Towards Privacy-Preserving Social-Media SDKs on Android

Haoran Lu, Yichen Liu, Xiaojing Liao, and Luyi Xing,
Indiana University Bloomington

https://www.usenix.org/conference/usenixsecurity24/presentation/lu-haoran

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

Towards Privacy-Preserving Social-Media SDKs on Android

Haoran Lu, Yichen Liu, Xiaojing Liao, Luyi Xing
Indiana University Bloomington
{haorlu, liuyic, xliao, luyixing}@iu.edu

Abstract

Integration of third-party SDKs are essential in the development of mobile apps. However, the rise of in-app privacy threat against mobile SDKs — called cross-library data harvesting (XLDH), targets social media/platform SDKs (called *social SDKs*) that handles rich user data. Given the widespread integration of *social SDKs* in mobile apps, XLDH presents a significant privacy risk, as well as raising pressing concerns regarding legal compliance for app developers, social media/platform stakeholders, and policymakers. The emerging XLDH threat, coupled with the increasing demand for privacy and compliance in line with societal expectations, introduces unique challenges that cannot be addressed by existing protection methods against privacy threats or malicious code on mobile platforms. In response to the XLDH threats, in our study, we generalize and define the concept of privacy-preserving *social SDKs* and their in-app usage, characterize fundamental challenges for combating the XLDH threat and ensuring privacy in design and utilization of *social SDKs*. We introduce a practical, clean-slate design and end-to-end systems, called *PESP*, to facilitate privacy-preserving *social SDKs*. Our thorough evaluation demonstrates its satisfactory effectiveness, performance overhead and practicability for widespread adoption.

1 Introduction

The integration of third-party software development kits (SDKs), also known as libraries, third-party libraries or TPLs, has been now essential in the mobile app development lifecycle, enhancing app functionalities (e.g., login through Facebook/Google/Twitter single-sign on, advertising, app monetization, and analytics). Recent research [64] and news reports [2, 6, 39, 52] have highlighted a concerning threat where in-app SDKs have become victims of serious privacy breaches. These incidents involve malicious data-harvesting libraries within mobile apps that extensively steal user data from other SDKs present in the same app (e.g., by calling the latter's functions and obtaining the return data, or by directly accessing their data storage, see § 2). This phenomenon, called cross library data harvesting (*XLDH*) attacks, represents a significant threat to user privacy. Prior work [64] have identified 42 *XLDH* libraries across more than 19,000 Google Play apps, cumulatively downloaded nine billion times. The type of data harvested is highly sensitive and diverse, including user ID, credentials such as Facebook access tokens, birthday, photos, genders, page likes, and friend lists. A notable example is *OneAudience*, a library integrated in 1,738 apps with over 100 million downloads, which was discovered collecting users' private data from Facebook and Twitter SDKs. In response, Facebook and Twitter initiated legal actions to take down `OneAudience` (see the lawsuit [36, 47] and media report [39, 52])

Given the rising privacy concerns of *XLDH* targeting mobile SDKs, SDKs that manage or provide user data emerge as prime targets. Among these, social-media SDKs (*social SDK*), associated with social networks and platforms, stand out, compared to other common SDKs like ad networks and analytics, which primarily collect rather than offer data. Notably, *social SDKs* are exceedingly prevalent in mobile applications; for instance, Facebook and Twitter SDKs are found in 17.82% and 1.47% of Google Play apps, respectively. This widespread integration makes them particularly appealing to *XLDH* attacks, underscoring the urgent need for robust privacy protection mechanisms to safeguard user data against such vulnerabilities.

In addition to endangering mobile users in the wild, *XLDH* has a pressing impact on legal compliance, causing serious concerns for both app developers and social media stakeholders. With *XLDH*, the data practices of in-app malicious libraries (collecting/exfiltrating data from *social SDK*) are generally *opaque* to the app vendor. However, laws (e.g., GDPR and CCPA) and public policies (e.g., term-of-service policy of Google Play app store and Apple app store [31]) all mandate/assume that app vendors should conspicuously disclose (using a privacy policy) all data practices (e.g., collection, sharing) occurring in the app, including those performed by

third-party libraries. The violation of privacy laws/regulations have caused billions of dollars of penalty by the Federal Trade Commission (FTC) [42] against the app vendors, often leaving the third-party code modules — the real culprit — hardly accountable/liable. Moreover, the *XLDH* related attacks have been seriously endangering user trust on mainstream social platforms, mobile platforms, and technology supply chain (e.g., Android, iOS with their supply chain). With significance of the emerging *XLDH* problems, it is imperative to come up with new technical design to fundamentally eliminate or control in-app and cross-library data access channels. The goal is to achieve privacy-preserving mobile SDKs.

Challenges to defeat *XLDH* and protect *social SDK*s. The emerging threat of *XLDH* against *social SDK*s and the call for privacy/compliance guarantee to keep up with citizen expectation and social norms impose specific new challenges, which preclude direct application of prior techniques against privacy threats or malicious code on mobile platforms. As shown by the prior work [64], the *XLDH* attacks may be performed by *TPL*s of diverse functionality categories, such as analytics, marketing, ads, app maker [1], geofencing [4], etc. From the defenders' perspective, accurately predicting the specific type of libraries responsible for *XLDH* is difficult. To protect *social SDKs*, isolating all untrusted *TPL*s may not be practical, which can significantly break or change current programming paradigm, app design and runtime performance for how an app invokes or interacts with its many *TPL*s (see detailed discussion in § 4.1). Instead of sandboxing potentially all TPLs that are untrusted, we consider that a more practical solution is to sandbox the *social SDK* and protect their data. Notably, prior privacy enhancing and library-isolation techniques [46, 48, 50, 51, 56–58, 63, 66, 71], including the ongoing efforts of Privacy Sandbox on Android (PSoA) [30] suffer from several key problems for effectively or practically defeating *XLDH* (see below).

We note that even PSoA, in its current state, has different protection goals and threat model from *XLDH*, and cannot address *XLDH*. PSoA currently focuses on and only supports advertising-related library (called ads libraries in this paper) by restricting advertising-related library into a separate process — called privacy sandbox. The sandbox environment does not inherit the host app's permissions and thus prevents the ads libraries from getting persistent identifiers from the underlying system (e.g., AdSplit [57]). At the design level, PSoA does not address *XLDH* or support *social SDK* for a few key reasons. First, PSoA cannot impose *social SDK* into a privacy sandbox while still supporting their common functionalities used by the apps. Unlike ads libraries which usually come with limited interactions with the host app (thus one can safely and easily isolate them), *social SDKs* come with sophisticated data flows and functionality-level interactions with the rest of the app, such as (1) single sign-on, e.g., login through Facebook, Google, Twitter, and Amazon, and (2) in-app posting to social networks, such as sending tweets or posting/sharing to Facebook (see a detailed survey of use cases at § 3.2). The PSoA even assumes that the app should not impact or interact with SDK behaviors such as contents or operations of the SDK UI display (to protect ads display integrity and promote anti-fraud) [13], which contradicts with designed use cases of social SDKs (see comparison at § 4.3). Second, PSoA and prior other techniques [54] come with no design-level privacy guarantee. Specifically, they could not mandate that data from *social SDK* be mediated from access by other libraries or never flows to the host-app. Once a private data flows into untrusted code space (i.e., the host app with untrusted libraries), it becomes intractable (e.g., due to hidden or asynchronous data flows [35], indirect calls [59] being hard to fully resolve through state-of-the-art static and dynamic analysis techniques [70]), being completely subject to *XLDH*. Although prior techniques [46, 48, 50, 51, 56–58, 63, 71] also attempted to completely isolate ads libraries from the rest of the app, similar to PSoA, this is not an option for *social SDK*, which requires more sophisticated interactions and data flows with the host app (see detailed comparison with related techniques in § 4.3).

Design for privacy-preserving *social SDK*s. We tackle the above challenges and take the first step to address the *XLDH* at design level under the context of *social SDK*s. We introduce the *privacy-preserving social SDK paradigm* (*PESP*), a clean-slate, privacy-preserving design for *social SDK* and their usage in mobile apps (§ 3.2), nevertheless our design is mostly general and can be extended to other categories of SDKs. Specifically, we aim to fundamentally address the data harvesting against/between libraries (*XLDH*) through clean-slate design of *social SDK* and deployable, end-to-end, open-source system implementation, while best preserve expected functionalities of *social SDK*s. With a principled approach, first, we propose and generalize three new, essential properties for the design of *social SDK*s (by their vendors such as Facebook, Twitter, Google) and their in-app operation (by regular apps that adopt them), called privacy-preserving SDK or *PPS* properties (§ 3): deterministic data collectors (DDC), controllable data collectors (CDC), and auditable data collectors (ADC). Further, although we envision the next generation of *social SDK*s to be privacy-preserving fulfilling the properties, we also ensure that our design is backward compatible by supporting current use cases expected by social media/platform providers and implemented by app developers, based on a comprehensive survey in the wild (§ 3.2).

In our design, we identify, characterize and tackle key design challenges for *privacy-preserving* SDKs and combating *XLDH*, in particular the dilemma between complete isolation (i.e., no data flows out of the *social SDK*) and the functionality-necessary data flows and interactions between the SDK and the rest of the app (with untrusted libraries). Once a data flows into untrusted/app code, we lose privacy guarantee (see § 4). To fundamentally address the problem, we base our design on rigorous isolation between *social SDK* and the app/untrusted

code (quarantining private data inside *social SDK*). Despite the isolation, we propose a novel SDK design that effectively enables all expected/current functionalities/workflows of *social SDK* in mobile apps (e.g., mobile apps can still conveniently support user/GUI interactions based on data from the *social SDK*), while fulfilling *PPS* properties for the first time.

End-to-end system implementation and evaluation. We fully implemented our design of *privacy-preserving social SDK* on the recent Android (version 13, r16), and evaluated our implementation for its effectiveness (fulfilling all *PPS* properties and defeating *XLDH*), compatibility with all current use cases of *social SDK*s, and performance overhead with satisfactory results (§ 5). In our implementation, we made no changes to the OS. We implemented a *privacy-preserving* Facebook SDK and Twitter SDK (by easily wrapping the current Facebook SDK v15.0.1 and Twitter SDK v3.3.0 into our design) and used them with two demo apps released by Facebook and Twitter respectively, fulfilling all expected/current use cases of the SDKs and the *PPS* properties.

