Understanding Malicious Cross-library Data Harvesting on Android

Jice Wang, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences; Indiana University Bloomington;* Yue Xiao and Xueqiang Wang, *Indiana University Bloomington;* Yuhong Nan, *Purdue University;* Luyi Xing and Xiaojing Liao, *Indiana University Bloomington;* JinWei Dong, *School of Cyber Engineering, Xidian University;* Nicolas Serrano, *Indiana University, Bloomington;* Haoran Lu and XiaoFeng Wang, *Indiana University Bloomington;* Yuqing Zhang, *National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences; School of Cyber Engineering, Xidian University; School of Computer Science and Cyberspace Security, Hainan University*

https://www.usenix.org/conference/usenixsecurity21/presentation/wang-jice

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

Understanding Malicious Cross-library Data Harvesting on Android

Jice Wang[1,2]*, *Yue Xiao*[2]*, *Xueqiang Wang*[2], *Yuhong Nan*[3], *Luyi Xing*[2]†,
Xiaojing Liao[2]†, *JinWei Dong*[4], *Nicolas Serrano*[2], *Haoran Lu*[2], *XiaoFeng Wang*[2], *Yuqing Zhang*[1,4,5]†
[1]*National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences,*
[2]*Indiana University Bloomington,*
[3]*Purdue University,* [4]*School of Cyber Engineering, Xidian University,*
[5]*School of Computer Science and Cyberspace Security, Hainan University*

Abstract

Recent years have witnessed the rise of security risks of libraries integrated in mobile apps, which are reported to steal private user data from the host apps and the app backend servers. Their security implications, however, have never been fully understood. In our research, we brought to light a new attack vector long been ignored yet with serious privacy impacts – malicious libraries strategically target *other vendors' SDKs integrated in the same host app* to harvest private user data (e.g., Facebook's user profile). Using a methodology that incorporates semantic analysis on an SDK's Terms of Services (ToS, which describes restricted data access and sharing policies) and code analysis on cross-library interactions, we were able to investigate 1.3 million Google Play apps and the ToSes from 40 highly-popular SDKs, leading to the discovery of 42 distinct libraries stealthily harvesting data from 16 popular SDKs, which affect more than 19K apps with a total of 9 billion downloads. Our study further sheds light on the underground ecosystem behind such library-based data harvesting (e.g., monetary incentives for SDK integration), their unique strategies (e.g., hiding data in crash reports and using C2 server to schedule data exfiltration) and significant impacts.

1 Introduction

Mobile apps today extensively incorporate third-party libraries (e.g., analytics, advertising, app monetization, or single-sign-on SDK), which enriches their functionalities but also brings in security risks. It has been reported that malicious SDKs stealthily collect private user data from the device running their host app [53, 73, 74, 78] (e.g., IMEI, GPS location, phone number, MAC address, SIM card ID, Android ID, etc.), the server or the cloud supporting the app [69]. With significance of such leaks, the security implications of library integration have yet been fully revealed: it is less clear whether a malicious library could endanger a user's sensitive information from other data sources, those not under the direct control of the affected app.

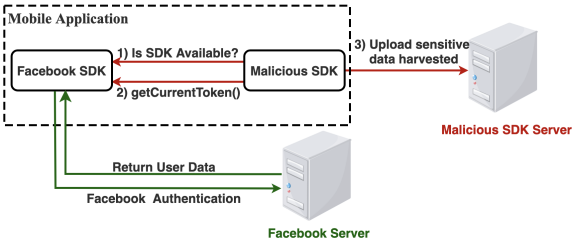


Figure 1: Workflow of cross-library data harvesting (XLDH)

Cross-library data harvesting. In our research, we discovered a type of data harvesting libraries never reported before, which strategically target the SDKs from other vendors also integrated by the host app. These SDKs carry sensitive user data. For example, the Facebook SDK extensively used by apps for single sign-on [34] also manages the information such as a user's name, birthday, locations she went to, social, health, and political groups she follows. The data could be exposed to the malicious library hosted by the same app integrating the Facebook SDK. Figure 1 illustrates such a malicious library, which first checks the presence of the Facebook SDK in its host app, and if so, invokes the Facebook API to acquire the user's Facebook session token and data. Since both the malicious library and the victim SDK co-exist within the same app, this invocation is not mediated. Given the wide deployment of the Facebook Login SDK (in more than 16% of the apps on Google Play [5]), the risk of such a data leak is significant. We call this attack *Cross-Library Data Harvesting* (*XLDH*).[1]

Beyond its threat to personal privacy, malicious data harvesting can also have serious social implications. A prominent example is the Cambridge Analytica scandal [17], in which the personal data of millions of Facebook users (profiles, page likes, current city, News Feed, etc. – enough to create psychographic profiles of the users) were collected and utilized for malicious political advertising [17]. *XLDH* also provides a new avenue for such political profiling and promotion, as discovered in our research (Section 5.4). Despite the importance of the problem, little has been done so far to understand

*The first two authors are ordered alphabetically. Work was done when Jice Wang was studying at Indiana University Bloomington.

†Luyi Xing, Xiaojing Liao and Yuqing Zhang are co-corresponding authors.

[1]In this paper, the terms "SDK" and "library" always refer to those developed by third-parties, i.e., vendors other than the host app vendor or OS vendor; also, we refer to "SDK" as the victim and "library" as a general term.

XLDH, not to mention any attempt to address this new security and privacy risk.

Finding XLDH in the wild. In this paper, we report the first study on *XLDH* on Android, aiming to understand its privacy and social impacts, underground ecosystem and challenges in controlling the threat. To this end, we developed a new, automatic methodology called *XFinder* to identify malicious libraries integrated in real-world apps on Google Play. Our idea is to discover restricted data managed by the SDKs and their *third-party data sharing policies*, which describes whether and how restricted data can be shared with or collected by *other libraries*. We automatically extract those policies from the *terms of service* (ToS, a.k.a., terms of use, terms and conditions) released by the SDK vendors, and then analyze the code of each integrated library to find out whether it makes any access to the SDK's data in violation of these policies. This turns out to be nontrivial due to the challenges in analyzing ToS to recover its semantics and evaluating apps to find cross-library interactions.

More specifically, unlike app privacy policies that protect known sensitive content (e.g., address, contact, etc.) and therefore can be identified by existing privacy policy analyzer such as Polisis [57], ToS describes restricted data whose security or privacy implications can only be determined from the context of their usage. Examples include *security-critical data* such as password and token, and *SDK-specific sensitive data* such as `utdid` used by Alibaba for identifying user devices [2], page likes, health or political groups of a user recorded by Facebook, and education and project information maintained by LinkedIn [18]. More challenging is to recover the data sharing policies from ToS that specify the restrictions on collecting and sharing different data items, which tends to be complicated. For example, Google allows developers to access advertising ID or device identifier (e.g., ssaid, mac address, imei), but restricts the collection of these two data item simultaneously; also Facebook user's page likes, timeline, etc., are open to the apps certified by Facebook [15], but not to other parties (including third-party libraries) [55], while Facebook user ID and password are not allowed to be sent out to the Internet by any party. Our research shows that existing techniques like Polisis [57] and PolicyLint [45] cannot be directly adopted for ToS analysis (see the evaluation in Section 3.2).

To address these challenges, *XFinder* utilizes a semantic analysis tuned towards the unique features of ToS, which leverages natural language processing techniques to capture sensitive data items and to recover complicated policies (Section 3.2). Further, our code analyzer module in *XFinder* is designed to handle potential evasion tricks played by malicious libraries when evaluating its interactions with a target SDK (Section 3.3). Our experiment shows that *XFinder* achieved a high precision of 86% and successfully detected 42 malicious libraries from more than one million Android apps.

Measurement and discoveries. From 1.3 million Google Play apps analyzed in our research, we are surprised to find the significant impacts of the new threat. More specifically, we discovered 42 distinct libraries that stealthily harvest data from third-party SDKs without a user consent. These libraries have been integrated into more than 19K apps, with a total of 9 billion downloads. The data harvested are highly sensitive, including access tokens, profile photos, and friend lists (see Section 5). As an example, `OneAudience`, a library integrated in more than 1,738 apps with more than 100 million users, collects users' private data from Facebook and Twitter SDKs. Based on a press release from Nielsen [29], OneAudience shared mobile user data with Nielsen – a marketing research firm, and the data can be used by Nielsen's customers for political marketing purpose, among other marketing usages. Hence, we suspect that the data harvesting campaign might lead to a Cambridge-Analytica-like political scandal if they were taken advantage of by the adversary. Although the campaign has been stopped after we reported it to Facebook (see below), already millions of Facebook users' data have been exposed, since the library has been continuously gathering user data, once per hour on both Android and iOS since 2014.

Also interesting is the ecosystem behind *XLDH*, which includes library distribution, stealthy data exfiltration channel, and data monetization. In particular, XLDH vendors are found to distribute their libraries through multiple channels, including colluding with free app building services, integrating into popular libraries, and offering app monetization (Section 5.4). For example, app monetization is used to attract app developers to integrate problematic libraries into their apps: app developers that integrate `OneAudience` and `Mobiburn` are paid $0.015 to $0.03 per app install. Furthermore, we revealed the techniques used by malicious libraries that made their data harvesting activities more stealthy and harder to detect, such as the abuse of Java reflection technique (see Section 3.3).

Our study also sheds light on the challenges in eliminating the *XLDH* risk. We found that although VirusTotal and Google Play are able to detect the libraries collecting data from mobile devices (such as IMEI, contact), they all failed to detect *XLDH* libraries and the apps integrating them, possibly due to the challenges in determining third-party data sharing policies and non-compliance with the policies. This has been addressed by *XFinder*. We reported our findings to affected parties, including Facebook, Twitter, Google Play and others, who are all serious about this new risk and expressed gratitude for our help with bounty programs. Google asked the developers of affected apps to remove the malicious libraries, or drop these apps to control the risk. Facebook and Twitter have taken legal actions to take down `OneAudience`, a XLDH library owned by Bridge, a digital marketing company.

Contributions. We summarize the contributions as follows.

• Our study brings to light a new attack vector that has long been ignored, yet with serious privacy implications: malicious libraries aiming at third-party SDKs integrated in the same

apps to harvest private user data. Our findings demonstrate the significant privacy and social impacts of this new threat. Our works also help better understand the underground ecosystem behind it, and the challenges in controlling the risk.

• Our study has been made possible by a novel methodology that automatically identifies *XLDH* from over a million Android apps, through semantic analysis on ToS and code analysis on cross-library interactions.

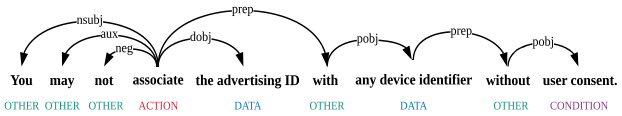
• We release the dataset used in this research and our source code for the automatic ToS analysis online [39].

