a transmission case shown in Fig. 6, DLtrace detects the leakage from *recognizeImage* to the *Log* API (i.e., *Log.v*). In the second substage of DL inferencing, the image provided feeds into the DL model, but the information is transmitted to log API illegally.

Answer to RQ2: Compared with FlowDroid, DLtrace is more effective in detecting sensitive leakage from third-party APIs with a wider API coverage. Meanwhile, for the first-party APIs, DLtrace can detect some sensitive leakage that FlowDroid does not report. Therefore, DLtrace can work as an extension for FlowDroid on detecting sensitive leakage in DL-based apps.

D. Bottleneck Factor for DLtrace (RQ3)

This section is in order to seek room for lifting the performance, java and Android characteristics constraint

```
SOURCE: 44036 SINK: 8610
Substage: 2
    <Lorg/reactnative/camera/tflite/Classifier;->recognizeImage(Landroid/graphics/Bitmap; I)>
    <Landroid/util/Log;->v(Ljava/lang/String; Ljava/lang/String;)I>
```

Fig. 6 A leakage example detected by the DLtrace

TABLE IV
THE OVERVIEW OF CHALLENGES FOR STATIC ANALYZERS

Java	DLtrace Android
dynamic code loading	Dalvil bytecode
reflection mechanism	entry point
native code integration	component lifecycle
multi-threading	usersystem events
polymorphism	inter-component communication(ICC)

the ability of static analyzer. The Table IV shows that the current static analyzers face main challenges. During the performance analysis of DLtrace, we find the java object-oriented inheritance mechanism is the major bottleneck factor to us, limiting the static analyzers to determine entire call relationship in the apps. Therefore, we summarized two enhancement patterns that can help enhance the information flow analysis. To explain our enhancement patterns in detail, we first declare the following definitions.

Definition 3 (Enhancement Pattern): Enhancement Pattern (EP) is a template for information flow auxiliary analysis. Given the two nodes N_a and N_b that have no edge connection in the CFG, they are not reachable, and two separate calling sequences $\{N_a = v_1, \ldots, v_{i-1}\}$ (starting from N_a) and $\{v_i, \ldots, v_n = N_b\}$ (ending to N_b), if polymorphism or anonymous inner-class exit between v_{i-1} and v_i. Then an enhancement pattern between N_a and N_b can been represented as:

$$N_a \xrightarrow{s} N_b, s \in (\alpha \cup \beta) \tag{1}$$

Note that, we currently focus on $d(v_{i-1}, v_i) = 1$ because $d > 1$ is more complex. Furthermore, we define the iterative enhancement pattern in Definition 4.

Definition 4 (Iterative EP): If the connection between N_a and N_b requires iteration after a series of enhancement pattern conversions, it can be represented as an iterative enhancement pattern $N_a \xrightarrow{s_1,\ldots,s_n} N_b, s_i \in (\alpha \cup \beta)\, i \in (1, \ldots, n)$, following the below definition:

$$\left(N_a, \ldots, N_{i-1} \xrightarrow{s_1} N_i, \ldots, N_{j-1} \xrightarrow{s_n} N_j, \ldots N_b\right) \tag{2}$$

The details of the two enhancement patterns are as below:

- Polymorphism α: Polymorphism allows different objects in the same class domain to respond to the same message. Only at running time can determine the object binding, and the static analyzer cannot locate the specific call.
- Anonymous internal class (AIC) β: As shown in Fig. 7, AIC relates to multi-threading and callback, there is no call relationship between B-function, C, and D-function.

Intuitively, polymorphism is incarnating the method overload and rewrite. AIC does not require defining a separate class. It also aims to support polymorphism.

Taking a critical DL-based app for example, as shown in Fig. 8 the analysis diagram of *TensorFlowdetect* app, there are two EPs: α and β. First, calls between APIs have non-sequential characteristics in the DL inference process, such as *getPixels* and *fetch* APIs, not direct call relationships. There is mode β between the *runInBackguard* and *run* nodes. The app combines multiple DL models the *recognizeImage*

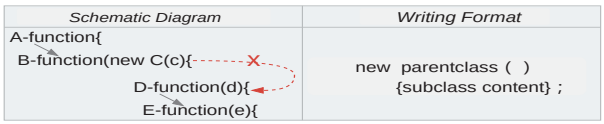


Fig. 7 A schematic diagram and writing format of anonymous internal classes

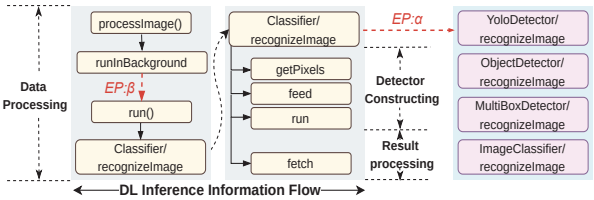


Fig. 8 Two enhancement patterns in the DL-based app *TensorflowDetect*

interface is defined, and then the *recognizeImage* API of the specific model is dynamically bound, which is example of the $EP\ \alpha$.

Answer to RQ3: We summarize two enhancement patterns that can effectively expand the analysis capability of DLtrace: the Polymorphism, and AIC. In the future, more advanced techniques (e.g., alias analysis) will be taken into account to realize a fully automated pipeline for DLtrace.