Applicability and relation with the official Privacy Sandbox on Android. Our design and implementation did not rely on the PSoA, evidenced by our full implementation on Android version 13 (r16) without PSoA. Nevertheless, since PSoA (currently only supporting ads libraries without breaking) is requesting feedback from the community, our design is compatible with PSoA and expected to be easily pluggable to PSoA to substantially enhance its privacy assurance.

Contributions. The contributions are summarized as follows.

• *New understanding and principles for privacy preserving social-SDKs*. We characterize fundamental challenges in the system design level for combating the emerging privacy threat against mobile SDKs (*XLDH*). We generalize privacy-preserving properties for the design of and operational practice with *social SDK* in mobile apps.

• *New techniques to achieve privacy-preserving design and (in-app) usage of social SDKs*. We present a practical, novel, clean-slate design and end-to-end systems to fundamentally enhance privacy of *social SDK* and their in-app usage against the emerging *XLDH*, enabling their *privacy-preserving* properties. Our techniques are practical, implemented and evaluated, pluggable to the state of the art system (e.g., PSoA). Our techniques are expected to significantly elevate privacy assurance and legal-compliance for users and stakeholders of social media/platforms, and app developers.

2 Background

Emerging privacy attacks between in-app libraries. Figure 1 illustrates a data-harvesting library (in a mobile app), which first checks the presence of the Facebook, Twitter and Google SDKs in the same app, and, if the SDKs exist, it invokes the API functions in those SDKs to acquire the user's Facebook/Twitter/Google access token, profile, groups, favorites, etc. *Notably, cross-library data harvesting (XLDH) in mobile apps can occur through multiple attack vectors besides direct function calls between libraries (and this paper aims at fundamentally eliminating all XLDH attacks).* For example, we found that the Facebook SDK maintains app users' Facebook identifiers and other personal data in a local JSON file, which can be easily accessed by any libraries in the same app, presenting a general problem on both Android and iOS. Essentially, as mentioned earlier, it is hard for the operating system to impose complete isolation between each individual library which will seriously break current functionalities and introduce intolerable programming/performance overhead. Actually, in modern software development, libraries normally invoke functions of other libraries (see the principle of modularization for software development [49], which each library fulfills an independent functionality/task) and hence it is unnatural to completely isolate each individual library (e.g., one may no longer call another conveniently and efficiently).

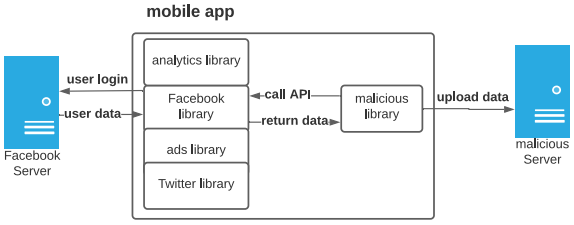


Figure 1: A Cross-Library Data Harvesting attack (XLDH)

Threat model. We consider that third-party libraries (*TPL*s) in mobile apps may intend to harvest user data from other libraries in the same app, particularly from *social SDK*, without awareness/consent of the app developers, users or the *social SDK* vendor. Although *social SDK*s may also collect user data from the app or from the underlying OS, this paper focuses on threats (*XLDH*) where *social SDK*s are the targets/victims of data harvesting. We consider that the *social SDK* vendors intend to defeat *XLDH* against them, which can be a means to improve user trust or at least mitigating possible trends of deteriorated user trust. In this regard, we consider the *social SDK* is benign. Since *XLDH* is often opaque to app developers leading to their privacy noncompliance based on current laws, we consider that app developers intend to prevent opaque data collection behaviors in their apps such as *XLDH*.

3 Goals and Definition for Privacy-Preserving Social-Media SDK

Current software engineering practices and underlying mobile systems provide low assurance for the traceability of user data, not to mention control who has access, especially in an era full of data brokers and data-harvesting sprees. To fundamentally address the data harvesting against/between libraries (*XLDH*), we propose and generalize three simple but essential properties for the design and in-app operation of *social SDK*, called *privacy-preserving SDK* (PPS) properties.

Under the context of *XLDH*, a *social SDK* that fulfills the properties is entitled *privacy-preserving* SDK.

We summarize the *PPS* properties in § 3.1. Further, although we envision the next generation of *social SDK*s and apps to be privacy-preserving fulfilling the properties, for convenient adoption and migration, we aim that a practical design should be backward compatible by supporting current use cases expected by mainstream social media and platform providers (§ 3.2).

3.1 Properties for Privacy-Preserving SDKs

PPS property 1: deterministic data collectors (DDC). Compared to *XLDH* attacks in which it is opaque or hardly known which party in an app (e.g., a library provided by a certain data-broker, ads network, analytic platform or any third-party) collects user data from the target SDK, the *DDC property* requires that any party that collects data from the SDK is deterministic.

PPS property 2: controllable data collectors (CDC). Compared to *XLDH* attacks in which the SDK provider and the app developers have little or no control regarding which party in the app is allowed to collect data from the SDK, the *CDC property* requires the SDK design to enable the app developers and SDK owners to fully control which party can collect data from the SDK. Note that based on the current laws (GDPR, CCPA, etc.) and public policies, app developers that failed to disclose data collection by third-parties in *XLDH* are liable for the privacy noncompliance, and SDK owners such as Facebook tend to be considered liable for failing to fully control user data it maintains [3].

PPS property 3: fine-grained, publicly auditable data collectors (ADC). In prior *XLDH* attacks and other privacy issues (e.g., partial, inaccurate or inconsistent privacy policies [14, 15]), key challenges remain for privacy compliance and audit: (1) the parties that actually collect user data are undisclosed or not fully disclosed to the users; (2) the disclosure granularity is too coarse-grained for effective audit (e.g., GDPR); (3) the disclosure format (i.e., usually being natural languages in privacy policies) is difficult for automatic machine audit. To fundamentally address the problems under the context of *XLDH*, the *ADC property* requires that (1) the list of parties that collect user data from the SDK is publicly auditable, i.e., being available/disclosed to normal users, app/SDK developers and policymakers, and easily readable by human and interpretable by machines, (2) the disclosure is at specific-data level with respect to specific data collectors

Consider two real examples in our study: in the app `com.fairytale.fortune`, the *XLDH* library `com.umeng.socialize` collects Twitter `User ID/E-mail` from the Twitter SDK and sends it to `http://plbslog.umeng.com`; in another app `com.africasunrise.skinseed`, the *XLDH* library `com.revmob` collects Facebook users' `AccessToken` from the Facebook SDK and sends it to `https://android.revmob.com`. In these cases, the *ADC property* requires the (app-specific) disclosure to be as fine-grained as a mapping from the data to all data collectors:

$$\begin{cases} \textit{Twitter user ID/email} \mapsto \textit{domain plbslog.umeng.com} \\ \textit{Facebook access token} \mapsto \textit{android.revmob.com} \end{cases}$$

Notably, with the *CDC property*, the above disclosure is determined by the app developers and SDK vendors and thus practical (see the design that fulfills all *PPS* properties in § 4).

Table 1: Use Case Stats

Social Media Provider	# of use cases	# number of type I workflow	# number of type II workflow
Google Play Game Services	19	12	19
Twitter Kit	13	9	13
Vkontakte	16	11	16
Kakao	19	8	19
Snap login, bitmoji kit	9	9	9
Tiktok	18	8	18
Wechat	20	17	20
Alipay	14	12	14
Weibo	20	12	20
Facebook	13	13	13

3.2 Backward Compatibility for Privacy-Preserving Social-SDK Design

As mentioned earlier, a key challenge for designing privacy-preserving SDK systems is that, prior/strict isolation techniques [30, 46, 48, 50, 51, 56–58, 63, 71] do not restrict between-library data access and easily break functionalities of *social SDK* whose major use cases feature sophisticate data flows and interactions with the rest of the app. On the other hand, allowing user data to flow out of the SDK immediately makes the data intractable — a fundamental loss of privacy guarantee. That is, to ensure the design of privacy-preserving SDK and its runtime environment fulfill the *PPS* properties without breaking current use cases of *social SDK* is a serious challenge, and we tackle this imperative problem in our study (see our design in § 4).

In this section, we report a comprehensive survey of *social SDK* use cases that are expected by 10 mainstream social platforms (e.g., Facebook, Twitter, Snap, Google Play, most popular worldwide in terms of user numbers, see Table 1), whose SDKs have been integrated into 0.01% to 17.86% of Google Play apps [7]. To comprehensively summarize their use cases, our approach includes two complementary efforts. (1) We manually inspected the developer manuals released by the social platform vendors, with a total of more than 20 MB HTML documents describing the intended usage of APIs provided by the 10 *social SDK*s (fully released online [5]). (2) We thoroughly inspected code level behaviors of 200 popular real-world apps (with a total of $100B+$ downloads [5]) that used these SDKs (20 apps for each SDK, see data collection below). Specifically, we inspected the apps at the Java and Smali code level (decompiled through the tools `Jadx` and `NinjaDroid`) and user level with end-to-end app/UI operations. This effort leverages both state-of-the-art program

analysis tool Flowdroid [32] and thorough manual efforts to inspect the code that accesses APIs of *social SDK*s and handle their data (see below).

In the manual efforts, after collecting all the 200 app samples, we have two researchers install the apps on rooted Android phones, grant all permissions asked by the apps, and manually explore and trigger all possible functionalities in the apps. We manually interacted with the app UI and pinpointed possible usages based on the service descriptions and descriptions in the app UI. During the app execution, we use Frida [41] to hook invocations of *social SDK* APIs and then manually analyze the subsequent control and data flows with respect to usage of the *social SDK*s an their data in the app. In this regard, we also adopt Flowdroid to complement manual analysis and help us more comprehensively identify control flows starting from where *social SDK* APIs are called in the apps, navigating through the code space and identify data and control flows related to how the apps use the *social SDK*s.

Based on the survey, this section below summarizes all functional use cases of *social SDK*s in the wild into two generalized types. We elaborate on our data collection, use-case survey results as follows. We released the full data set online [5] for reproducible experiments. The two researchers that perform this survey and summarize the survey results are considered co-authors of this work. This is aligned with common practices of research where intellectual contributors are acknowledged as co-authors, and their research results go through evaluation or related artifacts are released at [5] for experiment reproduction.