2 Background

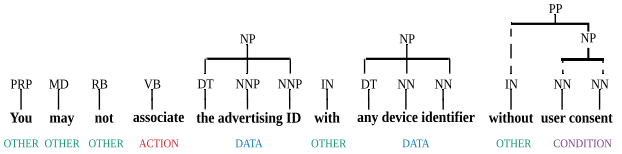
Cross-library API calls. Like an app that calls functions of a library, libraries in an app naturally can invoke the functions of another library. On Android, this is typically done through first explicitly importing the package name of the callee class (in Java), and then invoking the target function through the callee class' instance. Further, Java features a technique called reflection [26]. that allows function invocation in a more flexible manner. As illustrated in Figure 4a, to invoke a function `getCurrentAccessToken` in the Facebook library, one can first obtain a class object through Java reflection API `Class.forName`, by providing the class name (`com.facebook.AccessToken`); then through another reflection API `getDeclaredMethod`, one can obtain a method object using the name of the target function `getCurrentAccessToken`; last, calling reflection API `invoke` on the method object, one can invoke the target function. In our research, we observed that XLDH libraries often leverage reflection to call victim libraries, likely for making the behaviors more stealthy. *Note that Android provides a coarse-grained sandbox and permission model to regulate third-party libraries, allowing them to operate with the same permissions as their host apps [4, 76]. In particular, there is no security boundary between libraries within the same app, allowing one library to access another (e.g., invoking functions) without restrictions.*

SDK terms of service. Term of service (ToS) is an SDK developer document that lays out terms, conditions, requirements, and clauses associated with the *use* of a mobile SDK, e.g. copyright protection, accounts termination in the cases of abuses, data usage and management, etc. Note that in addition to the ToS for developers, an SDK vendor (e.g., Facebook and Twitter) may also have a ToS for regular users, such as [16], which is outside the scope of our study. In our research, we manually collected 40 ToSes from SDK vendors' developer websites to investigate the XLDH risks.

Unlike privacy policy, which aims at informing *end-users* about collection and use of personal data (e.g., name, email address, mailing address, birthday, IP address), ToS specifies rules and guidelines for *developers* who uses an SDK, as illustrated in Figure 2. Also, data protected under privacy policy



(a) Dependency parsing tree



(b) Constituency parsing tree

Figure 2: The dependency parsing tree and constituency parsing tree of the sentence: "you may not associate the Advertising ID with any device identifier without consent from the end user."

is different from that covered by ToS. The former is usually "personal information" (e.g., name, email address, mailing address), as guided by laws (e.g., GDPR, CalOPPA [48]). The latter also prevents the abuse of security-critical data (e.g., password and token) and SDK-specific data (e.g., API keys, access credentials). Table 1 shows the data items protected by the ToS of Twitter, Facebook and Google. We can see that 21 data items in the ToSes are SDK-specific and not mentioned by the privacy policies.

In our research, we found that the state-of-the-art privacy policy analyzer (e.g., Polisis [57]) cannot effectively analyze ToS to recover the content about sensitive data sharing policy (see Section 3.2), possibly due to the different grammatical structures of ToS (for addressing to different audience and describing different data items and rules) than those appearing in common privacy policy corpora.

Natural language processing. In our research, we leverage Natural language process (NLP) to automatically extract third-party sharing policies for sensitive data from SDK ToS. Below we briefly introduce the NLP techniques used in our research.

• *Named entity recognition*. Named entity recognition (NER) is a technique that locates named entities mentioned in unstructured text and classifies them into pre-defined categories such as person names, organizations, locations. The state-of-the-art NER tools such as Stanford NER and Spacy NER can achieve a 95% accuracy on open-domain corpora to recognize person names, organizations, locations. However, NER systems are known to be brittle, highly domain-specific — those designed for one domain hardly work well on the other domain [61]. A direct use of the state-of-the-art tools like Stanford NER [66] does not work, because the common predefined categories (names, organizations, locations) are not suitable for our task. In our study, we tailor named entity recognition techniques to identify sensitive data, which is protected by third-party sharing policies.

Table 1: Examples of data items protected by the ToS of Facebook, Twitter, and Pinterest

SDK	Term of Service
Facebook	access token, access credentials, Friend data, Facebook user IDs, trademarks, PSIDs(Page-scoped user IDs), Marketplace Lead Data
Twitter	API keys, access credentials, Twitter Content, Twitter passwords, Tweet IDs, Direct Message IDs, user IDs, Periscope Broadcasts
Pinterest	Wordmark, image, Ad Data, user ID and campaign reporting, secret boards

• *Constituency parsing and dependency parsing*. Constituency parsing and dependency parsing are NLP techniques to analyze a sentence's syntactic structure. Constituency parsing breaks a sentence into sub-phrases and displays its syntactic structure using context-free grammar, while dependency parsing analyzes the grammatical relations between words such as subject-verb (SBV), verb-object (VOB), attribute (ATT), adverbial (ADV), coordinate (COO) and others [11]. Figure 2 illustrates the constituency parsing tree and dependency parsing tree of a sentence. In the constituency parsing tree, non-terminals are types of phrases and the terminals are the words in the sentence. For instance, as shown in the figure 2b , NP is a non-terminal node that represents a noun phrase and connects three child nodes (the (DT), advertising (NNP), ID (NNP)). Here, DT means determiner and NNP means a noun in a singular phrase [30]. By comparison, the dependency parsing tree is represented as a rooted parsing tree (see Figure 2a). At the center of the tree is the verb of a clause structure, which is linked, directly or indirectly, by other linguistic units. This unit can either be a single word or a noun phase that merged by the parser's built-in phase merge API `phrase.merge` [28]. The state-of-the-art dependency and constituency parser (e.g., Stanford parser [49], AllenNLP [56]) can achieve over 90% accuracy in syntactic structure discovery from a sentence. In our study, we leverage both dependency and constituency parsing trees generated from sentences in ToS to recover the semantics of third-party sharing policies in SDK ToS.

• *word2vec*. Word2vec [67] is a word embedding technique that maps text (words or phrases) to numerical vectors. Such a mapping can be done in different ways, e.g., using the continual bag-of-words model [8] or the skip-gram technique [35] to analyze the context in which the words show up. Such a vector representation ensures that synonyms are given similar vectors and antonyms are mapped to different vectors. In our study, we build a customized word embedding model for data sharing policies to measure the similarity of words in this domain, as elaborated in Section 3.4.

Threat model. We consider an adversary who spreads malicious libraries that harvests private user data from *third-party SDKs* hosted by the same mobile apps. For this purpose, the adversary often offers appealing functionalities or monetary incentives to app developers for integrating a malicious library into their apps. In our study, victim SDKs in an app are those neither owned by the app vendor, nor provided by AOSP (Android Open Source Project) [42] – the official Android version not customized by original equipment manufacturers (OEMs). In this regard and for the best understanding of the threats, libraries developed by Google but not on AOSP are also studied in our research.

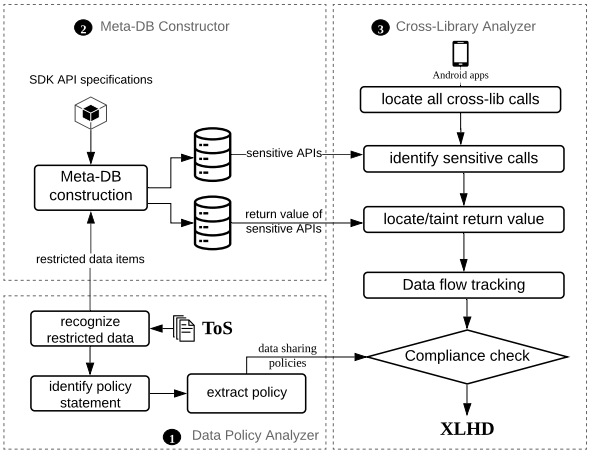


Figure 3: Overview of *XFinder*

3 Methodology

In this section, we elaborate on the design and implementation of *XFinder*, a methodology for discovering *XLDH* from real-world Android apps.

3.1 Overview

Architecture. As mentioned earlier, our approach relies on the extraction of data sharing policies from ToS and analysis of policy compliance during a library's interactions with other SDKs within the same app. In particular, the design of *XFinder* includes three major components, *Data Policy Analyzer* (*DPA*), *Meta-DB Constructor*, and *Cross-library Analyzer* (*XLA*), as outlined in Figure 3.

DPA takes as its input a set of SDK ToSes associated with popular SDKs, which are widely deployed in Android apps. These ToSes are processed by *DPA* to output the SDKs' data sharing policies (Section 3.2). The restricted data items governed by those data sharing policies, along with the corresponding SDK APIs that return those data items, are recorded in a Meta-DB (Section 3.4). To identify the leaks of those restricted data due to XLDH activities, *XLA* inspects the de-

compiled code of an app to find all cross-library invocations on the sensitive SDK APIs (those that return the restricted data, as recorded in the Meta-DB); *XLA* then tracks down the data flow of their return value to identify an exfiltration (Section 3.3). The cross-library interactions discovered in this way are then checked against the data sharing policies that *DPA* extracted, for finding policy violations.

Example. Here we use a real example to explain how *XFinder* works. Specifically, *XFinder* inspected the app "Columns Gembira" (`com.frzgame.columns`), which includes both an XLDH library `Mobiburn` and the Facebook SDK. In analyzing the app, *XLA* first scans all function calls to find out cross-library API calls, those with caller classes and callee classes in different libraries. For example, the class `com.mobiburn.e.h` in `Mobiburn` library is found to invoke function `com.facebook.AccessToken.getToken()` in the Facebook SDK, as shown in Figure 4a. Then *XLA* looks up the *meta-DB* to determine the return value of the function, which is the user's Facebook session token, and further tracks down the data flow using taint tracking. In the end, we found that the token is used to fetch a user's Facebook profile data (ID, name, gender, email, locale, link, etc.) in function `com.mobiburn.e.h.getFbProfile()`, and all the data including the token are sent out to the server of `Mobiburn`. (see Figure 4b)

XFinder then checks whether such a data practice violates the data sharing policies specified in Facebook ToS. More specifically, given the statement of "*keep private your secret key and access tokens*" in Facebook's ToS, *DPA* automatically extracted the data sharing policy *(access token, condition:null)*, which indicates that *access token* is the restricted data item and it cannot be shared with and transferred to a third-party under any conditions (i.e., *condition:null*). Hence, the XLDH of the `Mobiburn` library violates the data sharing policy of Facebook SDK, and thus, *XFinder* flags `Mobiburn` as an XLDH library.

Dataset summary. We summarize the dataset produced and consumed by each stage of our pipeline as below. Table 2 shows the datasets used in our study.

