VirtualHideDroid: User data anonymization through virtualization techniques

Francesco Pagano[1], Antonio Ruggia[1], Luca Verderame[1]*, and Alessio Merlo[2]

[1] DIBRIS - Università degli Studi di Genova, Genova, Italy
{francesco.pagano, antonio.ruggia}@dibris.unige.it , luca.verderame@unige.it

[2] CASD - School of Advanced Defense Studies, Rome, Italy
alessio.merlo@ieee.org

Abstract

The proliferation of smartphones has raised significant privacy concerns as mobile apps routinely collect and transmit personal user data for statistical analysis, compromising user privacy. Mobile app developers accumulate substantial personal information and profile user behavior by exploiting so-called mobile analytics libraries such as Google Analytics and Facebook Analytics. Unfortunately, in recent years, most of the existing privacy-enhanced solutions follow an "all or nothing" approach, leaving the user the sole option to accept or completely deny access to privacy-related data. If this is not the case, the available solutions present technical limitations (e.g., use of VPN features) or require modifications of the target app (e.g., app repackaging). This paper introduces VirtualHideDroid, a methodology that exploits virtualization techniques to anonymize user data extracted by analytics libraries. Unlike existing solutions, VirtualHideDroid empowers users to control privacy levels for individual apps, addressing the limitations of current approaches. The research demonstrates the effectiveness, feasibility, and performance of VirtualHideDroid through an experimental campaign using a prototype version of VirtualHideDroid for Android OS and 375 top-ranked apps from the Google Play Store.

Keywords: Mobile Security, Data Anonymization, Android Virtualization, Dynamic Analysis

---

The 7th International Conference on Mobile Internet Security (MobiSec'23), Dec. 19-21, 2023, Okinawa, JAPAN, Article No. 69

*Corresponding author: Università degli Studi di Genova, Genova, 13 - 16145, Italy, Email: luca.verderame@unige.it

1 Introduction

In the last few years, the number of people owning a smartphone has grown significantly. In particular, statistical studies report an increase of nearly 50% since 2017 and predict the upward trend in the following years [1]. The same direction applies to the number of available applications (hereafter, apps).

Whenever a user interacts with the apps installed on the device, the same apps collect and send personal user data to the service back-ends. The main reason is that personal user data is helpful for companies to perform statistical analysis, leading to a huge privacy issue: companies profile user behavior to maximize their profits. The raised privacy concerns mainly relate to the user being out of the loop in this collection and profiling process. For instance, Facebook analytics services extract sensitive information about the position of the user and the actions that she executes during the app usage. In this way, Facebook can customize the user experience, enticing the user to continue using the social platform.

To try to limit the privacy issue, in 2018 *General Data Protection Regulation* (GDPR) [2] has been released to impose obligations on all organizations that collect data related to users in the European Union. In particular, GDPR establishes that a company must evaluate the risks of data breaches to avoid introducing "privacy by design" solutions. GDPR requires companies to expose clear privacy policies to users, who must accept them before using their services. Furthermore, the GDPR limits data transfer operations and data storage within the EU. However, this regulation only affects non-European countries, leaving the data collection without law for those countries' citizens.

The privacy-related phenomenon is relevant in several scenarios (e.g., IoT [3]) but particularly recurrent in the mobile system, where a user's mobile device collects much personal information.

On the user side, the amount of produced personal data on mobile is increasing. To this aim, it is worth noticing that over the last two years, the spread of social apps has increased, as shown in [4].

The app developers profile the users through proper third-party libraries, known as **analytics libraries**, which can profile the users' actions and embed advertisements programmatically. The most popular analytics library is Google Analytics [5], which is embedded in 58% of the Android apps analyzed in [6]. However, there exist state-of-the-art works that analyze ana-

lytics libraries, i.e., the work of Chen et al. [7] demonstrates how an external adversary could extract sensitive information regarding the user and the app by exploiting two mobile analytics services: Google Mobile App Analytics and Flurry. The authors in [8] highlighted the privacy problem related to a misconfiguration of analytic services. Moreover, several works, such as [8], reported how analytic libraries share the same privileges and resources as the hosting app, allowing them to access and collect sensitive information regarding the users and their behaviors without proper privacy-preserving mechanisms. Furthermore, these libraries must comply with the GDPR to be used in European countries. In June 2022, for instance, the Italian Privacy Guarantor[9], an administrative authority that ensures the enforcement of the GDPR in Italy, definitively banned Google Analytics services. Although considerable privacy changes have been made, version 4 of Google Analytics (released in 2020) is incompatible with GDPR[10]. This again highlights how privacy is an open problem, especially on mobile devices.

To solve this issue and enhance the privacy of the collected data through anonymization techniques, the research community has proposed several works in the last few years to protect the users' privacy. The authors in [11] proposed an Android app called Lumen Privacy Monitor that allows the user to block requests containing personally identifiable information (e.g., IMEI, MAC, Phone Number). The authors in [12] proposed a modified version of the Android OS called MockDroid, which allows users to revoke access to specific resources at run-time. The most promising one is HideDroid [13, 14], which exploits local differential privacy techniques to anonymize user data extracted from analytics services, both allowing users to control the level of privacy of the collected data at the granularity of a single app and the developers to obtain some (albeit reduced) useful data for assessment. The privacy concerns have also reached Google [15] and Apple [16], which, in their last release, introduced a sandbox mechanism to protect and anonymize the user's data. Apple's App Tracking Transparency framework allows users to deny the app access to their data.