Data collection. We selected 10 most popular *social SDK* according to the ranking published by two app intelligence providers, i.e., AppBrain [7] and 42Matters [8]. For each SDK, we selected the 20 most popular apps based on the rankings released by the most popular app markets operating in the corresponding region of the *social SDK*s (i.e., Google Play for 7 SDKs and Huawei App Market for the other 3 SDKs from China). The full list of the 200 apps can be found in [5].

Generalization of workflow patterns for using *social SDKs* in mobile apps. Summarizing all use cases of *social SDK* expected/implemented by SDK providers and app developers, we generalize two work-flow patterns that are essential for *social SDK*s to interact with the rest of the app to fulfill expected functionalities as follows. We base our *privacy-preserving* SDK design in § 4 by fully regulating the two work-flow patterns to fulfill the above *PPS* properties, while preserving expected functionalities.

Type I workflow of *social SDK* usage in mobile apps: Once the app invokes APIs of a *social SDK*, data flow out of the *social SDK* (to the app's graphical user interface or GUI) for interactions with the user, including (1) the display of information to users, (2) letting users choose or edit the information from a *social SDK* (before sending to a remote server). High-level examples are shown in Appendix Figure 4 for more specific use cases including single-sign on (e.g., the user identifiers, names obtained from Facebook are shown in the app GUI) or posting (e.g., sharing/twitting) to a social platform or to a specific friend/group on the social platform. Before displaying the data in the app GUI, code developed by the app or third-parties may transform the data (e.g., using an image processing library to process a profile photo obtained from the social network before showing it in the app GUI). Essentially, *Type I work flow is for user interactions* on the app's user interface.

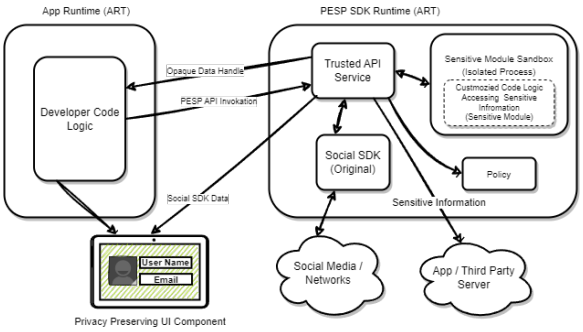


Figure 2: Design Overview of Privacy-Enhancing Social SDK

Type II workflow of *social SDK* usage in mobile apps: Once the app invokes APIs of a *social SDK*, data flow out of the *social SDK* into the app space, and eventually flow to a remote server, either the app server, the *social SDK* server, or a third-party's server (also called *Remote Host*). For example, after login with Facebook, Twitter or Google, the user's access token, email, names, etc., provided by the *social SDK* are sent to the app's sever, or sent to third-party analytics platforms and app-monetization platforms (by the app itself or third-party libraries in the app). Before sending the data out of the app, code developed by the app or third-parties may transform the data into their needed format (e.g., wrapping into a JSON or customized string). Essentially, *Type II work flow is for data sharing with a remote server* (the apps' or third-parties' servers).

4 Design for Privacy-Preserving, Social-Media SDKs

In this section, we present *privacy-preserving social SDK paradigm* (*PESP*), a clean-slate, *privacy-preserving* design for *social SDK* and their usage in mobile apps. We also provide end-to-end implementation of our design, and elaborate on how the design defeats *XLDH* and fulfills the *PPS* properties (§ 3.1) while practically enable all expected, current use cases of *social SDK* (i.e., the two generalized types of workflows in § 3.2).

4.1 Design Overview of *PESP*

As mentioned earlier, a *fundamental design challenge* for *privacy-preserving*, *social SDKs* that combat *XLDH* is the dilemma between complete isolation (i.e., no data flows out of the *social SDK*) and the necessary data flows and interactions between the SDK and the rest of the app (with third-party libraries that are untrusted). Once a data from a *social SDK* flows into untrusted code space, it is not fully tractable and we lose privacy guarantee (i.e., the data can be sent to the Internet/third-parties easily and stealthily bypassing state-of-the-art defenses including static code analysis, dynamic code analysis, traffic analysis [53]). To fundamentally address the problem, we base our design on complete isolation between *social SDKs* and the app/untrusted code (by adopting Linux UID based and process-level isolation of data, runtime, and GUI guaranteed by the operating system, see § 4.4), quarantining private user data inside a *social SDK* itself and never flowing to the untrusted/app code space. Despite complete isolation, *our key novelty includes new system design to still enable expected, sophisticated interactions between the SDK and untrusted/app code (§ 3.2), a security design goal never achieved before.*

System architecture and core components. Figure 2 outlines the core systems components under *PESP* including its isolation foundation and runtime backed by the Android OS. In an app, the *social SDK* and the rest of the app code (simply referred to as "the app", denoting all code/components of the app itself and TPLs excluding the *social SDK*) are strictly isolated (based on different Android Runtime (ART) and thus Linux UIDs, see § 4.4). Upon the app execution, the *social SDK* runs in a new Android Runtime process, called *SDK runtime*, separated from the app which runs in its original runtime (Android Runtime) provided by the Android OS.

• *App runtime*. The app code runs in the *app runtime* like a regular mobile app, except that it does not directly obtain any data from *social SDK* in the *SDK runtime*. App code in the *app runtime* invokes APIs of the *social SDK* through the exposed interface (called *trusted API service* of the *SDK runtime*, see below), and no longer directly gets the data but rather gets a data handle, called *opaque handles* with reference to the SDK data in the *SDK runtime* (here we adapt and generalize the concept of *opaque handle* [38, 44] to mobile SDKs). *PESP* ensures that necessary operations and interactions with the *social SDK* data are all performed inside the restricted *SDK runtime*, fully controlled and enforced by *SDK runtime* based on the *data-sharing policies* (see below).

• *PESP SDK runtime*. The *social SDK* runs in a separate Android Runtime process, similar to a regular service process. Essentially, the *SDK runtime* encapsulates the original *social SDK* (with zero change, see our demo and thorough evaluation of functionality and privacy on Facebook and Twitter SDKs in § 5). The invocation from the *app runtime* to the original *social SDK* functions is encapsulated as a simple IPC call to the *SDK runtime*, handled by a component called *trusted API service* (based on the Android `Service` class [23]), which relays the calls to the original *social SDK*. Once the *social SDK* returns the data, the *trusted API service* maintains the data but never returns the data to *app runtime* (to fulfill the privacy properties). The *trusted API service* instead returns a data handle (a kind of *opaque handle*) to *app runtime* or invokes related callback functions in the *app runtime*.

Despite the data quarantine in *SDK runtime*, in expected usages of *social SDKs* (§ 3.2), an app may want customized handling of the data (e.g., transformation, sharing to remote parties, display on UI for user operations). *PESP* securely enables any customized handling of the data inside the *SDK runtime*, specifically inside an internal sandbox (based on *isolated process* of Android [20]) called *sensitive module sandbox* (see Figure 2). Specifically, during app development, the app developers can develop certain code modules (e.g., some classes) or adopt specific *TPL*s, and compile them into a package (in the Android-common `.dex` format [18]) called a *Sensitive Module* in *PESP*, which is separate from the app's regular executable or package. The *Sensitive Module*s are to run inside the *sensitive module sandbox* (see examples in § 5 and example implementations online [5]). A *Sensitive Module* directly handles private data of *social SDKs* (exchanged with the *SDK runtime*), but cannot expose it. This is because *sensitive module sandbox* is based on *isolated process* of Android [20], which has least privileges such as the lack of network permissions (guaranteed by SE Android [27]) and only communicate with its creator process, the *SDK runtime* or specifically the *trusted API service*. To serve all functionality requests from the *app runtime* while preserving privacy, *trusted API service* just provides three simple but generic APIs facing the *app runtime* (detailed in § 4.2). The APIs allow the *app runtime* to specify which *Sensitive Module* to use to handle which data (see API details in § 4.2). In a *Sensitive Module*, the app developer's own code can invoke a *TPL* to help process data, and such a *TPL* is not expected to be changed due to *PESP*.

In *SDK runtime* only the *trusted API service* can decide to share specific data to a remote party upon the *app runtime* requests, based on *data-sharing policies* of the app and *social SDKs* (see *data-sharing policies* below).

• *Privacy-preserving UI paradigm*. In the Type I workflow of original usages of *social SDKs*, the SDK data was expected to flow to the app's UI for the users to view, operate or interact with: for example, during login with Facebook, the user's identifiers, names, etc. from Facebook are filled into the app's login window or account creation window (Figure 4c). To securely support app-specified, UI-based user interactions with the *social SDK* data, we introduce a privacy-preserving UI paradigm (or *PESP* UI paradigm) — note that this was not a common use case for ads libraries and thus unsupported by prior isolation techniques such as PSoA. Specifically, the app can securely designate a portion of the screen (called *s-screen*) to the *SDK runtime* and the *SDK runtime* can populate

data into *s-screen*, and allow users to operate on it without ever exposing data to the *app runtime*. How to customize the view and actions in *s-screen* is completely implemented in a *Sensitive Module* provided by the app and executed with privacy guarantee in the *SDK runtime* (inside the *sensitive module sandbox* based on *isolated process*). We elaborate on technical details in § 4.2 and end-to-end implementation and demo with Facebook and Twitter SDKs in § 5.

• *Data-Sharing Policies*. The *data-sharing policies* are specified by the app/SDK developers to fulfill the privacy properties (§ 3.1). In particular, the policy specifies for the specific app (or app vendor/domain) the specific date items that are allowed to be shared with specific data collectors (see the *DDC property* and *CDC property*). The *data-sharing policies* should/can be made publicly available under the app vendor's domain (e.g., example.com/PA_policy.json), signed by the app vendor. The *social SDK* has the authority to additionally impose a policy overriding part/all of the app vendor's policy, fulfilling its data sharing policies. See an example policy in our supporting websites [5].

With privacy by design, all parties that can collect the data from *social SDK* are deterministic, controllable and auditable, significantly elevating the privacy and compliance assurance of the *social SDK*, apps, and the underlying operating system.

Usability for app users. Our entire design is transparent to app users, who experience no changes. For example, the app user can operate on the *s-screen* populated with Facebook/Twitter SDK data, make selection, edits, and further submits the data to app/Facebook servers, e.g., during Facebook/Twitter based app login or account creation, or posting to the social networks. Figure 3 shows a screenshot in our end-to-end implementation/evaluation of a *privacy-preserving* Facebook SDK used in a demo app, fulfilling all expected/current use cases of Facebook SDK.