In total, we collected 1.3M Android apps (D_g) from Google Play for XLDH library detection. More specifically, the dataset was collected based on a publicly-available app list (AndroZoo [43]), using an open-source Google Play crawler [23], which has been widely used in previous research such as [87]. We used the default settings of the crawler to download the apps from Google Play from Oct. 03 to Oct. 15, 2019. In total, we successfully collected 93.12% of the apps on the list (1,341,148/1,440,160) from Google Play. Among them, we identified top 200 SDKs widely integrated into real-world apps (Section 3.4). After removing utility SDKs which are not associated with sensitive data, we further gathered ToSes for the remaining 40 SDKs from their vendor websites (C_{tos}).

```
public class h {
    public static String getAccessToken() {
        Class[] param = new Class[0];
        Class clz = Class.forName(
                "com.facebook.AccessToken");
        Method meth1 = clz.getDeclaredMethod(
                "getCurrentAccessToken", param)
        Object curToken = meth1.invoke(clz, null);
        Method meth2 = clz.
            getDeclaredMethod("getToken", param)
        return meth2.invoke(curToken, null);}


    public JSONObject getFbProfile(String token){
        String uri = Uri.parse(
            "https://graph.facebook.com/v2.10/me").
            appendParam("accesstoken", token).
            appendParam("fields","id,first_name,gender,
            last_name,link,locale,name,timezone,
            updated_time,verified,email");
        HttpsURLConnection httpsURLConnection =
            new URL(uri).openConnection();
        return new JSONObject(httpsURLConnection.
            getInputStream().readLine());}
    }
```

(a) Reading app users' Facebook access token and profile

```
 public class f{
    public void a(){
    JSONObject userData = new JSONObject();
    userData.put("accessToken", getAccessToken());
    userData.put("accountJson", getFbProfile());
    ...
    HttpsURLConnection httpsURLConnection =
        new URL(this.serverUri).openConnection();
    DataOutputStream dataOutputStream =
        httpsURLConnection.getOutputStream();
    dataOutputStream.write(userData);
    }
}
```

(b) Sending the Facebook token and profile to mobiburn server

Figure 4: Code of XLDH library *com.mobiburn*

After that, we bootstrapped our study by using *DPA* to automatically extract 1,056 data sharing policies, associated with 1,215 restricted data objects from the 40 SDK ToSes (Section 3.2). We constructed the *Meta-DB* (Section 3.4) which recorded all 936 sensitive APIs of the SDKs that return restricted data. Then, in *XLA*, we statically analyzed 1.3M Android apps (D_g) to extract cross-library API calls (Section 3.3). After filtering by Meta-DB, 1,934,874 of them are regarded as sensitive. Given those sensitive API calls, we tracked their data flows to check whether such flows are in compliance with the SDK's data sharing policies. In particular, for restricted data not allowing access by a third-party or any party, we consider the exfiltration of the data a violation of the ToS and identify 15 XLDH libraries; For restricted data access requiring user consent or complying with regulations, we check whether such behavior was disclosed in the caller library's privacy policy (C_p), which revealed 27 XLDH libraries. In total, our study reported 42 distinct XLDH libraries (4 manually found and 38 automatically detected) integrated in more than 19K apps and targeting at 16 victim SDKs.

Table 2: Summary of datasets and corpora

Name	Source	Size	Timestamp (yyyyMM)	Usage
D_g	Google Play	1.3M apps	201910	Detection
C_{tos}	40 victim SDK ToSes	8622 sentences	201910	Detection
C_{api}	40 victim SDK API specifications	Documentations of 27K APIs	201910	Detection
C_p	73 XLDH library privacy policies	10K sentences	202006	Detection
D_{ha}	Historical Google Play apps	300K apps	2014-2019	Measurement
D_{hl}	Historical XLDH library versions	42 XLDH libraries of 495 versions	2011-2019	Measurement

Table 3: The verbs related to data sharing policy

connect, associate, post, combine, lease, disclose, offer, distribute, afford, share, send, deliver, disseminate, transport, protect against, keep, proxy, request, track, aggregate, provide, give, transfer, cache, transmit, get, seek, possess, accumulate, convert, collect, use, store, gather, obtain, receive, access, save

3.2 Data Policy Analyzer

The goal of *Data Policy Analyzer* (*DPA*) is to extract third-party data sharing policies from an SDK ToS, which describe how restricted data items can be shared with or collected by other libraries. Here we describe a data sharing policy as a pair $(object, condition)$, where $object$ is the restricted data item of the SDK, such as utdid, password, and $condition$ is the requirements and clauses for the operations on the restricted data, which can be empty. For example, the policy statement "the advertising identifier must not be associated with any persistent device identifier without explicit consent of the user", can be represented as *(advertising ID and device ID, user consent)*. Note that in our study, we focus on ToS data sharing policies, and thus the subject of such a policy is the library developers (and their libraries) that call the target SDK and the operation on the restricted data is `GET`.

During the analysis, our approach first runs a NER model to recover restricted data items. More specifically, the NER is customized on the ToS corpora and the entity category of restricted data items (e.g., utdid, password) using an efficient constituent parsing technique. Then, based on the restricted data items, we identify the sentences related to third-party data sharing policies from the ToS. After that, we extract the pair $(object, condition)$ from the data sharing policies using restricted data as "anchors" to recognize the pattern of each policy's grammatical structures and to locate the condition on data sharing. We elaborate on our methodology as follows.

Restricted data object recognition. As mentioned earlier, identifying restricted data from an SDK ToS is an NER problem. Unfortunately, NER techniques today are known to be highly domain-specific [63]: open-domain NER model does not work well on the security corpora, as restricted data are different from the common entity categories (e.g., location, people, organization) whose annotated datasets are available. In our study, we observe that restricted data in the SDK ToS is often characterized by a long noun phrase (e.g., Google Advertising ID, Facebook password, Amazon purchase history) covered by a single or multiple consecutive noun phrases in the constituency tree (Figure 2) . Therefore we can utilize the features of the constituency tree to help identify such a phrase as an entity.

More specifically, we include the constituency tree of a sentence as a feature, which enables our NER model to learn that certain types of phrasal nodes, such as NPs, are more likely to be entities, i.e., restricted data. Hence, we crafted several features based on constituency parsing tree tags for each word, which include a word's tag, its parent tag, the left and right siblings, the location of the word in the span of NPs nodes. For example, as shown in Figure 2, the word "advertising" in the NP span "the advertising ID" in has 5 features: its tag "NNP", its parent tag "NP", its left sibling "DT: the", its right sibling "NNP: ID" and its position of 1 under the span. Such features help the model to learn and inference similar long noun phrase (eg., "the Twitter ID").

In our implementation, we utilize `AllenNLP` constituency parser [56] to generate the constituency tree related features for each sentence. Then, we built these features into the state-of-the-art conditional random fields (CRF) based NER model - Stanford NER [66]. As these features are not built-in features [37] in Stanford NER, we configure the feature variables of them using the `SeqClassifierFlags` class, and then read the feature set into the `CoreLabel` class. In addition, we updated training data using SDK ToSes. Particularly, we manually annotated 534 sentences from 6 SDK documents using IOB encoding [25] to retrain the NER model.

To evaluate the model, we perform 10-fold cross validation on the annotated sentences. Our result shows that by leveraging constituency tree features, the model achieves a precision of 95.2% and a recall of 90.8%. Compared with the model without constituency tree features, our model shows a increase of 1.3% and 2.1% for the precision and recall, respectively. After that, the model was also evaluated on additional 103 randomly selected and manually annotated sentences from two *previously-unseen* SDK ToSes, which yields a precision of 88.2% and a recall of 90.4%.

Policy statement discovery. From each ToS, the analyzer identifies the sentences describing how restricted data items can be shared with or collected by other libraries. These sentences are selected based on restricted data identified by the

aforementioned model, the subject (i.e., library developer) and the operation (i.e., `GET`) they contain.

We first need to construct the keywords list associated with data collection and sharing (e.g., use, collect, transfer, etc.). For this purpose, we leverage the OPP-115 [86] and APP-350 [87] datasets, which contain 46,259 manually annotated privacy policy statements. Among them, 14,100 annotated sentences are related to first-party collection and third-party collection. After that, we use constituency parser [56] to recognize the verb in the verb phrases and further identify the lemma of a verb from those sentences. In this way, we collect 38 keywords related to data collection and sharing as shown in Table 3.

After that, we use this keywords list in Table 3 and the restricted data to filter out the sentences irrelevant to data sharing and collection policy. Specifically, after parsing the HTML content of each ToS and splitting the text into sentences, we run our NER model to find all statement sentences that contain restricted data. Then we leverage dependency parser [71] to locate the verb or nominal modifier of restricted data. In particular, if (1) the dependency relationship between them is the `direct object` (e.g., collect personal information) or `nominal modifier` (e.g., the usage of personal information) and (2) such verb or nominal modifier is in the keywords list, we will regard the sentence describing data sharing and collection policy.

After this, we check that the sentence subject is not the SDK itself but library developer. Specifically, we build a dependency parsing tree to recognize the subject of a sentence. We eliminate sentences with the "first-party" as the subject. For example, the target SDK's name (e.g., "Twitter", "Facebook") and first-person plural (e.g., "we", "us"). Note that for the sentences with an ambiguous subject reference (e.g., "it", "this", "that"), we run a co-reference resolution tool [56] on the paragraph in which the sentence exists to identify the subject.

Altogether, we gathered 1,056 sentences associated with data sharing policy from 40 ToSes. By manually inspecting 200 sentences, we found that our method yields the recall of 89.3%. The miss-reported sentences are mainly due to wrong dependency parsing results from underlying tools we use. For example, "store non-public Twitter content" is parsed as a noun phrase instead of verb-object phrase by [71].

Data sharing policy identification. To extract the pair $(object, condition)$ from the policy statements of an SDK ToS, our approach first uses a dependency parser [58] to transform a sentence into a dependency parsing tree, which describes the grammatical connections between different words, and then leverages the restricted data *object* as known anchors to locate the *condition* by traversing the parsing tree. Here we prune the dependency tree of the policy statement into a subtree that represents the grammatical relation among *object*, *condition* and the operation (e.g., "transfer", "use"). This is because that the policy statement is usually long and consists of noisy information, and the subtree is most relevant to the understanding of the relation.

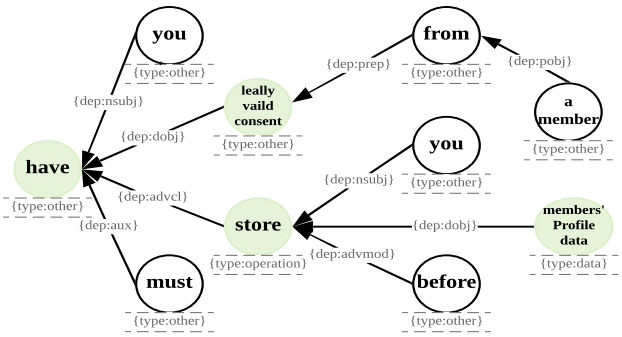
• *Object identification*. In our research, we observe *object* in the data sharing policy sometimes consists of more than one restricted data, e.g., *"Don't collect usernames or passwords"*. Hence, to extract the *object* from each policy statement, we first identify the restricted data $d_1, d_2, ..., d_n$ using the aforementioned method, and then use dependency tree to determine whether they have *conjunctive* relation and their coordinating conjunction (a.k.a., CCONJ [32]) is "OR". If so, we recognize them as n different *objects*.

Similarly, for the restricted data $d_1, d_2, ..., d_n$ are with *conjunctive* relation but their coordinating conjunction (a.k.a.,CCONJ [32]) is "AND", we recognize them as one *object*. However, things get complicated when the policy statement illustrates multiple objects can not `GET` at the same time, e.g., *Don't associate user profiles with any mobile device identifier*. Here, we use specific verbs (e.g., associate with, combine with, connect to) to identify this relationship. In this way, we recognize them as one *object*, i.e., $d_1 \wedge d_2 ... \wedge d_n$.