V. Conclusion

In this paper, we conduct the first empirical assessment to investigate the information flow in DL-based apps. To facilitate the assessment, we propose DLtrace, a static analysis tool for identifying information flow in DL-based apps, as well as a series of magic APIs to verify the functionality of DL models. We also expand the source and sink APIs in FlowDroid for comprehensive evaluation with the taint analysis techniques in these apps. Through the analysis of 208 popular DL-based apps collected from Google Play, we compare DLtrace with FlowDroid and find that DL-based apps are widely exposed to sensitive information leakage. In addition, DLtrace is more effective than FlowDroid in detecting these leakages during DL inference stage. FlowDroid analyzes the bytecode and configuration files of apps to find potential privacy leaks, either caused by carelessness or created with malicious intention. According to the same standard, we report the reachable paths between sensitive source and sink APIs. In the future, we will make further detailed analysis, which requires the cooperation of dynamic analysis technique, and will be our direction of efforts. Furthermore, these experimental results also demonstrate the inefficiency of existing security analysis methods when applied to DL-based apps. New security analysis methods targeting the characteristics of DL-based apps are desired. The findings of this study provide valuable insights to multiple stakeholders in mobile DL ecosystems, such as developers of DL-based apps and DL researchers. Although the static analysis for DL-based apps remains in its early stage, DLtrace is the starting point for evaluating the security of DL-based apps.

Acknowledgment

This paper was supported by the National Key Research and Development Program of China (No. 2021YFF1201102).

References

[1] A. B. Nassif, I. Shahin, I. Attili, M. Azzeh, and K. Shaalan, "Speech recognition using deep neural networks: A systematic review," *IEEE access*, vol. 7, pp. 19 143–19 165, 2019.
[2] Y. Li, "Research and application of deep learning in image recognition," in *2022 IEEE 2nd International Conference on Power, Electronics and Computer Applications (ICPECA)*. IEEE, 2022, pp. 994–999.
[3] D. W. Otter, J. R. Medina, and J. K. Kalita, "A survey of the usages of deep learning for natural language processing," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 2, pp. 604–624, 2020.
[4] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," *arXiv preprint arXiv:1710.09282*, 2017.
[5] R. J. Bolton and D. J. Hand, "Statistical fraud detection: A review," *Statistical science*, vol. 17, no. 3, pp. 235–255, 2002.
[6] M. Fredrikson, E. Lantz, S. Jha, S. Lin, D. Page, and T. Ristenpart, "Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 17–32.
[7] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "A first look at deep learning apps on smartphones," in *The World Wide Web Conference*, 2019, pp. 2125–2136.
[8] 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.
[9] X. Sun, X. Chen, K. Liu, S. Wen, L. Li, and J. Grundy, "Characterizing sensor leaks in android apps," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 498–509.
[10] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.
[11] É. Payet and F. Spoto, "Static analysis of android programs," *Information and Software Technology*, vol. 54, no. 11, pp. 1192–1201, 2012.
[12] F. Tong and Z. Yan, "A hybrid approach of mobile malware detection in android," *Journal of Parallel and Distributed computing*, vol. 103, pp. 22–31, 2017.
[13] 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.
[14] 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.
[15] P. Feng, J. Ma, C. Sun, X. Xu, and Y. Ma, "A novel dynamic android malware detection system with ensemble learning," *IEEE Access*, vol. 6, pp. 30 996–31 011, 2018.
[16] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe." in *NDSS*, vol. 15, no. 201, 2015, p. 110.
[17] 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.
[18] F. Wei, S. Roy, and X. Ou, "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.
[19] 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.
[20] Google. (2022) TFlite. [Online]. Available: https://www.tensorflow.org/lite
[21] Facebook. (2022) Caffe2. [Online]. Available: https://caffe2.ai/

[22] Apple. (2022) Core ML. [Online]. Available: https://developer.apple.com/cn/documentation/coreml/

[23] M. A. Ayub, W. A. Johnson, D. A. Talbert, and A. Siraj, "Model evasion attack on intrusion detection systems using adversarial machine learning," in *2020 54th Annual Conference on Information Sciences and Systems (CISS)*. IEEE, 2020, pp. 1–6.

[24] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 1322–1333.

[25] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks," 2017.

[26] S. Shen, S. Tople, and P. Saxena, "Auror: Defending against poisoning attacks in collaborative deep learning systems," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 508–519.

[27] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction {APIs}," in *25th USENIX security symposium (USENIX Security 16)*, 2016, pp. 601–618.

[28] B. Wang, Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Y. Zhao, "Neural cleanse: Identifying and mitigating backdoor attacks in neural networks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 707–723.

[29] Z. Li, C. Hu, Y. Zhang, and S. Guo, "How to prove your model belongs to you: A blind-watermark based framework to protect intellectual property of dnn," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 126–137.

[30] D. Hitaj and L. V. Mancini, "Have you stolen my model? evasion attacks against deep neural network watermarking techniques," *arXiv preprint arXiv:1809.00615*, 2018.

[31] Y. Huang, H. Hu, and C. Chen, "Robustness of on-device models: Adversarial attack to deep learning models on android apps," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2021, pp. 101–110.

[32] Z. Sun, R. Sun, L. Lu, and A. Mislove, "Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1955–1972.

[33] (2022) Amazon. [Online]. Available: https://docs.aws.amazon.com/zh_cn/personalize/latest/dg/personalize-dg.pdf

[34] (2022) Google. [Online]. Available: https://firebase.google.com/docs/ml-kit

[35] (2022) Microsoft. [Online]. Available: https://www.microsoft.com/en-us/ai