Discussion. Instead of sandboxing potentially all *TPL*s that are untrusted, the design of *PESP* chooses to sandbox *social SDK*s, which can be more practical while fulfilling our protection goal (defeating *XLDH* attacks). Otherwise (sandboxing all non-social libraries), all simple app-to-sdk or sdk-to-sdk function calls will go with inter-process communications, which will significantly break current programming paradigms, runtime performance and app design (a regular mobile app may utilize tens of third-party SDKs for diverse functionalities [34]). Further, based on the *PESP* design, specific *TPL*s (e.g., an image processing library) adopted by the app to process *social SDK* data are packaged and sandboxed in a *Sensitive Module*, while other non-trusted *TPL*s run in the *app runtime*.

4.2 *PESP* SDK Runtime

We elaborate on design of key components for the *SDK runtime* as follows.

Trusted API Service: interface between the app runtime and SDK runtime The *trusted API service* provides three generic APIs facing app code in the *app runtime*:

• `API 1: getDataHandle(original_SDK_API_name, arguments, optional_callback_function)`. Similar to using the original API of the *social SDK*, the app can use the original SDK API name and arguments in this `getDataHandle`, which will return to the *app runtime* `data_handles` to specific data items. Note that the *trusted API service* never returns actual data of *social SDKs* to the *app runtime*. The last argument is optional and specifies a callback function in the *app runtime* for the *trusted API service* to invoke asynchronously if the original SDK API cannot return immediately (e.g., delayed by Internet communication with the *social SDK* server).

• `API 2: sendSensitiveData(data_handles, remote_host, sensitive_module, optional_callback_function)`. API 2 is directly related to the Type II workflow (§ 3.2), where app developers intend to send data from the *social SDK* out to third-party partners or the app servers. To support this workflow, the app code from *app runtime* can use API 2 to let the *trusted API service* send specific data (referenced by the `data_handles` argument) to specified remote server endpoint (the second argument).

The third argument `sensitive_module` is optional and if specified, the *trusted API service* will launch the *Sensitive Module* (in an *isolated process*, see below) to process the data before sending them out to remote servers. Notably, the *Sensitive Module* can implement flexibly customized functionalities or operations on the data of *social SDK* (texts, images, videos, or binary data without limitation). This API return new data_handles referencing to the data after customization. As mentioned earlier, the *trusted API service* enforces the *data-sharing policies* and determines whether sharing the specific data with the domain is allowed before sending data to remote parties. Similar to API 1, the last argument is optional and specifies a callback function in the *app runtime* for the *trusted API service* to invoke asynchronously.

• `API 3: launchSensitiveModuleView(data_handles, sensitive_module)`. Recall the Type I workflow where the app may want display of the *social SDK* data in the app specific UI for the user to view or operate. Upon this API request, the *trusted API service* launches a GUI *view*, implemented by the sensitive module (the second argument) to display the data (the first argument) — a sensitive module runs as an *isolated process* considered as the *sensitive module sandbox*. In such a case, the *Sensitive Module* can be implemented as a typical Android GUI view (based on the `android.view.View` class [29], with UI elements such as text-boxes and labels defined and laid out like programming a regular Android UI [25]), including code logic that can populate provided data to the view's UI elements (Figure 2) and further take customized actions on the data.

Implemented by the *Sensitive Module*, this GUI *view* uses a portion (or entirety) of the screen (see Figure 2), referred to as *s-screen* in this design proposal. Note that the *Sensitive Module* is customized by the app developers (in contrast to *social SDK* vendors) and provided to the *SDK runtime* at app packaging time (see § 4.1). This API returns a `SurfaceControlViewHost.SurfacePackge` type handle referencing the view *s-screen* (returning to the *app runtime* that invokes this API). As detailed below, *app runtime* with the view's handle cannot access contents in the view at all (based on how the Android's System Display Service [19] works and how Android manages screens).

In the evaluation (§ 5), we demonstrate that the three simple yet versatile APIs are capable of satisfying all the use cases of both Facebook and Twitter SDKs. We showcase this through an example app provided by Facebook and a proof of concept app that we developed, incorporating the Twitter SDK. *PESP* supports multiple social-SDKs used by one app in respective runtimes. The app interacts with each SDK-runtime similarly. We evaluated and released an example app using both the Facebook and Twitter SDKs under *PESP* in § 5.4.2.

Security discussion. With *XLDH* attack surfaces significantly reduced by *PESP*, a potential remaining threat is that a malicious *TPL* adopted by the app leverages API 2 to post sensitive data to public remote hosts, and then the attackers can monitor the specific endpoints and harvest data there. Such a threat can be prevented based on *data-sharing policies* of *PESP* or existing privacy enhancing techniques (PETs). Specifically, thanks to enforcement of *data-sharing policies*, the remote hosts targeted in such an attack are considered to be either hosts of the app or the online social network (OSN) behind the *social SDK*. Targeting the former, the attack must be app-specific, which is not an *XLDH* and has significantly more limited attack surface than *XLDH*. If OSN-owned endpoints are targeted, the OSN's data-sharing policies may disallow SDK data to be posted to its public endpoints. Also, thanks to *PESP*, the data must be sent to the OSN before the attacker can harvest them, and thus prior PETs such as anomaly detection [9] can identify the attack: for example, massive data of diverse identities are sent to specific OSN endpoints or timelines, apparently being abnormal.

***PESP* Privacy-preserving UI paradigm**. To securely support app-specific, UI-based user interactions with the *social SDK* data, and enable Type I workflows (SDK data flow into the app UI), we adapt uses of Android's System Display Service (*d-Service*) and propose a novel GUI rendering technique, called *PESP-GUI*. The core idea is that *PESP-GUI* enables the trusted *SDK runtime* and the untrusted *app runtime* to each render and operate a portion of the entire screen; *SDK runtime* (i.e., the *Sensitive Module* with API 3) shows the SDK's private data only in its screen portion (called sdk-screen portion or just *s-screen*) without ever exposing SDK data to the *app runtime*. *PESP* is aligned with and leverages the design of Android's System Display Service, which manages how different views, overlays, and windows are stacked at the z-axis to form a user-facing screen. Specifically, we leverage Android *d-Service* to securely stack the *s-screen* (owned by *Sensitive Module* in the *SDK runtime*) on top of specific position of the *app runtime*'s screen (at z-axis, see Figure 3), forming a user-facing window — users see a normal app window with *social SDK* data to view and edit.

More specifically, the API 3 returns to the *app runtime* a view handle called *surfaceView* (of type `SurfaceView` [28]), referencing the *s-screen* operated by the *Sensitive Module* confined in the *SDK runtime*. The app code in the *app runtime* can then call the Android framework API `surfaceView.setChildSurfacePackage` [28] to designate that it wants its specific screen portion (up to 100%) to be stacked with and thus overwritten (on the z-axis) using the particular *surfaceView*. Note that since both the *Sensitive Module* (with view layout in the *s-screen*) and the app window are developed by the app developers, the final combined user-facing GUI will look normal for users.

Security discussion. Note that the above screen stacking technique is based on the true intention of (1) the app (by calling the above API to inform the Android *d-Service* about the target screen to use on its top and specific position), and (2) the *SDK runtime* by returning reference to the *s-screen* to the app (through API 3). Hence, this has key differences from prior overlay attacks [40] where a malicious app put its overlap or screen on top of another (victim) app — placing an overlay on other apps is now strictly restricted by Android requiring a signature level permission `System_Alert_Window` and by Google Play in app vetting [24]. Also note that, in the design of *PESP*, despite the screen stacking, the app code or the *app runtime* cannot access contents in the *s-screen* owned by the *SDK runtime*. We provide implementation details in § 4.4.

Sensitive Modules. Upon invocation of API 2 and API 3 with a *Sensitive Module* name specified, a *Sensitive Module* is launched by the *trusted API service* as an *isolated process* [20] — called a *sensitive module sandbox*. Like a regular *isolated process*, the *Sensitive Module* has a *Binder* object [17] to communicate and exchange data with its creator process (the *SDK runtime*), but is restricted for other IPC or permissions (e.g., no Internet access). The process terminates once the related function being launched returns.

Sensitive Storage. Despite the three APIs and the *trusted API service* only return data handles, and never return the actual data, the *trusted API service* keeps a private storage inside the *SDK runtime* for the specific data and its data handles in the format of {data_handle, data type, data value}. Note that, with API 3, after a *Sensitive Module* performs transformation on a set of multiple data items, it can produce new data and return them to the *trusted API service* (through IPC between an isolated process and its creator process). Then *trusted API service* can store the new data, assign new data handles and record data type as the set of the original data types. This enables the *trusted API service* to enforce the *data-sharing*

policies based on the data types when sending data to remote parties (API 2).

Generality of *SDK runtime*. Our design of *SDK runtime* is potentially general for all *social SDKs*. Our implementation (§ 4.4) is open-source can be easily used by *social SDK* vendors to wrap their current *social SDKs* without changes (see our demo and evaluation for Facebook and Twitter SDKs in § 5). § 5 further encompasses our evaluation of the practical efforts required for app developers to adopt the new privacy-preserving paradigm of *social SDKs*.

4.3 Comparison with Related Work

Comparison with PSoA. There are key design-level differences between *PESP* and Google's ongoing development of PSoA for the privacy assurance and functionality.

• *Privacy assurance*. While PSoA aims to separate SDK permissions from the app, by design, PSoA simply allows SDK-runtime to return data to untrusted app-runtime (by invoking SDK APIs) [30] (subject to *XLDH*-attacks). In sharp contrast, *PESP* confines SDK data (in SDK-runtime), thus elevating privacy assurance (against *XLDH*).

• *Enabling social-SDK functionalities*. Confining SDK data while enabling app-to-SDK functional workflows is challenging. Such a property is achieved by the PESP UI paradigm (§ 4.1), securely enabling app-specific, app-implemented operations on the SDK data inside SDK runtime. In contrast, the PSoA (with its remote-view feature [11]) does not enable such a key property: operations/implementations in PSoA remote-views are done by the SDK, and apps just determine the views' on-screen location. PSoA currently only focuses to support advertising-related SDKs [12]. The PSoA even assumes that the app should not impact or interact with SDK behaviors such as UI contents or operations of SDK-runtime (to protect ads display integrity and promote anti-fraud) [13], which contradicts with designed use cases of *social SDKs*.