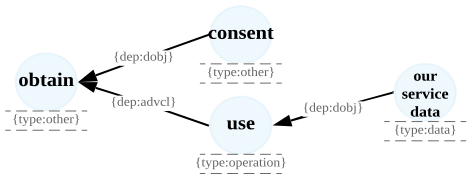
In addition, we use the lexicosyntatic patterns discovered in [45] to find the object hyponym and then use the specified object hyponym in the policy tuple. For example, given the pattern "X, for example, Y_1, Y_2,...Y_n ", where Y_1, Y_2,...Y_n is the hyponym of X, and the sentence "device identifier, for example: ssaid, mac address, imei, etc", we will extract five policies of "device identifier", "ssaid", "mac address" and "imei".

• *Condition extraction*. By manually inspecting 1K sentences from 10 SDK ToSes, we annotated 14 generic patterns (in terms of dependency trees), which describe the grammatical relation among *object*, *condition* and the operation. The annotated pattern list is shown in Table 9. Then, we fed them into the analyzer which utilizes these patterns to match the dependency parsing trees of the policy statements, using the *object* and *operation* nodes as anchors. More specifically, given a policy statement, we use the depth-first search algorithm, which starts at *object* and *operation* nodes, to extract all subtrees for pattern similarity comparison. Then we identify the most similar subtree of a policy statement by calculating a dependency tree edit distance between each subtree and the patterns in Table 9. Here we define a dependency tree edit distance $D(t_1, t_2) = \min_{(o_1,...,o_k) \in O(t_1,t_2)} \sum_{i=1}^{k} o_i$, where, $O(t_1, t_2)$ is a set of tree edits (e.g., node or edge's insertion, deletion and substitution) that transform t_1 to t_2, and we consider t_1 and t_2 are equal when all node types and edge attributes are matched. After that, we locate the condition node based on the matched subtree.

For example, Figure 5a illustrates the dependency tree structure of the policy statement. In the tree, each edge has an attribute `dep` that shows the dependency relationship between nodes, and each node has an attribute `type` which indicates whether it is *object*, *operation*, or none of the above (*other*).



(a) Tree structure of policy statement



(b) Tree structure of matched pattern

Figure 5: A example of condition extraction with the policy statement "You must have legally valid consent from a Member before you store that Member's Profile Data"

The subtree which consists of all green nodes is the most similar subtree with 0 edit distance to the pattern shown in the Figure 5b Traversing the matched pattern, we can locate condition node which is *(have legally valid consent)*.

As the graph matching problem is *NP-complete*, the computation time grows dramatically with the increase of node and edge. In our implementation, to reduce the matching time, when constructing subtrees using depth-first search, we define the search depth which will not exceed twice of pattern length and the threshold of edit distance's threshold to be 3.

Discussion and evaluation. *DPA* recognized 1,215 pairs (*object*, *condition*) from 1,056 policy statements from 40 ToSes. We manually inspect all of the detected pairs based on the relevant policy statements. The results show that our method achieves a precision of 84.2%. However, still our technique misses some cases. We acknowledge that the effectiveness of the method can increase with more annotated sentences for pattern matching. In our research, to guarantee the diversity of the annotated patterns, we design a sample strategy for annotated sentence selection. Specifically, we keep randomly sampling sentences for annotation until not observing new patterns for continuous 200 sentences. In this way, we annotated 14 patterns by inspecting 1K sentences from 10 SDK ToSes. Another limitation is the capability to process long sentences. Our method utilized the dependency parsing trees of the sentences for condition extraction. However, the state-of-the-art dependency parser cannot maintain its accuracy when sentences become too long.

Analyzing these 1,215 pairs of data sharing policies, we found that most are from Facebook SDK (9.4%), followed by Amazon (8.8%) and LinkedIn (7.3%). Also, 37 of them have the *object* with more than one restricted data. We observe the objects *advertising ID* and *mobile device identifier* always co-occur (7 pairs), because the user-resettable advertising ID will be personally identifiable when associated with mobile device identifier, which is not privacy-compliant [9]. To understand the data sharing conditions, we manually analyzed all of the recognized data sharing policies and categorized them into five types: *No access by any party*, *Requiring user consents*, *No third-party access*, *Complying with regulations (i.e., GDPR, CCPA, COPPA)* and *Others*, as shown in Table 5. Note that 96% of the data sharing policy are with the first four types of data sharing conditions. The *Others* type of data sharing condition is rarely observed and sometimes associated with some vaguely-described condition, such as "Only certain application types can access." In our policy compliance check (see Section 3.3), we did not check the policy compliance related to this type of condition. We acknowledge that checking those policies would allow for a more holistic view of XLDH activities. However, doing so will require subjective analysis of the vague and ambiguous policies and a large amount of manual efforts for corner cases.

Comparison with other policy analyzers. Since there is no public-available ToS analyzer, we compared *DPA*'s policy statement discovery with two state-of-the-art privacy policy analyzer Polisis [57] and PolicyLint [45].[2] Note that Polisis and PolicyLint are designed for privacy policy analysis, not ToS analysis.

More specifically, we manually annotated 200 sentences from 3 SDK ToSes (e.g., Twitter, Google, Facebook), which yielded 83 sentences are associated data collection and sharing policy and the rest (117) are not. In our experiment, we evaluated the approaches on this dataset. Table 4 shows the experiment results. Our study shows that *DPA* outperforms both approaches in precision and recall.

We also compared *DPA*'s restricted data object recognition module with PolicyLint [45]. We use the aforementioned 534 IOB-encoded sentences to evaluate both *DPA* and PolicyLint via 10-fold cross validation. For PolicyLint, we retrained its NER model using our annotated corpora. We also show the performance of PolicyLint with its original model. Table 4 shows the precision and recall of both approaches. Our study shows that *DPA* has a much better recall (90.8% vs 82.7%).

3.3 Cross-library Analysis

To capture malicious data harvesting in an app, our *Cross-library Analyzer* (*XLA*) identifies cross-library API calls to find the data gathered by a third-party library from a co-

[2]PolicyCheck [46] shared the same policy analysis module with PolicyLint.

Table 4: Comparison with Polisis and PolicyLint/PolicyCheck

Tasks	Tools	Precision	Recall
Statement finder	XFinder	87.2%	89.3%
	Polisis	27.5%	19.2%
	PolicyLint	71.4%	25.3%
Restricted data detector	XFinder	95.2%	90.8%
	PolicyLint (original)	82.2%	71.3%
	PolicyLint (retrained)	86.5%	82.7%

located SDK (within the same app), and then checks the compliance of such activities with the SDK's data sharing policies recovered by *DPA* (Section 3.2). To this end, *XLA* runs a program analysis tool that integrates existing techniques.

Locating cross-library API calls. *XLA* looks for cross-library calls by walking through the call graph generated by FlowDroid [47]. Specifically, each node in the graph represents a function and carries the information about the function's class and package (according to Java's reverse domain name notational convention [33]); each edge (with direction) describes a call from the caller node to the callee node. On the graph, *XLA* identifies cross-library calls by comparing the package names of the caller and callee class: if their top and second level domains (1) do not match with each other, and (2) do not match the host app's package name, the call is considered cross-library – an approach also used by MAPS [87].

Also, a cross-library call can leverage Java's reflection to implicitly trigger a function (see the example in Figure 4a). Hence, *XLA* inspects all reflection calls on the call graph, and checks whether the caller and callee classes belong to different libraries. To this end, *XLA* first locates reflection calls from a set of call patterns (see Table 6). As shown by a recent study [62], these patterns cover the most common reflection use cases in Android apps. Further, our approach recovers the callee's class name and method name from the arguments passed to the reflection functions. For example, the argument of `Class.forName(target_class_name)` indicates the callee class name, e.g., `com.facebook.AccessToken` in Figure 4a. A problem here is, the argument could be a variable. To find its value, *XLA* utilizes DroidRA [62], an inter-procedural, context-sensitive and flow-sensitive analyzer dedicated to resolve reflection calls, to track the string content propagated to the variable.

Identifying cross-library leaks. With the discovered cross-library calls, *XLA* then identifies the restricted data items returned to the caller library, and performs taint tracking to detect potential data exfiltration (to the Internet) by the caller library. In particular, *XLA* leverages *Meta-DB* to recognize restricted data items being returned , as *meta-DB* recorded which are the sensitive SDK APIs and the restricted data they return (see Section 3.4).

Further, we need to track down the data flow of the restricted data. Instead of directly using the techniques of FlowDroid (e.g., with deep object sensitivity), which is considered heavy-weight for an analysis of 1.3M apps [77, 87], we need a relatively light-weight tool. Hence, we opt for existing taint track techniques that are capable of inter-procedural analysis, field-sensitive but not object-sensitive. We take the return value of the cross-library calls as the taint source, and networking APIs as the sink. For example, the return value of the reflection call on `com.facebook.AccessToken.getToken()` is a taint source including Facebook user's session token (Figure 4a); once the data reaches a sink in the caller library, e.g., OutputStream.write(String.getBytes()) API to send the data to the Internet, *XLA* reports a potential data exfiltration.

Checking policy non-compliance. Given a potential exfiltration of the restricted data from a victim SDK, we check whether it violates the ToS policy of the target SDK (obtained by *DPA* in Section 3.2). Depending on the conditions with which the ToSes restrict the access to individual data items, our approach for a compliance check is as follows.

• *No third-party access; no access by any party*. If the ToS (e.g., those of Facebook, Twitter and Pinterest) prohibits an access to the data by a third-party (e.g., a third-party library or its vendor) or by any party (e.g., Facebook user ID and password are not even allowed to be exfiltrated/stored by the host app vendor), we consider the exfiltration of the data a violation of the ToS – an XLDH activity is identified.

• *Requiring user consents; complying with regulations*. Some ToSes ask that the access to certain data items should require a user consent or comply with privacy regulations (i.e., GDPR, CCPA, COPPA). In XLDH, data sharing and collection occur between caller library and victim SDK without being processed by the host app. Hence, we consider the caller library to be a data controller [19], which has obligations to comply with regulations and disclose the data practice in its privacy policy [19]. In our study, we check the privacy policy of the caller library to determine whether it discloses the data collection and sharing behaviors in its privacy policy. To automatically analyze the privacy policy, we use PolicyLint [45] to extract privacy policy tuples *(actor, action, data object, entity)* associated with that restricted data. Here the tuple *(actor, action, data object, entity)* illustrates *who* [actor] *collects/shares* [action] *what* [data object] with *whom* [entity], e.g., "We [actor] share [action] personal information [data object] with advertisers [entity]". In our study, we care about the tuples with caller library as *actor*, share/collect as *action* and the restricted data as *entity*. Note that for non-English privacy policies which PolicyLint [45] can not handle, we translate them into English for further processing.

Discussion. Recent studies such as [80, 87] on privacy compliance considered the data as leaked out once an API returning the data is invoked by an unauthorized party. We found this is imprecise in detecting XLDH, due to the pervasiveness of service syndication (e.g., Twitter4j, Firebase Authentication) in which a benign library wraps other SDKs (Facebook login, Twitter login) to support their easy integration into apps. Such

Table 5: Summary of 40 vendor's data sharing policies

Condition	Percentage	Example
No access by any party	396 (38.7%)	Don't proxy, request or collect Facebook usernames or passwords.
User consent	249 (24.34%)	Obtain consent from people before using user data in any ad.
No third-party access	206 (20.13%)	Don't use the Ads API if you're an ad network or data broker.
Comply with regulations	123 (12.02%)	Any End User Customer Data collected through your use of the Service is subject to the GDPR
Other	47 (4.59%)	Only certain application types may access Restricted data for each product.