Unfortunately, state-of-the-art solutions have some limitations. First, they - except HideDroid - do not provide any control over the collected data and the anonymization process to the user, who is the data owner. Also, Google's sandbox does not provide automatic user-data anonymization mechanisms but leaves the task to every app developer, who can decide what information to anonymize. Finally, they have some technical limitations: [11] follows an "all or nothing approach", giving the sole option to either fully accept or deny all the collection of personal data. At the same time, HideDroid [14] performs an app repackaging step [17] to

intercept and modify the network requests, which is problematic if the app uses encryption or custom data encoding. For instance, HideDroid cannot anonymize Google Analytics requests encoded in protobuf, a private Google data packing format.

The common and current objective is to guarantee the user a level of privacy that allows him not to be identified as a single individual but which, on the other hand, allows companies to extract valuable data for their statistical and market purposes.

Paper Contribution. In this work, we aim to anonymize the *flow of information and data between analytics services and mobile applications without requiring modifications of the target apps or exploiting OS features discontinued in the newest OS version*. Using *Android Virtualization*, the methodology can intercept API calls of analytics libraries within the target app and anonymize the collected data through generalization and differential privacy techniques.
Also, we implemented ***VirtualHideDroid***, an Android application capable of hosting other Android apps and anonymizing the user's data they extract through the analytics libraries, including Google Analytics.
Finally, to demonstrate the effectiveness, feasibility, and performance of *VirtualHideDroid*, we conducted an experimental campaign on 375 of the most downloaded apps from the Google Play Store, taken from different categories.

Paper Structure. The paper is organized as follows: section 2 presents background knowledge on analytic libraries, data anonymization, and virtualization technologies. Then, Section 3 contains an overview of the current state-of-the-art in data privacy in Android; Section 4 describes the methodology at the basis of *VirtualHideDroid*, while Section 5 provides more technical details on its prototype implementation. Section 6 discusses the experimental setup and the experimental results. Finally, Section 7 concludes the thesis with a brief recap and a look at possible future work.

2 Background

This section introduces the concepts of mobile analytics libraries and Android Virtualization and gives an overview of the leading Data Anonymization techniques.

2.1 Analytics Libraries

The increasing demand for high-quality apps and functionalities leads to fierce competition among mobile developers to improve the quality of their products. The best way to find what people want and need to satisfy is to study and understand their behavior and approach to the apps. To this aim, developers started to adopt analytics libraries to collect information about the user, the usages of the app, and the device to understand the market need. In particular, analytics libraries generally consist of two parts:

1. a back-end service that exposes a set of APIs to generate, collect, and manipulate information regarding the apps' usage and offers developers monitoring dashboards to configure and access the collected data;

2. a Software Development Kit (SDK) that includes tools and libraries that support the interaction with the background service.

From a technical standpoint, the data are collected as *Events* or *User Properties*. The Events allow measuring users' interaction on an app. They are formed by a name that identifies them and some parameters that give more information about the event. For example, the *add_to_cart* event, has *currency*, and *value*, *items* as parameters. Typically, analytic libraries offer a set of standard events but allow the creation of custom ones. User Properties, instead, are attributes that can describe segments of the user base. For instance, the developer can set properties like the `favorite_food` to specify the user's favorite food or the `language` to acquire the user's main language.

2.1.1 Google Analytics Libraries

One of the most widely used analytic services is Google Analytics; it is leveraged by Firebase, an app development platform backed by Google, for data collection in the mobile environment[18]. According to the statistical data of appBrain [19], Firebase analytics is the most used analytic service in the mobile app context. In particular, nearly 70% of the apps on the USA Play Store use Google Firebase Analytics, and nearly 95% of the top 500 apps leverage analytics services.

The most recent version of Google Analytics is GA4, which was created to cope with privacy problems and improve some functionalities. In such a library, the Events for mobile apps are divided into three categories: automatically collected events, recommended events, and custom events [20]. The automatically collected ones are, for example, the events related to

actions like app removal, *app_ remove*, first open after installation, *first_ open*, changing screen, *screen_ view*, and so on. Instead, the developer must trigger the other two types into the app's code. Firebase library provides a method – i.e., `logEvent()` – which takes the name and the bundle with the parameters as input. The recommended events help you measure additional features and behavior and generate more useful reports. Since these events require additional context to be meaningful, they're not sent automatically. They are, for example, *add_ payment_ info* [21], which has to be used when a user has submitted their payment information. Custom events are those whose names and parameters can be set arbitrarily by the developer based on the context.