Comparison with other related work. Generally, prior techniques for library isolation, restriction or protection suffer from several key problems for effectively or practically defeating *XLDH* attacks (Table 4).

• *Problem 1: Isolation targets*. Many prior techniques aimed to isolate libraries from the host app [26, 48, 50, 51, 56–58] (e.g., with separate permissions, processes, storage, or runtime). They were designed to isolate specific types of *TPL*s (e.g., ads related libraries) from the host app, and were usually not intended or designed to isolate *social SDKs* or many libraries. Using their approaches to generally isolate many libraries may significantly change app-to-library or library-to-library interactions, programming paradigms, and app designs (see discussion in § 4.1), thus being less practical. Additionally, adopting their approaches to isolate *social SDK* or address *XLDH* can incur Problem 2 or Problem 3 discussed below.

• *Problem 2: Incomplete isolation (privacy assurance)*. Although many prior techniques come with certain levels of isolation [45, 54, 55, 60, 73] (e.g., permission isolation which ensures that libraries do not directly inherit permissions of the host app), like PSoA, they usually come with no design-level guarantee that data from *social SDKs* (or generally an SDK that offers data) never flows to the untrusted, non-isolated space (e.g., app runtime), directly violating our *XLDH* security goals.

• *Problem 3: Functionality compatibility for social SDKs*. Unlike previous works' assumption [45, 46, 50, 57, 58, 71] that ads-related or other SDKs expected little or no functionality interactions with the host app, thus being more easily isolated without breaking their functionalities, *social SDKs* come with sophisticated or app-specific interactions with the app. Quarantining SDK user data (inside a *social SDK* or potentially any SDK that offers data) while enabling necessary, sophisticated interactions between the SDK and untrusted/app code is a unique contribution and advantage of *PESP*.

4.4 Implementation

Privacy-preserving UI paradigm. The type (class) of the view returned by API 3 to the *app runtime* is `SurfaceControlViewHost.SurfacePackage` [28], which serves as a reference to the *s-screen* operated by the *Sensitive Module* that includes the data of *social SDKs*. When the view of the *Sensitive Module* is created, it is stored in the screen drawing buffer of Android, shared between the Android `SurfaceFlinger` service and the *sensitive module sandbox* process (where *Sensitive Module* runs). Then, an instance of the class `SurfaceControlViewHost.SurfacePackge` is created as a reference to that buffer, and returned to the *app runtime* by API 3. Once the app code in the *app runtime* receives the reference, it can use the reference as the parameter to invoke the Android API `SurfaceView.setChildSurfacePackage` which essentially tell the `SurfaceFlinger` service to take the content of the drawing buffer owned by the *Sensitive Module* to stack on specific position of its own screen and overwrite the content of its own drawing buffer or `SurfaceView`. Note that, the overwrite happens in the trusted Android `SurfaceFlinger` process, which synthesize the drawing buffer to finalize display of the user-facing screen, and neither the *app runtime* nor the *SDK runtime* can access contents of each other's screen contents (i.e., contents in their respective `SurfaceViews`).

When the app wants to reclaim the display area taken by the *s-screen*, it can simply invoke the Android `ViewGroup.removeView` API to remove the `SurfaceView` of the *SDK runtime* from its screen. Hence, data of *social SDK* in the *SDK runtime* is never exposed to the *app runtime*, but allowing the app to control how, when and where the data is shown to the user on the screen. Note that it's possible that apps leverage the Android's `MediaProjectManager`

API to take screenshot of the whole display, but every invocation of `MediaProjectManager` for whole-display screenshot requires user consent [21].

SDK runtime. In our current implementation, the *trusted API service* is similar to a regular Android service based on the class `Service`. The *SDK runtime* wrapping the original *social SDK* (Figure 2) is packaged with its own package name. The *sensitive module sandbox* is created using the `IsolatedProcess` flag of Android `Service` which assigns the sandbox process a restrictive uid that has IPC limitations and strict SELinux policies enforced [20]. The *Sensitive Module*s are loaded from the file system using the `DexClassLoader` Android API to load all the necessary Classes in to the sandbox process.

The *Sensitive Module* in the *isolated process* can use the *Binder* to exchange information and perform IPC with the *trusted API service*. The *Binder* instance was created and accessible to the *Sensitive Module* when the *trusted API service* create the *isolated process*. Besides the three public APIs facing *app runtime* (§ 4.2), the *trusted API service* internally implemented a few more functions to serve requests (through the IPC) of the *Sensitive Module*, especially when the *Sensitive Module* wanted to make HTTPS requests to the remote servers (*sensitive module sandbox* cannot access the Internet). Once a *Sensitive Module* finishes its code execution, its isolated process terminates. Considering that multiple apps may use the same *social SDK* (e.g., the Facebook SDK), in our implementation, the system installs the *social SDK* as a unique service that can be invoked and shared by multiple apps.

App runtime. The app code runs in *app runtime*, which is exactly the same as the Android Runtime (ART)(with its predecessor as Dalvik runtime) [22] capable of running `Dex` bytecode. Since the *trusted API service* with the *SDK runtime* is implemented as a standard Android service [23], invocations to the three APIs of *trusted API service* follow the standard IPC process like invoking any service APIs (using the Android API `bindService` [16]). In our implementation, for app code to more easily use the *trusted API service* APIs, we additionally provide a convenient *client SDK wrapper* (of *trusted API service*), which is just a trivial wrapper offering the same function signatures as the three *trusted API service* APIs; the app code can import the *client SDK wrapper* as an SDK and invokes the three APIs just like invoking normal SDK functions and the *client SDK wrapper* consequently makes the IPC calls.

Isolation in *PESP*. Our rigorous isolation between the *SDK runtime* and *app runtime* and between the *SDK runtime* and *Sensitive Module* is based on mature techniques, i.e., Linux UID [26] and Android isolated process [20] respectively. In our current implementation, similar to and compatible with Google's PSoA [30] and the new SDK distribution model [10], when installing an app utilizing *social SDK*, the system installs the app and the *social SDK* as two separate packages, assigning different app identifiers and Linux UIDs. Android package name/identifier serves as a basic building block when it comes to differentiating the app and *social SDK*. Note that different *social SDKs* have different package names and identifiers, because we do not assume there exist trust between different *social SDKs* in our threat model.

Distribution. Being compatible with the distribution model of PSoA [10], a *social SDK* is packaged as a self-contained package with its transitive dependencies. As mentioned earlier (§ 4), a *Sensitive Module* with its dependencies is built into a self-contained package (.dex format), which is loaded by SDK-runtime into isolated processes.

5 Evaluation

In this section, we thoroughly evaluate effectiveness, usability, and performance overhead of *PESP*. Specifically, we use two open-source Android apps released by Facebook and Twitter that integrated the Facebook login SDK [37] and Twitter Kit SDK [61] respectively: the two apps originally implemented functionalities for major use cases related to the SDKs such as login with Facebook and Twitter. In our evaluation, we implemented our design *PESP* into the two example apps while keeping the original functionalities; this is done by wrapping the in-app Facebook and Twitter SDKs into our *SDK runtime* and migrate the app code that originally invokes the SDK functions into code that invokes the *trusted API service*. In the following, we provide three demonstration examples of the new apps (with *PESP*) that have implemented key use cases of Facebook and Twitter SDKs: login with Facebook (§ 5.1), in-app display of Facebook user profile information (§ 5.2), login with Twitter (§ 5.3). For each use case, we evaluate the effectiveness of privacy protection and usability (efforts needed for the app developers to migrate this app to the *PESP* paradigm).

In § 5.4, we report evaluation of end-user facing performance overhead under *PESP*. The evaluation is based on the above two apps (released by Facebook and Twitter) equipped with *PESP*, and we additionally evaluated multiple app scenarios with each app integrating multiple *social SDKs* under *PESP*. The results showed that *PESP* is efficient and practical. All apps used in our experiments were released with source code online [5].

5.1 Case Study and Security Analysis: Login with Facebook under *PESP*

Login with *social SDK* is a common use case that is a prerequisite step for accessing sensitive user data from the Social Network platform based on the `OAuth` protocol, adopted by all 200 apps surveyed in our use case study (§ 3.2). Originally the login is implemented by the app developers through placing a `Login Button` provided by the Facebook SDK which wrapped the function call to Facebook SDK APIs.

Once clicked, it will initiate the login flow by navigating to the web browser on the user's device (or the Facebook app) to input the users' login credential on facebook.com and finish the authorization to the app. After the login process ends, the app checks whether the user is logged in by calling `AccessToken.getAccessToken`, a Facebook SDK API that will return the sensitive `AccessToken` if the user login is successful. Note that, `AccessToken` is a sensitive data because it could be used for fetching additional sensitive information (e.g., user emails, profile images) from Facebook's remote server. If the login succeed, the app will navigate the user to a new `Activity` for more functionalities; otherwise the user will stay in the current `Activity` that has the `Login Button` for user to attempt login again.

Usability evaluation: migration efforts for app developers. We made 279 lines of code (LOC) changes for this login functionality, including 47 LOC for implementing a *Sensitive Module* (see below). We additionally developed a *data-sharing policy* with 5 LOC that only allows the Facebook data to be sent to the app server (see the policy at [5]). The Facebook demo app originally has more than 1,200 LOC. Specifically, we made the following two code logic changes for the app. First, we replace the app's function calls to the Facebook SDK API behind the `Login Button` with an invocation to the `getDataHandle` API (API 1) provided by our new *SDK runtime* (i.e., *trusted API service*), and upon the invocation, the *trusted API service* will relay the call and actually invoke the APIs in the original Facebook SDK. Then the login process continues, allowing the user to enter their Facebook login credentials in the web browser (or in the Facebook app if already installed on the device) and authorize the developer's app. The second change needed is to refactor the app's code logic that navigates to different activities based on the login results. Specifically, we implemented a *Sensitive Module* (47 LOC) that uses the Facebook SDK API `LoginManager.getLoginStatus` or `AccessToken.getCurrentAccessToken` (by calling API 1 of the *trusted API service* and pass in the SDK API names): the former returns login success or failure, and the latter returns `AccessToken` whose value exists only after a successful login. Our *Sensitive Module* then requests the *trusted API service* to send the login result to the app's server using the `API 2: sendSensitiveData(data_handles, remote_host)`. Then, extra code logic in the *app runtime* that pulls from the developer's server the actual login result is added to the app.