Table 6: Most common patterns of reflection call sequences

Sequence pattern
Class.forName() → getMethod() → invoke()
getDeclaredMethod() → setAccessible → invoke()

syndication libraries also acquire restricted data from these third-party SDKs but rarely send them out to their servers. Therefore, a policy violation can only be confirmed once the collected data are delivered to the unauthorized recipient.

3.4 Meta-DB Construction

Our *Meta-DB* records the API specifications and metadata of top 40 third-party libraries, which cover 91% of Google Play apps (see below). For each API, *Meta-DB* records the data it returns (e.g., session token, page likes, user ID, profiles, groups followed) and whether or not the return data is restricted by the SDK's ToS.

Identifying popular third-party SDKs. To find the most popular SDKs which are appealing XLDH-attack targets, we ranked the third-party SDKs based on the number of apps using them. Specifically, we randomly sampled 200,000 apps in D_g and identified the third-party SDKs using by those apps. Just like MAPS [87], we considered a SDK as third-party if the top and second level domains in its package name do not match the app's package name. After ranking those SDKs, we selected the top 200 excluding those with obfuscated package names, and further manually reviewed and removed utility SDKs which are not associated with restricted data, e.g., Google gson SDK [21]. The remaining 40 SDKs were then used in our research to construct *Meta-DB* (Figure 3).

Note that in our study, the 40 SDKs (from top 200) recorded in *Meta-DB* are integrated in 91% of apps. This indicates a high chance for them to co-locate with a malicious library in an app. In contrast, the remaining 6,273 SDKs we found were less popular: the 201st popular SDK was integrated in just 0.8% of Google Play apps.

Identifying privacy-sensitive APIs. We gathered 26,707 API specifications provided by the aforementioned 40 SDK vendors. Such documentations, especially those provided by popular vendors, tend to be highly structured, with well specified API names, argument lists, and return data. This allowed us to build a parser to extract the API names and the return values. Particularly, for each API, we use regex (e.g., "`returns(\W*\w*)*|retrieves(\W*\w*)*|get(\W*\w*)*`") to match the return values. Note that API specifications are often well-structured and the regex based method is efficient to identify the return values. In particular, we evaluate the regex-based method on 200 labelled data and achieve a precision of 100% and a recall of 98.74%. Altogether, we extracted 10,336 APIs and their associated return values from 26,707 API specifications.

Our study marked an API as privacy-sensitive if its return values were protected by data sharing policies. This is done by checking each API's return values against the restricted data reported by *DPA*. However, this can not be achieved by simply using a string matching method, because the API specification and ToS usually describe protected information differently. For example, ToS tends to describe a data object in a more generic way (e.g., user profile), while the API documentation usually use more specific terms (e.g., username). Hence, we align the data objects in API specification with that in the ToS based upon their semantics (represented by the vectors computed using an embedding technique). Specifically, we train a domain-specific word embedding model to get the data object vectors, and then measure the similarity by calculating the cosine distance between the vectors. In our implementation, we gather 1.5G domain-specific corpora (e.g., privacy policies, ToSes, API documentations) and 2.5G open-domain corpora (e.g., Google News, Wikipedia) to train a skip-gram based word2vec model. Here, we leverage data augmentation technique [84], which generate a new sentence by randomly replacing synonym, inserting word, swapping positions of words and deleting words, to enlarge our domain-specific corpora.

Evaluation. To evaluate the model, we randomly sampled 300 APIs from 6,394 APIs associated with 10 SDKs' API specifications. We manually checked the API specification and labeled 153 privacy-sensitive APIs and 147 non-privacy-sensitive APIs. By setting a similarity threshold of 0.7, our approach achieved 87% precision and 93% recall on the annotated dataset. In total, our model discovered 1,094 sensitive APIs from 26,707 APIs of 40 SDKs meta-DB. We manually checked all of them and got a precision of 85.6%. Note that we only used the validated sensitive APIs in the XLDH detection.

4 Evaluation and Challenges in Detection

This section reports our evaluation study on *XFinder* to understand its effectiveness and performance, and the challenges in identifying XLDH from a large number of real-world apps.

4.1 Effectiveness

Evaluation on ground-truth set. We evaluated *XFinder* over the ground-truth dataset including a "bad set" and a "good set", with 40 apps each. The apps in the bad set are integrated with 4 XLDH libraries (*com.yandex.metrica*, *com.inmobi*, *com.appsgeyser*, *cn.sharesdk*), which were found manually early in our research (before we built *XFinder*). The good set includes the apps randomly sampled from the top paid app list on Google Play [22]. They are considered to be mostly clean and were further confirmed manually in our research to be free of XLDH libraries: we inspected cross-library calls in these apps against the top 40 SDKs (recorded in *Meta-DB*) and concluded that their corresponding data flows do not violate the callees' ToSes. Running on these ground-truth sets, *XFinder* achieved a precision of 100% and a recall of 100%.

Evaluation on unknown set. Then, we evaluated *XFinder* on a large "unknown" dataset – D_g excluding 13018 apps integrating known XLDH libraries, which contains 1,328,130 free Android app in total with 40 SDK ToSes. *XFinder* reported 2,968 apps associated with 37 distinct XLDH libraries (distinguished based on their package names). To measure the effectiveness of *XFinder*, we randomly selected three apps for each identified XLDH library (105 in total) and manually validated the detection results: 32 (out of 37) identified XLDH libraries were true positives (a precision of 86%), affecting 93 out of the 105 apps. We performed manual end-to-end tests on seven XLDH libraries (including `OneAudience`, `Mobiburn`, and `Devtodev`) in real-world apps, and confirmed that they indeed exfiltrated Facebook user data to their servers (using Xposed [40] for app instrumentation and Packet Capture [1] for inspecting networking traffic).

Looking into the five falsely reported libraries, we found that three of them (`com.parse`, `com.batch` and `com.gigabud`) were caused by the taint analysis of *XLA*. As mentioned in Section 3.3, for better scalability, our taint tracking is object-insensitive. Specifically, after our approach taints a field f (holding a Facebook token) in an object *obj* of class C, which causes the whole class to be tainted; as a result, when the taint of the field f' (not storing a sensitive data) in another object *obj2* of the same class is propagated to a sink, *XLA* could not distinguish the two objects and simply considers the token-related information to be exposed to the sink, thereby leading to the false alarms.

Another two false positives (`com.xcosoftware` and `fr.pcsoft`) were introduced because our current program analysis could not fully resolve the server endpoints of data exfiltration. Although *XFinder* found that the two libraries expose a Facebook access token to the Internet (so reporting them as XLDH), the libraries actually send the token to the Facebook server (to retrieve additional user data, e.g., name, ID, page likes), not an unauthorized recipient. Fully automated resolution of such an endpoint is challenging, since the Facebook endpoint used in the networking API is heavily obfuscated (using a complicated control flow to transform the endpoint string, see the code snippet in our released dataset [39]). We utilized one of the state-of-the-art tools [89] capable of statically resolving string values in Android apps (using a *value set analysis* approach, with backward slicing and string related operation analysis), which, however, still failed to handle a case. In our research, we also observed that certain XLDH libraries such as `com.mobiburn` fetch a dynamic exfiltration endpoint whose value cannot be resolved statically (see the evasion techniques in Section 5.3). Hence, for a better coverage, *XFinder* opts to report all exfiltration cases even if the endpoints could not be resolved, and then relies on a manual process to validate the results. Note that the percentage of such false cases is low in our results.

Discussion of potentially missed cases. Due to the lack of ground truth, determining the number of missed XLDH libraries on a large scale is challenging. In general, false negatives can be introduced for two reasons: (1) challenges in automatic data-sharing policy analysis on ToSes; (2) limitations of today's static program analysis techniques, e.g., precise taint tracking, building complete call-graphs, and resolving reflection-call targets.

Specifically, although *DPA* achieved a high precision and recall in ToSes analysis (see evaluation), it missed vague and complicated cases as mentioned in Section 3.2. This can be improved in the future by investigating efficient dependency parsing on long sentences. Second, *XFinder* shares the limitations of current static analysis techniques. In particular, false negatives could be introduced due to the limited capabilities of taint tracking in complicated real-world apps/libraries. For example, an XLDH library could store the restricted data in the host app's datastore (e.g., Android SharedPreferences, SQLite database, files [3]) and later use another thread/module to retrieve a specific date item and send it out. Such a complicated data flow could not be automatically taint-tracked by our current approach, nor could it be handled by other state-of-the-art tools like FlowDroid. We will leave the systematic study of the convoluted XLDH data practices to our future work. Furthermore, limited by the capability of DroidRA [62] to resolve targets of reflection calls, our approach may not identify all cross-library calls if the target class name and function name are stored in variables, passed from other threads, or obfuscated. Also, since we leverage FlowDroid to build the call graphs, which may not be complete, we may not find all cross-library calls based on the graphs.

4.2 Performance

Running *XFinder* on 1.3 million apps and 40 SDK ToSes, it took around two months to finish all the tasks including *DPA*, *Meta-DB construction*, and *XLA*. Among these three components, *XLA* was the most time-consuming one (around two months). To analyze all 1.3M apps (D_g), we utilized a set of computing resources available to us, including one supercomputer (shared in our organization), two servers (20 cores/251GB memory, 12 cores/62GB memory respectively), and 24 desktops (4 cores/15GB memory each). We configured a 300-second timeout for taint tracking, with 83.7% of the apps successfully analyzed without timeouts or decompilation errors (11.4% and 4.9% of them with timeouts and de-compilation errors respectively). *DPA* took 2 hours to extract 1215 (*object*, *condition*) pairs on a Mac machine with processor 2.6 GHz Quad-Core Intel Core i7 and memory of 6 GB 2133 MHz LPDDR3. *Meta-DB construction* took 4 hours to find privacy-sensitive APIs from the API documentations of the top 40 SDKs.

5 Measurement

Based on the detected XLDH libraries and affected apps, we further conducted a measurement study to understand the *XLDH* ecosystem. In this section, we first present the overview of the real-world *XLDH* ecosystem discovered in our study (Section 5.1), and then describe the scope and magnitude of this malicious activity, as well as the infection techniques and distribution channels of the XLDH libraries.

5.1 XLDH Ecosystem

Before coming to the details of our measurement findings, we first summarize the *XLDH* ecosystem. As outlined in Figure 6, an adversary, who owns an *XLDH* based data brokerage platform (e.g., `OneAudience`), releases an *XLDH* library that aims to harvest data from Facebook SDK. To this end, the adversary needs to distribute the library to a large number of real-world mobile apps, so he reaches out to app owners, especially those with popular apps embedding with Facebook SDK, to provide them monetary incentive to integrate the XLDH library (❶). The app integrated with the library and the Facebook SDK, once passing the SDK vendor's review (❷) and the app store vetting (❸), is available for downloading (❹). When innocent users install the app and log in Facebook, the XLDH library will stealthily access the Facebook token to harvest the user's Facebook data (❺) and send them out to its back-end platform (❻). Meanwhile, the app owner receives commissions from the adversaries based on the number of app installation (e.g., 0.03$ per installation for OneAudience) (❼). Finally, the brokerage platform monetizes users' Facebook data by sharing it with a marketing company (e.g, Nielsen which offers political and business marketing) (❽).