Effectiveness evaluation: privacy enhancement. In the original app, calls to the sensitive `AccessToken.getCurrentAccessToken` function is used to check whether the login is successful or not by the app. This exposes the token from the Facebook SDK to the app code space; also any third-party library in the app can invoke this Facebook SDK API to get the user's Facebook access token. In addition, the function calls to initiate the login process has to be replaced with the invocation to the `getDataHandle` (compared to the original app that directly invokes Facebook SDK API), also ensuring that sensitive information stayed in the SDK Runtime, not exposed to the *app runtime*. In our *PESP* enhanced version of the app, Facebook SDK data cannot flow out to the app runtime regardless of how the app invokes *trusted API service* or the *SDK runtime*, all confined in the *SDK runtime*.

5.2 Case Study and Security Analysis: Display of Facebook User Profile under *PESP*

Another common use case used by 169 apps out of the 200 apps we found in the use case survey (§ 3.2) is to display usernames and email addresses obtained from the social network inside the app's UI. To implement this use case, app developers would first acquire the sensitive user profile information from the Facebook SDK with an invocation to the SDK API `Profile.getCurrentProfile`, whose return value contains the username and email address. After that, the username and email are used to update the content in the Android `View` objects to display them in the UI, so the user can check the profile information they are going to use with this app, and modify them on the UI as needed. If the user confirmed the profile information, the app will send the profile information to the app server for persistent storage.

Usability evaluation: migration efforts for app developers. We changed 196 lines of code (LOC) for this functionality, including a *Sensitive Module* (73 LOC). This demo app of Facebook originally has more than 1,200 LOC. Specifically we make the following code changes for the app to adopt *PESP*. We replaced the invocation to the sensitive SDK API `Profile.getCurrentProfile` with an invocation to the `getDataHandle` API of the *trusted API service*, which returns an opaque data handle to the `Profile` object (the actual value is stored in the SDK Runtime's in its *Sensitive Storage*). Next, we removed the original app code logic that updates the `View` objects with the sensitive user information and instead developed a *Sensitive Module* (73 LOC) that implemented a GUI view to display the user information (also see the "*PESP* privacy-preserving UI paradigm" in § 4.2). Further, the app code logic of sending the user profile information to the app server is replaced. Instead, the view of the *Sensitive Module* includes action code that sends the user data to its app server. This involves invocation of the API 2 of the *trusted API service*, i.e., `API 2: sendSensitiveData(data_handles, remote_host)` by the *Sensitive Module*. The *data-sharing policy* includes 5 LOC that only allows the Facebook data to be sent to the app server (released online [5]).

Effectiveness evaluation: privacy enhancement. Thanks to the Sensitive Module and *PESP* privacy-preserving UI paradigm, sensitive user profile are displayed in the UI without being exposed in the *app runtime* at any time after the

migration, eliminating the *XLDH*. In addition, the *trusted API service* of the *SDK runtime* can enforce the *data-sharing policies* whenever the *Sensitive Module* tries to (through the *trusted API service*) send any data out to any remote hosts.

5.3 Case Study and Security Analysis: Login with Twitter under *PESP*

We changed 125 lines of code (LOC) for this functionality, including a *Sensitive Module* (52 LOC). This demo app of Twitter originally has more than 1,300 LOC. The design of Twitter Kit (SDK) for Android is very similar to the Facebook SDK for implementing the Login with Twitter use case. Specifically, Twitter Kit also provided a `Login Button` that wrapped the actual `OAuth` based user login process and provided `Call Back` interface such that the app developer could trigger their own code logic depending on the success or failure of the login. Different in the Twitter Kit case is that Twitter uses a pair of `oauth_token` and `oauth_token_secret` inside the `TwitterSession` object as the credentials for pulling user private information from the Social Network platform. The `TwitterSession` object also contains sensitive user information like the user id and user name. In addition, the demo app included an extra step of calling Twitter Kit SDK API `TwitterAuthClient.requestEmail`, which essentially sends a `HTTP Get` request to the Twitter's server to retrieve the user's email address, to verify the credentials are valid.

Usability evaluation: migration efforts for app developers. The Twitter Kit's API `TwitterAuthClient.requestEmail` original implementation takes an app developer implemented `Call Back` interface and return the user's email in the success clause of the `Call Back`. After adopting our design, an opaque data handle is returned whether the login is successful or not to avoid privacy leakage. As a result, we developed a Sensitive Module (19 LOC) to check the existence of user email with the opaque data handle and report the result to the app's server (which is registered in advance and approved by the *data-sharing policies*), then the app code retrieves the results with a network request. The *data-sharing policy* includes 4 LOC that only allows the Twitter data to be sent to the app server (released online [5]).

Effectiveness evaluation: privacy enhancement. In the original app, the `TwitterSession` object returned by the original Twitter Kit API that could have leaked user email, user id and sensitive Twitter API credentials to the untrust app space exposing malicious libraries. With the *PESP* paradigm in the app, this is protected based on opaque data handles and the actual data never went from the Twitter SDK to the *app runtime*. Further, the Sensitive Module design ensures the sensitive user email to be accessible for app functionalities while the data flows are fully controlled based on the *data-sharing policies*.

Table 2: Performance overhead (single social SDK per app)

Use Case	Original (ms)	PESP-migrated (ms)	Δ (ms)
Login with Facebook	66.45 ± 5.60	138.88 ± 13.24	72.42
Display User Profile	50.01 ± 0.38	89.19 ± 6.732	39.19
Login with Twitter	167.50 ± 6.02	230.10 ± 16.52	62.60

Assuming Normal Distributions, Intervals for 95% confidence

5.4 Performance Overhead

5.4.1 Evaluation for the Three Use Cases

We conducted our experiment on a Google Pixel 6 phone running stock Android 13 (r16). We used the three migrated use cases (with *PESP* adopted in the two apps) above to evaluate performance overhead of our design. Since the three use cases' original implementation in the sample apps is mainly for demo purpose (Facebook and Twitter released the sample apps to show how to use their SDKs and for what use cases) with no code logic for other app functionalities (i.e., functionalities unrelated to Facebook or Twitter) in place, our evaluation is thus conservative and approximates the potential upper bound of the performance overhead. We run the test 20 times for both the original and migrated, *PESP*-based app implementation and Table 2 summarized the results.

Login with Facebook. We measure the time the app needs to receive the login success result after the login button is clicked. In the workflow, once the user clicks the in-app login button, she is redirected to Facebook.com in the browser (or the Facebook app if installed), authorizes the app, and is finally redirected back to the app's specific UI window with successful login results. For measurement purpose, we subtracted the time the user needed to authorize the app in the Web browser (or in the Facebook app), which can be influenced by the user's reaction speed and network latency. The subtracted time is measured by inserting code for timestamp logging in the apps related to UI operations (e.g., when the login button is clicked) and activity switching (e.g., right upon the app is switching to the browser and right after the login success UI activity is triggered following the authorization). The average overhead for the migrated implementation is 72.42 *ms*.

Display Facebook User Profile. We measure the time needed starting from in-app invocation of the function that retrieves user profile information to when the rendering of the views containing the sensitive information is finished. In the workflow, once the app triggers the API `GraphRequest` of the Facebook SDK (using the API `getDataHandle` of *PESP*), the SDK will retrieve the user data from the Facebook server, and then display the data on a UI activity implemented by the app (running in a *Sensitive Module*). We subtracted the time spent on network requests for the user information from the Facebook server to exclude the impact of network latency variations. The subtracted time is measured by inserting code in the open-source Facebook SDK for timestamp logging

at corresponding network request callbacks (i.e., before and after the request). The result shows an overhead of 39.19 *ms*.

Login with Twitter. Same as the Login with Facebook case, we measure the time the app need to receive the login successful result after the login button is clicked excluding the time the user spent in the browser. The result shows a overhead of 62.60 *ms*.

5.4.2 Apps with Multiple Social SDKs

In the above experiments, each app comes with one *social SDK*. We further evaluated performance overhead on multiple apps, with each app using both Facebook and Twitter SDKs under *PESP*. Specifically, by combining the above two PESP-enabled apps that use either the Facebook or Twitter SDK (§ 5.1 to § 5.3), we developed an app that uses both social SDKs under *PESP*. We take this app as the evaluation target app. As its control group, by combining the two original example apps (provided by Facebook and Twitter), we developed an app using both the Facebook and Twitter SDK (without *PESP*). The evaluation target app and its control group share exactly the same functionalities with both the Facebook and Twitter SDKs. For the evaluation, we performed experiments under three scenarios as follows (each result is based on the average of 20 trials, shown in Table 3).

Scenario 1: multiple PESP-enabled apps use the same *social SDK* (Facebook or Twitter SDK). We launch two instances of the target app (compiled and built under different package names, treated by the Android OS as different apps). We refer to the two instances as PESP Instance 1 and 2 respectively. For each use case (`Login in with Facebook`, `Display User Profile` and `Login with Twitter`), we run PESP Instance 1 and then Instance 2, with their performance overhead measured respectively (Table 3). There is no obvious performance difference between the two instances. Similarly, we also run two instances of the control group app (Original Instance 1 and 2 in Table 3). The performance overhead compared to the control group is low; e.g., Instance 1 shows an average overhead of about 74.79*ms*, 38.47*ms*, 59.91*ms* compared to Original Instance 1 for the three use cases respectively.

Scenario 2: single PESP-enabled app uses multiple *social SDKs* (Facebook and Twitter SDKs). We launch one instance of the target app, run the two use cases sequentially (`Login in with Facebook` and `Login with Twitter`) and measure the time consumed for both use cases. Compared to the control group, the performance overhead for executing the two use cases is low (74.0*ms* and 60.7*ms* respectively for the two use cases), which is about the same with Table 2 that evaluates single *social SDK* in each app.

Scenario 3: multiple PESP-enabled apps use multiple *social SDKs* (Facebook and Twitter SDKs). Similar to Scenario 1, we launch two instances of the target app. Then in each instance we run the two use cases (`Login in with Facebook` and `Login with Twitter`) and measure the time (PESP Instance 1 followed by PESP Instance 2). Compared to the control group, the average performance overhead for executing the two use cases is around 85.6*ms* and 63.6*ms* (PESP Instance 1). There is no obvious performance difference between the two PESP app instances.

6 Discussion

Long-term maintenance and updates. As mentioned in § 4.1, the design and implementation of *PESP* is directly compatible with PSoA of the official Android. *PESP* can supplement PSoA to protect *social SDKs*, while PSoA currently only supports ads-related SDKs. Notably, in *PESP*, the individual *social SDKs* are wrapped under the *SDK runtime* and the original *social SDKs* expect minimum or no changes. The design and implementation of such a *SDK runtime* is general, not specific to individual social networks (our implementation that can directly wrap the Facebook, Twitter or other *social SDKs* is released online [5]). With future emerging threats against the SDKs, one may just update the general *SDK runtime*, which can be maintained with the Android or community efforts including ours. We expect that *PESP* will motivate more research and development efforts that tackle hard defense problems or emerging privacy threats and improve privacy assurance for mobile apps and *TPL*s.

Sensitive information returned from the app server. After collecting user data from the *SDK runtime*, benign app servers might hand the data to the *app runtime*, flowing private data into untrusted code space. Although we acknowledge that this presents a privacy threat (similar to prior studies showing that malicious libraries can harvest the host-app's data from the app-specific UI or server), it is not a *XLDH* threat and has significantly more limited attack surface than *XLDH*. Specifically, from the attackers' perspective (a data-harvesting library that has sneaked into mobile apps), with *XLDH*, the malicious library can succeed in all apps using the same attack vector/approach (accessing the same function/interface of the target *social SDK*); in contrast, the other threat in question has to target app-specific UI components, code modules or interfaces to find the private data, which is significantly less effective and more costly for the attackers. Hence, our design defeats *XLDH*, significantly reduces privacy attack surfaces and elevates privacy assurance.

Generalizability to other types of SDK. Our design is general (not relying on *social SDKs*), and may be applicable to potentially all SDKs that provide or manage data, or want to protect their data. Also note that to ensure the privacy assurance against *XLDH* while supporting in-app functionalities of *social SDKs*, our design is based on a thorough survey of in-app use of 20 popular *social SDKs* (§ 3.2). To use *PESP* or extend it for protecting more types of SDKs, a thorough survey of their functionality use cases are necessary, which we leave for our future work. Moreover, *PESP* and PSoA share

Table 3: Performance overhead (multiple social SDKs per app)

Use Case	Original Instance 1	Original Instance 2	PESP Instance 1 (ms)	Δ (ms)	PESP Instance 2 (ms)	Δ (ms)
Login with Facebook (Scenario 1)	65.91 ± 1.92	67.69 ± 2.69	140.7 ± 3.93	74.79	142.3 ± 5.91	74.61
Display User Profile (Scenario 1)	50.33 ± 0.33	50.55 ± 0.53	88.8 ± 1.81	38.47	89.25 ± 3.93	38.7
Login with Twitter (Scenario 1)	167.04 ± 12.51	173.95 ± 17.68	226.85 ± 13.45	59.91	240.15 ± 20.59	66.2
Login with Facebook/Twitter (Scenario 2)	FB:70.2 ± 8.48	–	FB:144.2 ± 11.69	74.0	–	–
	TW:159.9 ± 24.90		TW:220.6 ± 25.40	60.7		
Login with Facebook/Twitter (Scenario 3)	FB:62.0 ± 3.73	FB:70.3 ± 15.11	FB:147.6 ± 25.38	85.6	FB:149.1 ± 36.3	78.0
	TW:158.7 ± 21.1	TW:185.4 ± 25.56	TW:222.3 ± 20.2	63.6	TW:251.2 ± 26.69	65.8

Assuming Normal Distributions, Intervals for 95% confidence (FB: Login in with Facebook; TW: Login with Twitter)

Table 4: Comparison with prior privacy protection techniques

Year	Name	Techniques Category	P1	P2	P3
2012	Aurasium [66]	App Compartmentalization	n/a	✗	✔
2013	AppGuard [33]	App Compartmentalization	n/a	✗	✔
2013	LayerCake [54]	App Compartmentalization	n/a	✗	✔
2014	COMPAC [65]	App Compartmentalization	n/a	✗	✔
2016	DroidDisintegrator [60]	App Compartmentalization	n/a	✗	✔
2016	CASE [73]	App Compartmentalization	n/a	✗	✔
2017	FineDroid [72]	App Compartmentalization	n/a	✗	✔
2017	CompARTist [45]	App Compartmentalization	n/a	✗	✗
2018	BreakApp [62]	App Compartmentalization	n/a	✗	✔
2021	SEApp [55]	App Compartmentalization	n/a	✗	✔
2011	AppFence [43]	Taint Tracking	n/a	✗	✔
2012	DroidScope [68]	Taint Tracking	n/a	✗	✔
2016	TaintART [59]	Taint Tracking	n/a	✗	✔
2016	PIFT [69]	Taint Tracking	n/a	✗	✔
2017	TaintMan [70]	Taint Tracking	n/a	✗	✔
2018	NDroid [67]	Taint Tracking	n/a	✗	✔
2012	AdDroid [50]	TPL Isolation	✗	✗	✗
2012	AdSplit [57]	TPL Isolation	✗	✗	✗
2013	AFrame [71]	TPL Isolation	✗	✗	✗
2013	Sanadbox [46]	TPL Isolation	✗	✗	✗
2014	NativeGuard [58]	TPL Isolation	✗	✗	✗
2015	PEDAL [48]	TPL Isolation	✗	✗	✔
2016	FLEXDroid [56]	TPL Isolation	✗	✗	✔
2016	LibCage [63]	TPL Isolation	✗	✗	✔
2021	LibCapsule [51]	TPL Isolation	✗	✗	✔
2022	PSoA [30]	TPL Isolation	✔	✗	✔
2023	*PESP* (This Work)	TPL Isolation	✔	✔	✔

fundamentals in the runtime and distribution model, particularly isolating the SDK in a runtime separate from the app. With shared fundamentals, *PESP* can be incorporated into PSoA to support social use cases and potentially all SDKs that offer data. Note that currently PSoA primarily focuses on supporting ads-related SDKs, which, compared to *social SDKs*, expect much less or no functionality interactions with the host app.

Applicability to other OS. Our design isn't limited to Android. Android implemented the easy-to-use isolation and flexible UI composition (Section 4.4), which were needed to implement our design. These features/flexibilities were designed for better engineering/security/testing. Other OSes may also implement such features, enabling implementation of our design.

7 Related Work

Third-party library (TPL) isolation. To enable privilege separation, the TPLs are isolated in these techniques through various design. In all the prior approaches [46, 48, 50, 51, 56–58, 63, 71], the design usually focused on preventing TPLs from accessing the Android system resource (by restricting their permissions), without restricting them from accessing data from *social SDKs*. Those prior techniques generally suffered from the Problem 1 and 2 summarized in § 4.3.

App compartmentalization. App compartmentalization related techniques provides finer granularity by differentiating inner parts of an app from other parts when enforcing access control and isolation (e.g., attributing API calls or resource access to individual SDKs, classes, or processes, depending on the designed granularity of individual approaches) [33, 45, 54–56, 60, 62, 65, 66, 72, 73]. Those prior techniques generally suffer from Problem 2 summarized in § 4.3.

Taint based approaches. Taint based approaches [35, 43, 59, 67–70] are not sufficient as a protection method against *XLDH* due to their incomplete coverage and incomplete modeling of the implicit flow, leading to sensitive user information leak without privacy guarantee that our design achieves.

8 Conclusion

In this paper, we generalize and define *privacy-preserving social SDK* and their uses, characterize fundamental challenges for combating the XLDH threat and guaranteeing privacy. Specifically, we present a practical, clean-slate design and end-to-end systems to enable *privacy-preserving social SDK*. Our evaluation demonstrates its satisfactory effectiveness, performance overhead and feasibility for adoption. Our techniques will contribute to significantly elevating privacy and compliance assurance for multiple stakeholders. Our efforts will help policymakers better understand, define and regulate the *privacy-preserving* mobile software supply chain.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. The authors are supported in-part by Indiana University's IAS Collaborative Research Award and by NSF CNS-2330265, 1850725.

References

[1] Appsgeyser App generator. https://appsgeyser.com/.

[2] Facebook and twitter profiles silently slurped by shady code. https://www.theregister.com/2019/11/26/facebook_twitter_data_loss/.

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

[4] Radar sdk. https://radar.com/.

[5] Support Materials. https://sites.google.com/view/socialsdk.

[6] Two third-party sdks allowed secret harvesting of twitter and facebook user data. https://www.zdnet.com/article/two-third-party-sdks-allowed-secret-harvesting-of-twitter-and-facebook-user-data/.

[7] Android Social Library Statistics and Market Share. https://www.appbrain.com/stats/libraries/social-libs, March 2022.

[8] Mobile Sdks Explorer. https://42matters.com/sdks, March 2022.

[9] Wasim A Ali, KN Manasa, Malika Bendechache, Mohammed Fadhel Aljunaid, and P Sandhya. A review of current machine learning approaches for anomaly detection in network traffic. Journal of Telecommunications and the Digital Economy, 8(4):64–95, 2020.

[10] Andorid. New trusted distribution model for SDKs. https://developer.android.com/design-for-safety/privacy-sandbox/sdk-runtime#trusted-sdk-distribution, March 2023.

[11] Andorid. SDKRuntime API. https://developer.android.com/design-for-safety/privacy-sandbox/guides/sdk-runtime#prepare-sdk-app, March 2023.

[12] Andorid. SDKRuntime FAQ. https://developer.android.com/design-for-safety/privacy-sandbox/sdk-runtime#faq, March 2023.

[13] Andorid. SDKRuntime Goals. https://developer.android.com/design-for-safety/privacy-sandbox/sdk-runtime#goals, March 2023.

[14] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. {PolicyLint}: Investigating internal privacy policy contradictions on google play. In 28th USENIX security symposium (USENIX security 19), pages 585–602, 2019.

[15] Benjamin Andow, Samin Yaseer Mahmud, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Serge Egelman. 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.

[16] Android. Android bind service. https://developer.android.com/guide/components/bound-services#Binding, March 2022.

[17] Android. Android binder. https://source.android.com/docs/core/architecture/hidl/binder-ipc, March 2022.

[18] Android. Android dex format. https://source.android.com/docs/core/runtime/dex-format, March 2022.

[19] Android. Android graphics. https://source.android.com/docs/core/graphics, March 2022.

[20] Android. Android IsolatedProcess. https://cs.android.com/android/platform/superproject/+/master:system/sepolicy/private/isolated_app.te?q=isolatedProcess, March 2022.

[21] Android. Android mediaprojection class. https://developer.android.com/reference/android/media/projection/MediaProjection, March 2022.