Table 7: App Categories

Categories	# of apps	Proportion
Game	5556	28%
Entertainment	1296	7%
Food and Drink	1160	6%
Books	827	4%
Business	827	4%

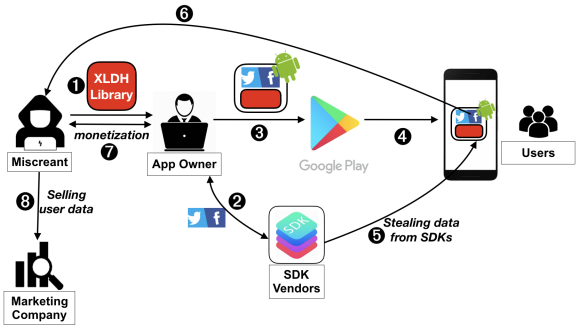


Figure 6: XLDH ecosystem

5.2 XLDH Libraries in the Wild

Prevalence of XLDH. Our study reveals that *XLDH* activities are indeed trending among the real-world apps. Altogether, we detected 42 distinct *XLDH* libraries integrated in more than 19K apps and targeting at 16 SDKs. Apps including XLDH libraries, as discovered in our research, are found in 33 categories on Google Play. As shown in Table 7, over 35% of apps are in the categories of game and entertainment. Those apps have been downloaded more than 9 billion times in total on Google Play. Among all affected apps, some are highly popular with more than 100 million downloads (Table 8).

Table 8 illustrates the top-10 XLDH libraries based on the number of apps integrating them and the data harvested (we release the full list of XLDH libraries online [39]). We found that a few XLDH libraries dominate the XLDH ecosystem. Particularly, *com.yandex.metrica* is the most popular XLDH library and appears in 40% of the affected apps. *com.yandex.metrica* is the SDK provided by Yandex, a Russian Internet corporation, for traffic analytics service. *com.yandex.metrica* leverages the reflection technique to fetch Google advertising ID and Android device ID from the Google play service SDK. Associating Google advertising ID with Android device ID is privacy sensitive since it can be used to identify a specific Android user. However, *com.yandex.metrica* does not declare this activity in its privacy policy, which violates the ToS of the Google play service SDK [13]. Similar behavior is also found in *com.appsgeyser*, which has affected more than 4 thousand apps with 15 million downloads.

Historical versions of XLDH libraries. To understand the evolution of XLDH libraries' behavior, we collected their old

Table 8: Top-10 XLDH libraries (integrated in the most apps)

XLDH library	# of apps/downloads	Harvested data
com.yandex.metrica	8,014/2B+	Google Advertising ID, Android ID
com.inmobi	4,283/4B+	Google Activity Recognition
com.appsgeyser	4,202/15M+	Google Advertising ID, Android ID, IMEI, Mac Address
com.oneaudience	1,738/100M+	Facebook ID/name/gender/email/link, Twitter user data
cn.sharesdk	815/191M+	Bytedance ID/name
com.umeng.socialize	495/175M+	Facebook/Twitter/Dropbox/Kakao/Yixin/Wechat/QQ/Sina/Alipay/Laiwang/Vk/Line/Linkedin's AccessToken and user data (ID/name/link/photo)
com.revmob	340 /36M+	Facebook AccessToken
ru.mail	299/100M+	Google Advertising ID, Mac Address, Android ID, IMEI
com.ad4screen	245/183M+	Facebook appid, AccessToken
com.devtodev	231/318M+	Facebook user gender, birtbday

versions, and then investigated the change of their malicious functionality. Specifically, we gathered historical versions of XLDH libraries from the library websites, Maven repositories [27] and GitHub [20]. In this way, we found 495 versions from October 31, 2011 to February 12, 2020 for all the 42 XLDH libraries. After that, for each XLDH library, we monitored the code change related to malicious cross-library data harvesting by checking its fingerprints (e.g., class names of the reflections, $forName$ and $getMethod$) across different library versions.

We selected 7 XLDH libraries (*com.ad4screen*, *com.onradar*, *ru.wapstart*, *io.radar*, *com.devtodev*, *com.yandex.metrica* and *com.inmobi*), which had at least three versions recorded in our dataset, to look at their trend individually. Figure 7 illustrates the maliciousness of XLDH libraries across different versions. We observe that libraries tend to have XLDH code in their newer versions. Among them, *com.yandex.metrica* and *ru.wapstart* began to release the XLDH versions since late 2014. Interestingly, we found that after Oct. 2019, the versions of *io.radar* and *com.devtodev* removed the XLDH function to steal users' Facebook data. We believe that this is at least in part thanks to our report to Facebook, Twitter, and Google Play (in October, 2019), which then warned the vendors of the offending libraries. Also, in 2018, *com.inmobi* released the version without XLDH to stop collecting Google activity recognition data. This could result from the attempt to comply with GDPR.

To understand the presence of XLDH library in the wild, we performed a longitude study of the Google Play apps with XLDH libraries *com.oneaudience*, *com.devtodev* and *io.radar* from January, 2015 to December, 2019. All of these libraries stealthily access app users' Facebook data (e.g., gender, birthday, visited place, etc.). Specifically, we started from

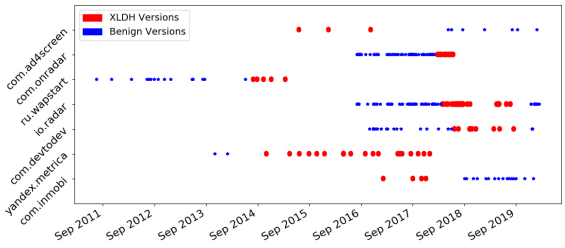


Figure 7: Distribution of XLDH versions

the 2,076 apps (out of the 1.3 million set), which are found to have integrated the above libraries, and fetched the historical versions of these apps on [41] to evaluate the presence of XLDH libraries in each version. In this way, we were able to collect 936 apps with 5,732 versions. Among them, 1,976 of the versions were affected.

Figure 8 illustrates the evolution of the number for the newly-appearing apps with the XLDH libraries, compared with the number of the apps with the libraries removed over time. We can observe that a large number of the affected apps came into sight from July, 2017 to December, 2019, and the growth started to slowdown in 2019 (in part thanks to our report to Facebook, Google Play, etc., in October, 2019). Interestingly, for each library, the trend of disappeared apps is almost identical to that of newly appearing apps with the delay of about half a year, e.g., 165 new *com.oneaudience* apps were published during the second half of 2018, while 166 apps disappeared half a year later. Comparing these two app sets, we found that there were actually 159 overlapped apps generated by an app builder *appsgeyser.com* – these apps share similar package names, which we released online [39]. This indicates that the adversaries were leveraging *appsgeyser.com* to quickly release and then remove a bunch of *com.oneaudience* apps periodically, which is likely to be a strategy for gathering data from different users. as further evidence, we observed that since March 2017, AppsGeyser in its privacy policy acknowledged that it would include OneAudience in app generation [7]. Similarly, the technique is adopted by a game app provider, *duksel.com*, to integrate *com.devtodev*.

5.3 Dissecting Infection Operations

Most targeted SDKs. Figure 9 shows the victim SDKs and the number of XLDH libraries (top 20 XLDH libraries based on the number of apps integrating them) that attack them. Google ads service is the most commonly affected SDK, followed by Facebook login and Twitter login. Among victim SDKs, 7 of them are OSNs, 2 are advertising and tracking platforms, 6 are instant messaging service, 1 is cloud service.

Given the list of victim SDKs, we observe around half of them are in the category of online social network (OSN). It suggests that high-profile OSN platforms present an invalu-

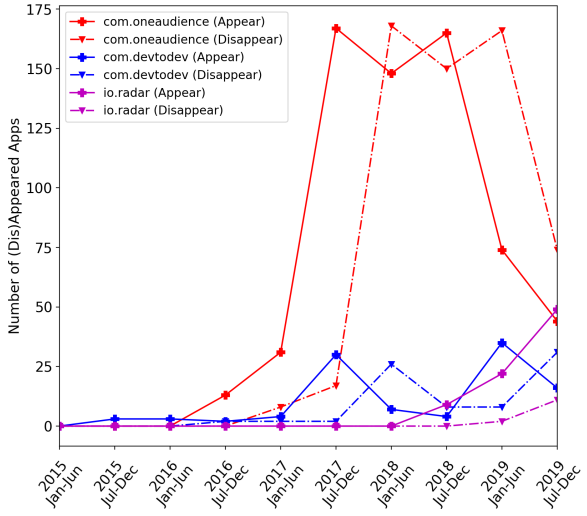


Figure 8: Number of newly appearing and disappeared XLDH libraries

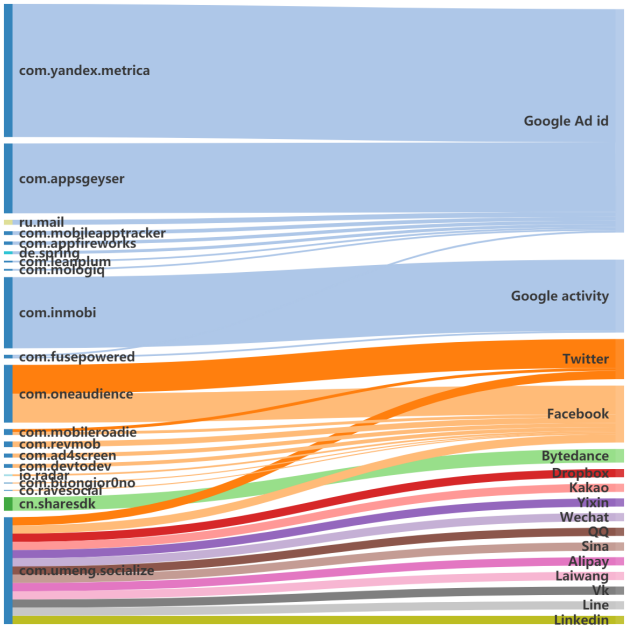


Figure 9: XLDH flows, where XLDH libraries (left) fetch data from victim SDKs (right)

able source of private user data to the XLDH adversaries. In our study, we observe the adversaries stealthily collect Facebook token, WeChat token and LinkedIn token, which can be used to access OSN-specific semantically-rich data. For example, we observe *com.oneaudience* and *com.mobiburn* stealthily access Facebook token and further leverage that token to fetch semantically-rich Facebook data, including Facebook ID, Gender, Email, Pages likes, Followed groups, etc. In particular, Facebook data, such as the pages likes, social, political, health groups the users followed, timeline, etc. can be used to create the users' psychographic profiles, as suggested by recent Facebook political scandal [85]; similarly, the data of Twitter, LinkedIn, Amazon, including a user's tweets, page likes, education background, celebrities she followed, purchase history, etc., can also be exploited by the adversaries to profile the user's personality, values, opinions, attitudes, interests, and lifestyles. The risk is especially serious since we observed that data harvested through *com.oneaudience* indeed have been shared with a political and business advertising company Nielsen.

Evasiveness of XLDH libraries. In our study, all affected apps were published on Google Play, which means the XLDH libraries have bypassed the app vetting of the Google Play and the victim SDK vendors (e.g., Facebook and Twitter, if app review process exists). The result suggests that XLDH is a new type of threat not well understood before. To further assess whether existing techniques can detect XLDH libraries, we leveraged VirusTotal [38], which aggregates more than 70 antivirus products, to scan all the XLDH libraries we found. Interestingly, no single product in the VirusTotal can detect any XLDH libraries we found; in contrast, VirusTotal was known to be capable of detecting harmful libraries studied in prior works [50, 60, 64, 75]. More than half of XLDH libraries we found abused Java reflection to call the target SDK (see Section 2), which leverages a generic Java function, to implicitly call the target SDK API. Further, unlike regular code in Java that needs to import the target class (e.g., `import com.facebook.AccessToken` [10]), the malicious library does not have to explicitly import the target class. This makes the cross-library intention more evasive. Interestingly, in our study, we also observe that XLDH libraries strategically hide the data exfiltration channel to evade detection, as elaborated below.

Hidden data exfiltration channels. We manually analyzed the decompiled code of XLDH libraries and observed some other techniques they use, with which the data-harvesting behaviors are harder to analyze:

• *Dynamic exfiltration endpoints*. In some XLDH libraries, we observed the adversaries remotely control the data exfiltration process, where the endpoints to receive the data and the time to exfiltrate data are configured to be dynamic. More specifically, at runtime, the XLDH library (e.g., *com.mobiburn*) fetches a remote configuration file, which specifies the endpoint (a URL) that data should be exfiltrated to, and when to do so. Such a treatment helps XLDH library evade the domain blocklist based blocking, once its server endpoint is on the blocklist and thereby blocked by local firewalls, ISP, etc.; not hard-coding the malicious endpoint can also render static vetting less effective. Furthermore, a dynamic, controlled exfiltration schedule gives the adversaries better control to evade detection, e.g., no exfiltration during vetting.

• *Hidding data exfiltration in crash reports*. Another in-

teresting observation is that XLDH libraries put the exfiltrated data in crash reports. For instance, the XLDH library *com.kongregate* appends the harvested Facebook session token in its crash report, and sends it out to the Internet.

• *Data encryption*. We observed the XLDH libraries (e.g., *com.mobiburn* and *com.umeng*) encrypted the data before sending to the Internet: particularly, RSA used by *com.mobiburn* and AES used by *com.umeng* for data encryption. This can render network-based privacy detection less effective, e.g., [73], which used mitmproxy to decrypt HTTPS traffic for data leakage inspection.

Backend server endpoints. Our study uncovers a series of backend servers of the XLDH libraries (see Table 10 in Appendix). We used VirusTotal to scan all of the backend server URLs. However, none of them were flagged as malicious. We observe that while an XLDH library typically exfiltrates data to their own domain – which might be blocked by domain blocklists [31, 36, 38] – XLDH developers also leverage reputable domain to receive data. Specifically, the XLDH library *com.kongregate* appends the data into its crash reports, and thus, the data was sent to Google's Crashlytics, a platform that helps app developers collect crash information and analyze stability issues [12]. Similarly, *com.adience* sent their data to a free web hosting service `frwrd.me`.

5.4 XLDH Library Distribution Channels

Behind the popularity of XLDH libraries are the promotion mechanisms that XLDH vendors take to attract app vendors. To understand their promotion and distribution channels, we searched the library websites, news reports, and analyzed the code and harvested data of XLDH libraries, and summarize their distribution channels as below.

• *Pre-installed libraries.* As mentioned in Section 5.2, we observe that XLDH libraries were widely pre-installed in the apps, which were generated by free app building services (e.g., appsgeyser [6], duksel [14]). For instance, 18% of the affected apps of *com.devtodev* were generated by the duksel [14], whose apps show similar package name pattern "`com.duksel.(\W*\w*)*free`".

• *Embedding in other libraries/services.* Another important channel to distribute malicious libraries in real-world apps is to be integrated by other popular libraries or services. As mentioned earlier, the vendor of *com.appsgeyser*, AppsGeyser, is the biggest free Android app builder on the market [6] who integrates XLDH libraries in the Android apps it created. Interestingly, we found that *com.appsgeyser* appears 34% and 29% of the time with *com.onaudience* and *com.yandex*, respectively. Hence, the users of apps generated by AppsGeyser face the risks of exposing Facebook and Twitter user data and Google Advertising IDs and device IDs.

As another example, we found the XLDH library *com.mobiburn* included another XLDH library *com.onaudience*. So any app integrates the former library could silently include the latter, increasing the chance of XLDH-library distributions. This was observed in 8 older versions of *com.mobiburn*, released from April 11, 2018 to June 17 2019 (v1.5.3 to v1.9.0). This might come out of the collaboration relation between the XLDH developers.

• *Offering app monetization*. Another important promotion channel XLDH vendors take is to offer app vendors monetary incentives to integrate the libraries. For example, `OneAudience` offers app vendors 0.03 USD per app installation; `Mobiburn` offers 0.015 USD per app installation.

Among those libraries offering app monetization, *com.oneaudience* and *com.mobiburn* do not provide any functionalities to the apps, except for data harvesting. Interestingly, *the app vendors may not fully understand the risk such a library incurs in her app*. For example, `OneAudience` claims in its privacy policy (dated in February 19, 2019, and accessed in our study in October, 2019) that it collects the user's device ID, operating system type, device make and model, etc., which are albeit private but commonly considered acceptable if properly disclosed; however, behind the scene, it collects a full spectrum of Facebook and Twitter data via unauthorizedly using AccessToken.

• *Offering appealing functionalities*. Some libraries (*com.sharesdk* and *com.umeng.socialize*) offer app developers helpful functionalities, although behind the scene performing XLDH. The functionalities they offer include integration with social media (e.g., single-sign on, posting to Facebook/Twitter), analytics, crashreports, pushing messages, in-app purchases, etc.

6 Discussion

Impacts to privacy regulations. Our study complements the recent understandings to privacy compliance: a thread of recent works [87][44] assessed whether an app's data practice (e.g., data collection and sharing with third-party) is consistent with what is disclosed in its privacy policy (a.k.a., *flow-to-policy consistency analysis*). These studies generally suggested that app vendors are at fault by not properly disclosing the data sharing with third-parties (e.g., advertisers, analytics providers, etc.), and correspondingly, app vendors have been charged by regulatory agencies (e.g., FTC) [24]. To complement these studies, our study shows that app vendors can also be victims, since an in-app data flow to a malicious third-party by *XLDH* can be opaque to app vendors. In this regard, our work has serious implications to privacy compliance regulations; also, *XFinder* can be used by both regulators and app vendors to inspect in-app data practice with third-parties.

Responsible disclosure. We have reported the affected apps and XLDH libraries we found to the app vendors and app store (e.g., Facebook, Twitter, Google Play store) and helped them understand the threats since October, 2019. Google has

removed the affected apps from the Google Play or asked the app owners to remove those libraries. Facebook and Twitter have taken legal actions against the XLDH library providers.

Future work. As discussed in Section 4, an automatic and sound detection of XLDH libraries in the wild is limited, at least in part, by today's techniques for document analysis (ToS, privacy polices) and program analysis (e.g., tracking complicated data flows, resolving reflection-call targets). Note that, as we observed, reflection-based calls are widely used in XLDH activities, likely because they are more stealthy and difficult to detect than conventional calls (developers may also use reflection calls with less/none malicious motivations, such as maintaining backward compatibility when the target class may not exist [62]). Hence, better capability to resolve reflection calls in the future may contribute to a more sound detection of XLDH and more in-depth understanding of the XLDH activities. Further, although we focused on Android in this study, XLDH is completely feasible on iOS, which we plan to study in the future work. We indeed made a preliminary attempt by looking at the iOS version of an XLDH library `OneAudience`, and found it has the same XLDH behavior as on Android (harvesting data from Facebook and Twitter SDKs). We communicated XLDH risks to Apple in late 2019, who worked with us to analyze the iOS counterparts of XLDH libraries we found on Android and asked affected apps to remove the malicious libraries.

7 Related Work

Study on malicious mobile SDKs. Prior studies such as extensively explored the risks of malicious mobile SDKs. In particular, prior research showed that malicious SDKs could collect users' sensitive data from the host apps and mobile devices, leading to serious privacy leakage due to their wide integration/adoption by popular mobile apps [53, 72, 78, 79, 81]. The problem has been studied through large-scale measurements using both static [69] and dynamic [52, 72, 73] program analysis. The sensitive data studied include on-device data (e.g., IMEI, phone number, GPS coordinates), as well as user profiles (e.g., age, gender, preferences) from app server [54, 69]. To mitigate the problems, prior research proposed different fine-grained mechanisms to isolate third-party SDKs [59, 76, 82, 83]. Unfortunately, these mitigates are hard to be fully adopted by current ecosystem due to different deployment limitations (e.g., requiring app code instrumentation [76, 82] or human-crafted policies [83]). Recent studies also studied malicious SDKs involved in the ad-fraud scheme, using techniques like click injection and click flooding [65]. Different from prior research, our study sheds lights on a new type of privacy harvesting channel (i.e., the cross-library data harvesting) which is significantly different from prior studies in terms of the diversity of the private data and complexity of in determining their data sharing policies (specific to individual SDKs). In addition, our measurement covers 1.3M apps for a comprehensive understanding of the problem in the wild, the largest scale compared to all previous research for privacy leakage study.

Text analysis for mobile privacy. Our approach for identifying privacy in-compliance between cross-library invocations through their documentations (i.e, Terms-of-service) follows a history of proposed text analysis over mobile apps, using a mixed technique of NLP and machine learning. Topics within this range include identifying sensitive data items [68, 69], their desired destinations [87], policy contradictions [45] as well as the the benign usage contexts [44, 51]. In terms of mobile privacy compliance, Whyper [70] is among the first to use NLP techniques for automatically reasoning the permission usage in mobile apps through text analysis from app descriptions. Later, a thread of recent works [44, 57, 80, 87, 88] provide better understanding for the privacy policy and its compliance with mobile app's data practice. Specifically, Polisis [57] proposes neural-network based classifiers to automatically annotating privacy policies with both high-level and fine-grained labels. Maps [87] performs large-scale measurement analysis to identify those privacy leakage of mobile apps which are not disclosed by their privacy policies. PolicyLint [45] investigates the internal contradictions of a given privacy policy, by identifying and analyzing the data collection and sharing statements at the sentence-level. However, these works are more focused on the privacy implications caused by app developers. Instead, our research look into the privacy compliance among different parties. This recalls a more in-depth analysis over a border range of privacy-related documents (i.e, Terms-of-service of third-party libraries). The sensitive data considered in our research among cross-library invocations rely on parsing the ToS statement, rather than a pre-defined list, as did in previous studies [69, 87]. In addition, identify such privacy violations requires more fine-grained rules (Section 3.2), which is not addressed by previous research.

8 Conclusion

In our paper, we report the first systematic research on XLDH libraries aiming at third-party SDKs to harvest private user data, based upon a suite of techniques that addresses the challenges in analyzing SDK ToSes to recover the semantics of data sharing policies and evaluating apps to find cross-library interactions. Our study demonstrates the significant privacy and social impacts of this new threat. Our research further uncovers a series of unique characteristics of the XLDH libraries, such as their distribution channels, hidden data exfiltration channels. We discussed the limitations of our current tool and the future research that is needed for a more in-depth understanding of XLDH activities.