[22] Android. Android runtime (art) and dalvik. "https://source.android.com/docs/core/runtime, March 2022.

[23] Android. Android service class. https://developer.android.com/reference/android/app/Service, March 2022.

[24] Android. Android system alert window permission. https://developer.android.com/reference/android/Manifest.permission#SYSTE_ALERT_WINDOW, March 2022.

[25] Android. Android UI Development. https://developer.android.com/develop/ui, March 2022.

[26] Android. Application sandbox. https://source.android.com/docs/security/app-sandbox, March 2022.

[27] Android. SEandroid. https://selinuxproject.org/page/SEforAndroid, March 2022.

[28] Android. SurfaceControlViewHost. https://developer.android.com/reference/android/view/SurfaceControlViewHost.SurfacePackage, March 2022.

[29] Android. View. https://developer.android.com/reference/android/view/View, March 2022.

[30] Android. Privacy sandbox sdk runtime on android. https://developer.android.com/design-for-safety/privacy-sandbox/sdk-runtime, March 2023.

[31] Apple. Apple Privacy Detail. https://developer.apple.com/app-store/app-privacy-details/, 2022.

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

[33] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp Von Styp-Rekowsky. Appguard–enforcing user requirements on android apps. In International Conference on TOOLS and Algorithms for the Construction and Analysis of Systems, pages 543–548. Springer, 2013.

[34] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pages 2187–2200, 2017.

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

[36] Facebook. Facebook Taking Legal Action Against OneAudience Abuse. https://about.fb.com/news/2020/02/taking-action-against-platform-abuse/, February 2020.

[37] Facebook. Facebook sdk demo app. https://github.com/facebook/facebook-android-sdk/tree/main/facebook-login, March 2022.

[38] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. {FlowFence}: Practical data protection for emerging {IoT} application frameworks. In 25th USENIX security symposium (USENIX Security 16), pages 531–548, 2016.

[39] Forbes. Facebook Sues Analytics Firm It Says Was Harvesting User Data. https://www.forbes.com/sites/emmawoollacott/2020/02/28/facebook-sues-analytics-firm-it-says-was-harvesting-user-data/, February 2020.

[40] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In 2017 IEEE Symposium on Security and Privacy (SP), pages 1041–1057. IEEE, 2017.

[41] Frida. Frida Android. https://frida.re/docs/android/, March 2022.

[42] FTC. FTC Civil Penalty Amounts. https://www.ftc.gov/news-events/news/press-releases/2022/01/ftc-publishes-inflation-adjusted-civil-penalty-amounts-2022, 2022.

[43] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In Proceedings of the 18th ACM conference on Computer and communications security, pages 639–652, 2011.

[44] Jie Huang, Michael Backes, and Sven Bugiel. A11y and privacy don't have to be mutually exclusive: Constraining accessibility service misuse on android. In 30th USENIX Security Symposium, 2021.

[45] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. 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.

[46] Hideaki Kawabata, Takamasa Isohara, Keisuke Takemori, Ayumu Kubota, Junya Kani, Harunobu Agematsu, and Masakatsu Nishigaki. Sanadbox: Sandboxing third party advertising libraries in a mobile application. In 2013 IEEE International Conference on Communications (ICC), pages 2150–2154. IEEE, 2013.

[47] Court Listener. Facebook, Inc. v. OneAudience LLC (3:20-cv-01461). https://www.courtlistener.com/docket/16898689/facebook-inc-v-oneaudience-llc/, February 2020.

[48] Bin Liu, Bin Liu, Hongxia Jin, and Ramesh Govindan. Efficient privilege de-escalation for ad libraries in mobile apps. In Proceedings of the 13th annual international conference on mobile systems, applications, and services, pages 89–103, 2015.

[49] Alessandro Narduzzo, Alessandro Rossi, et al. Modularity in action: Gnu/linux and free/open source software development model unleashed. Technical report, Department of Computer and Management Sciences, University of Trento, Italy, 2008.

[50] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, pages 71–72, 2012.

[51] Jun Qiu, Xuewu Yang, Huamao Wu, Yajin Zhou, Jinku Li, and Jianfeng Ma. Libcapsule: Complete confinement of third-party libraries in android applications. IEEE Transactions on Dependable and Secure Computing, 2021.

[52] The Register. Facebook fires sueball at 'malicious' app SDK makers, accuses them of gobbling up people's personal information. https://www.theregister.com/2020/02/28/facebook_sues_developer/, February 2020.

[53] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. 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.

[54] Franziska Roesner and Tadayoshi Kohno. Securing embedded user interfaces: Android and beyond. In USENIX Security Symposium, pages 97–112, 2013.

[55] Matthew Rossi, Dario Facchinetti, Enrico Bacis, Marco Rosa, and Stefano Paraboschi. {SEApp}: Bringing mandatory access control to android apps. In 30th USENIX Security Symposium (USENIX Security 21), pages 3613–3630, 2021.

[56] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. Flexdroid: Enforcing in-app privilege separation in android. In NDSS, 2016.

[57] Shashi Shekhar, Michael Dietz, and Dan S Wallach. {AdSplit}: Separating smartphone advertising from applications. In 21st USENIX Security Symposium (USENIX Security 12), pages 553–567, 2012.

[58] Mengtao Sun and Gang Tan. Nativeguard: Protecting android applications from third-party native libraries. In Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks, pages 165–176, 2014.

[59] Mingshen Sun, Tao Wei, and John CS Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 331–342, 2016.

[60] 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.

[61] Twitter. Twitter kit demo app. https://github.com/twitter-archive/twitter-kit-android/tree/master/samples/app, March 2018.

[62] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. Breakapp: Automated, flexible application compartmentalization. In NDSS, 2018.

[63] Fabo Wang, Yuqing Zhang, Kai Wang, Peng Liu, and Wenjie Wang. Stay in your cage! a sound sandbox for third-party libraries on android. In European Symposium on Research in Computer Security, pages 458–476. Springer, 2016.

[64] Jice Wang, Yue Xiao, Xueqiang Wang, Yuhong Nan, Luyi Xing, Xiaojing Liao, JinWei Dong, Nicolas Serrano, Haoran Lu, XiaoFeng Wang, et al. Understanding malicious cross-library data harvesting on android. In 30th USENIX Security Symposium (USENIX Security 21), pages 4133–4150, 2021.

[65] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce component-level access control in android. In Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, pages 25–36, 2014.

[66] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In 21st USENIX Security Symposium (USENIX Security 12), pages 539–552, 2012.

[67] Lei Xue, Chenxiong Qian, Hao Zhou, Xiapu Luo, Yajin Zhou, Yuru Shao, and Alvin TS Chan. Ndroid: Toward tracking information flows across multiple android contexts. IEEE Transactions on Information Forensics and Security, 14(3):814–828, 2018.

[68] Lok Kwong Yan and Heng Yin. {DroidScope}: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis. In 21st USENIX security symposium (USENIX security 12), pages 569–584, 2012.

[69] Man-Ki Yoon, Negin Salajegheh, Yin Chen, and Mihai Christodorescu. Pift: Predictive information-flow tracking. In Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, pages 713–725, 2016.

[70] Wei You, Bin Liang, Wenchang Shi, Peng Wang, and Xiangyu Zhang. Taintman: An art-compatible dynamic taint analysis framework on unmodified and non-rooted android devices. IEEE Transactions on Dependable and Secure Computing, 17(1):209–222, 2017.

[71] Xiao Zhang, Amit Ahlawat, and Wenliang Du. Aframe: Isolating advertisements from mobile applications in android. In Proceedings of the 29th Annual Computer Security Applications Conference, pages 9–18, 2013.

[72] Yuan Zhang, Min Yang, Guofei Gu, and Hao Chen. Rethinking permission enforcement mechanism on mobile systems. IEEE Transactions on Information Forensics and Security, 11(10):2227–2240, 2016.

[73] Suwen Zhu, Long Lu, and Kapil Singh. Case: Comprehensive application security enforcement on cots mobile devices. In Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, pages 375–386, 2016.

Appendix

Due to space limit, more Appendix can be found at [5].

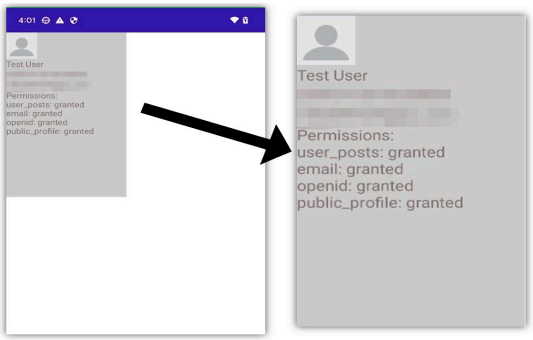
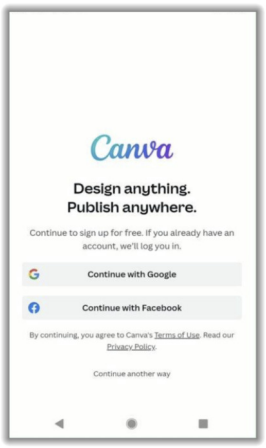
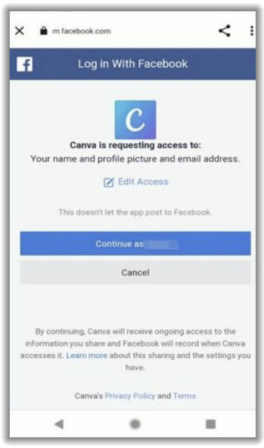


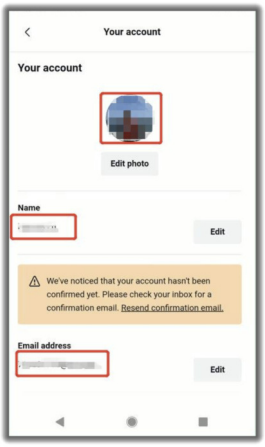
Figure 3: A screenshot of a *privacy-preserving* Facebook SDK used in a demo app, fulfilling the Display User Profile use cases of Facebook SDK (Sensitive Information masked, the area with gray background indicates the displayed filled by the *privacy-preserving* Facebook SDK)



(a) Login with Facebook



(b) To authorize the app



(c) Display of user profiles

(d) In-app posting to social networks

Figure 4: Prominent examples of *social SDK* usage in mobile apps (personal information blurred in red frames, including user profile information and names of the group being joined