Acknowledgement

Jice Wang, Jinwei Dong and Yuqing Zhang are supported by National Natural Science Foundation of China U1836210 and CSC scholarship. The authors of Indiana University are supported in part by Indiana University FRSP-SF and NSF CNS-1618493, 1801432 and 1838083.

References

[1] Packet Capture. https://play.google.com/store/apps/details?id=app.greyshirts.sslcapture.

[2] Aliyun - resolve conflicted utdids. https://help.aliyun.com/knowledge_detail/59152.html.

[3] Android app storage. https://developer.android.com/training/data-storage.

[4] Android application sandbox. https://source.android.com/security/app-sandbox.

[5] Android social library statistics and market share. https://www.appbrain.com/stats/libraries/social-libs.

[6] Appsgeyser. https://appsgeyser.com/.

[7] Appsgeyser's privacy policy. http://appsgeyser.com/privacy/app/.

[8] Bag-of-words. https://en.wikipedia.org/wiki/Bag-of-words_model.

[9] Best practise of advertising id. https://developer.android.com/training/articles/user-data-ids.

[10] com.facebook.accesstoken - facebook. https://developers.facebook.com/docs/reference/android/current/class/AccessToken/?locale=en_US.

[11] Constituency parsing. https://web.stanford.edu/~jurafsky/slp3/13.pdf.

[12] Crashlytics - google. https://en.wikipedia.org/wiki/Crashlytics.

[13] Developer policy center- google. https://play.google.com/intl/en_us/about/developer-content-policy/.

[14] Duksel, we create amazing games. https://duksel.com/.

[15] Facebook app review – app development. https://developers.facebook.com/docs/apps/review/.

[16] Facebook terms of service. https://www.facebook.com/terms.php.

[17] Facebook–cambridge analytica data scandal - wikipedia. https://en.wikipedia.org/wiki/Facebook%E2%80%93Cambridge_Analytica_data_scandal.

[18] Full profile fields - linkedin. https://docs.microsoft.com/zh-cn/linkedin/shared/references/v2/profile/full-profile?context=linkedin/consumer/context.

[19] GDPR. https://gdpr-info.eu/.

[20] Github. https://github.com.

[21] Google gson sdk. https://github.com/google/gson.

[22] Google play top apps. https://play.google.com/store/apps/top.

[23] googleplay-api. https://github.com/NoMore201/googleplay-api.

[24] In the Matter of Goldenshores Technologies, LLC, and Erik M. Geidl. https://www.ftc.gov/enforcement/casesproceedings/132-3087/goldenshores-technologies-llcerik-m-geidl-matter.

[25] Inside-outside-beginning. https://en.wikipedia.org/wiki/Inside--outside--beginning_(tagging).

[26] Java reflection- oracle. https://www.oracle.com/technical-resources/articles/java/javareflection.html.

[27] Maven repository. https://mvnrepository.com.

[28] merge-phrase. https://spacy.io/api/pipeline-functions.

[29] Nielsen and bridge marketing. https://www.nielsen.com/us/en/press-releases/2016/nielsen-and-bridge-marketing-collaborate-to-bring-exclusive-oneaudience-data-to-nielsen-marketing-cloud/.

[30] Part-of-speech. http://web.stanford.edu/class/cs124/lec/postagging.pdf.

[31] phishtank. https://phishtank.org/user.php?username=cleanmx.

[32] pos-attributes. https://spacy.io/api/dependencyparser.

[33] Reverse domain name notation - java. https://en.wikipedia.org/wiki/Reverse_domain_name_notation.

[34] Single sign-on (sso) - wikipedia. https://en.wikipedia.org/wiki/Single_sign-on.

[35] Skip-gram. https://en.wikipedia.org/wiki/N-gram#Skip-gram.

[36] spamhaus. https://www.spamhaus.org/lookup/.

[37] standford-doc. https://nlp.stanford.edu/software/crf-faq.html.

[38] virustotal. https://www.virustotal.com/en/user/cleanmx/.

[39] XLDH Project Dataset. https://sites.google.com/view/roommatetheft/.

[40] Xposed. https://repo.xposed.info/.

[41] Apkpure, download apk free online. https://apkpure.com/, 2020.

[42] Open Handset Alliance. Android open source project. https://source.android.com/.

[43] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[44] Benjamin Andow, Samin Yaseer Mahmud, et al. Actions speak louder than words: Entity-sensitive privacy policy and data flow analysis with policheck.

[45] Benjamin Andow, Samin Yaseer Mahmud, et al. Policylint: investigating internal privacy policy contradictions on google play. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 585–602, 2019.

[46] Benjamin Andow, Samin Yaseer Mahmud, et al. Actions speak louder than words: Entity-sensitive privacy policy and data flow analysis with policheck. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 985–1002, 2020.

[47] Steven Arzt, Rasthofer, et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps.

[48] CalOPPA. California online privacy protection act (caloppa). https://leginfo.legislature.ca.gov, 2020.

[49] Danqi Chen and Christopher D Manning. A fast and accurate dependency parser using neural networks.

[50] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, et al. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 357–376. IEEE, 2016.

[51] Yi Chen, Mingming Zha, et al. Demystifying hidden privacy settings in mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 570–586. IEEE, 2019.

[52] Jonathan Crussell, Ryan Stevens, et al. Madfraud: Investigating ad fraud in android applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 123–134, 2014.

[53] Soteris Demetriou, Whitney Merrill, et al. Free for all! assessing user data exposure to advertising libraries on android. In *NDSS*, 2016.

[54] Mojtaba Eskandari, Bruno Kessler, et al. Analyzing remote server locations for personal data transfers in mobile apps. *Proceedings on Privacy Enhancing Technologies*, 2017(1):118–131, 2017.

[55] Facebook. Facebook platform policy. https://developers.facebook.com/policy?locale=en_US.

[56] Matt Gardner, Joel Grus, Mark Neumann, et al. Allennlp: A deep semantic natural language processing platform. 2017.

[57] Hamza Harkous, Kassem Fawaz, et al. Polisis: Automated analysis and presentation of privacy policies using deep learning. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 531–548, 2018.

[58] Matthew Honnibal and Ines Montani. spacy 2: Natural language understanding with bloom embeddings.

[59] Jie Huang, Oliver Schranz, et al. The art of app compartmentalization: Compiler-based library privilege separation on stock android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1037–1049, 2017.

[60] Bum Jun Kwon, Mondal, et al. The dropper effect: Insights into malware distribution with downloader graph analytics. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1118–1129, 2015.

[61] Guillaume Lample, Miguel Ballesteros, et al. Neural architectures for named entity recognition.

[62] Li Li, Tegawendé F Bissyandé, et al. Droidra: Taming reflection to support whole-program analysis of android apps. 2016.

[63] Xiaojing Liao, Kan Yuan, et al. Acing the ioc game: Toward automatic discovery and analysis of open-source cyber threat intelligence. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 755–766, 2016.

[64] Xiaojing Liao, Kan Yuan, XiaoFeng Wang, et al. Seeking nonsense, looking for trouble: Efficient promotional-infection detection through semantic inconsistency search. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 707–723. IEEE, 2016.

[65] Bin Liu, Suman Nath, et al. {DECAF}: Detecting and characterizing ad fraud in mobile apps. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 57–70, 2014.

[66] Christopher D Manning, Mihai Surdeanu, et al. The stanford corenlp natural language processing toolkit.

[67] Tomas Mikolov, Ilya Sutskever, Kai Chen, et al. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[68] Yuhong Nan, Min Yang, Zhemin Yang, et al. Uipicker: User-input privacy identification in mobile applications. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 993–1008, 2015.

[69] Yuhong Nan, Zhemin Yang, et al. Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps. In *NDSS*, 2018.

[70] Rahul Pandita, Xusheng Xiao, et al. {WHYPER}: Towards automating risk assessment of mobile applications. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 527–542, 2013.

[71] Peng Qi, Yuhao Zhang, Yuhui Zhang, et al. Stanza: A python natural language processing toolkit for many human languages.

[72] Abbas Razaghpanah, Rishab Nithyanand, et al. Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem. 2018.

[73] Jingjing Ren, Martina Lindorfer, et al. Bug fixes, improvements, ... and privacy leaks: A longitudinal study of pii leaks across android app versions. 2018.

[74] Jingjing Ren, Ashwin Rao, et al. Recon: Revealing and controlling pii leaks in mobile network traffic. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 361–374, 2016.

[75] Sankardas Roy, DeLoach, et al. Experimental study with real-world data for android app security analysis using machine learning. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 81–90, 2015.

[76] Jaebaek Seo, Daehyeok Kim, et al. Flexdroid: Enforcing in-app privilege separation in android. In *NDSS*, 2016.

[77] Rocky Slavin, Xiaoyin Wang, et al. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 25–36. ACM, 2016.

[78] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In *NDSS*, 2016.

[79] Ryan Stevens, Clint Gibler, et al. Investigating user privacy in android ad libraries. In *Workshop on Mobile Security Technologies (MoST)*, volume 10. Citeseer, 2012.

[80] Peter Story, Sebastian Zimmeck, Ravichander, et al. Natural language processing for mobile app privacy compliance. In *AAAI Spring Symposium on Privacy-Enhancing Artificial Intelligence and Language Technologies*, 2019.

[81] Kurt Thomas, Elie Bursztein, et al. Ad injection at scale: Assessing deceptive advertisement modifications. In *2015 IEEE Symposium on Security and Privacy*, pages 151–167. IEEE, 2015.

[82] Eran Tromer and Roei Schuster. Droiddisintegrator: Intra-application information flow control in android apps. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 401–412, 2016.

[83] Nikos Vasilakis, Ben Karel, et al. Breakapp: Automated, flexible application compartmentalization. In *NDSS*, 2018.

[84] Jason W Wei and Kai Zou. Eda: Easy data augmentation techniques for boosting performance on text classification tasks. *arXiv preprint arXiv:1901.11196*, 2019.

[85] Wikipedia. Facebook–cambridge analytica data scandal. https://en.wikipedia.org/wiki/Facebook--Cambridge_Analytica_data_scandal.

[86] Shomir Wilson, Florian Schaub, et al. The creation and analysis of a website privacy policy corpus.

[87] Sebastian Zimmeck, Peter Story, et al. Maps: Scaling privacy compliance analysis to a million apps. *Proceedings on Privacy Enhancing Technologies*, 2019(3):66–86, 2019.

[88] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, et al. Automated analysis of privacy requirements for mobile apps. In *2016 AAAI Fall Symposium Series*, 2016.

[89] etc. Zuo, Chaoshun. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1296–1310. IEEE, 2019.

9 Appendix

Table 9: Examples of SubTree Patterns (with the full list released online [39])

sentence	patterns
store the confidential information without our prior written consent	((data:dobj,(condition:pobj) condition:prep)action:ROOT)
collect such information when the applicable End User has consented to such activities	((data:dobj,(other:nsubj,has:aux, (condition:pobj)condition:prep) condition:advcl)action:ROOT)
obtain consent before you use our service data	((condition:dobj,(data:dobj) action:advcl)obtain:ROOT)

Table 10: Examples of Exfiltration Endpoints (with full list released online [39])

XLDH Library	Endpoint
com.oneaudience	https://api.oneaudience.com/api/devices
com.revmob	https://android.revmob.com
io.radar	https://api.radar.io/
com.adience	http://frwrd.me/sdkserver
com.buongiorno	https://auth-api.newton.pm/