Similarly to the Events, there are automatic and custom User Properties. The custom user properties can be set by the developer using the `setUserProperty()` method provided by Google Analytics libraries, which take as input a name and a value, e.g., *favorite_food* and *pizza*. The automatic user properties send information about the app's usage, the user, and the session, *first_ open_ time*, *session_ id*.

Since they are very similar, we generically refer to Events and User Properties with *events*, making explicit some differences only when necessary.

2.2 Android Virtualization

Android virtualization (AV) enables an app (known as *container*) to create a virtual environment where other apps (known as *plugins*) can run. A user can install several containers, which generate their corresponding virtual environment, where the user can execute several plugins independently from the underlying Android OS and other virtual environments. DroidPlugin [22] and VirtualApp [23] are the two most well-known frameworks supporting the generation of Android virtual environments and sharing a standard design. The AV technology does not require any additional privileges enabled on the device (e.g., root privileges, custom ROM). It allows the use of multiple apps simultaneously, even if they are not installed on the device. To do so, the container must manage the life cycle of all the plugin components, hiding the app from the Android OS.

The virtualization technology relies on the *Dynamic Code Loading* (DCL) and *Java dynamic proxy* [24]. The former enables an Android app to load external code in a supported executable format, such as DEX and APK files. From a technical standpoint, DCL allows bypassing the 64K reference limit imposed by the DEX format. On the other hand, the Java dynamic proxy

allows the creation a wrapper object to intercept all method calls toward a specific object instance, eventually adding some functionality.

Figure 1 shows the internal architecture of an app generating a virtual environment for a plugin. The architecture involves the following components: the Container App (i.e., container), the Plugin App (i.e., plugin), the Dynamic Proxy Module, and the ClassLoader.

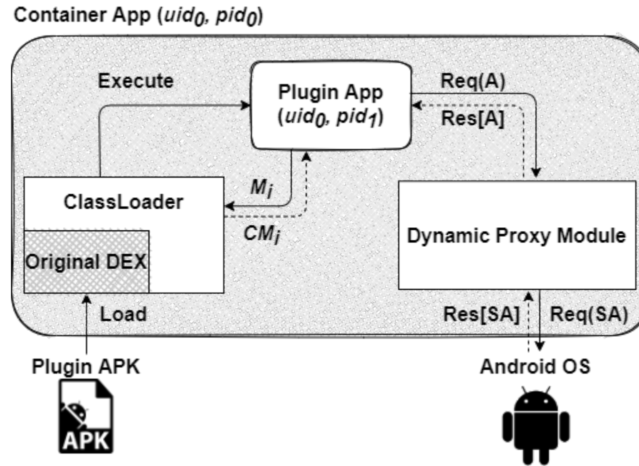


Figure 1: The internal structure of an Android virtual environment.

In the Android system, Java code (i.e., APK or DEX files) is loaded by ClassLoader. To make the classes and methods of the plugin, the container extracts the bytecode of the plugin from its APK (i.e., the `classes.dex` file(s)) and loads them into the ClassLoader of the container. Then, it forks its process (i.e., pid_0) into a new process to host the plugin (i.e., pid_1). Each Android app is assigned a unique user id (UID) and group (GID) when installed. Therefore, the container and the plugin share the same UID (i.e., uid_0), allowing the container to access all the plugin's resources and vice-versa. However, plugins and containers run in different processes. At runtime, the ClassLoader resolves classes and methods (i.e., M_i) for the plugin by returning the appropriate code (CM_i).

Unlike the UI components (e.g., button, view), the life cycle of the app components (e.g., Activity) can be managed only by the Android system. Thus, to hide the plugin app, the container has to manage the life cycle of all the plugin's components. In detail, Android apps contain several components (i.e., Activities, Services, Broadcast Receivers, and Content Providers), which must be declared in the `AndroidManifest.xml` file so the Android OS can register them at the installation time. The management of each component involves an interaction between the

Android app and the Android OS [25]: for example, to show an Activity (`A`) on the screen of the smartphone, an app has to send a request to open that Activity (`startActivity(A)`) to the Android OS and it has to receive a reply containing the details of the device where the app is running (e.g., the dimension of the screen). If an app runs in a virtual environment, the Android OS receives all requests from the container. Thus, the container is expected to declare the same components as the plugins running in its virtual environment: it declares a generic number of *stub* components in its Manifest file. Thus, it can rely on the Dynamic Proxy Module to intercept each request (reply) going towards (coming from) the Android OS to dynamically change the component's name. From a technical standpoint, this module relies on the Java dynamic proxy, creating a proxy object (known as *hook*) for each Android manager. In the example of Figure 1, the Proxy module creates a Stub Activity (SA) and sends the modified call to the Android OS (`startActivity(SA)`). Then, the result of the invocation ($Res[SA]$) is mapped back to the original component ($Res[A]$), fooling the plugin app into being executed on the actual device. Concerning permissions, the container requires all existing Android permissions to support all the possible plugins.

2.3 Data Anonymization Techniques

In data privacy, the set of attributes in a microdata set [26, 27] can be mainly divided into three categories:

- *Explicit Identifiers (EI).* EI are user-identifying attributes, such as the name/surname, the social security number (SSN), or the Insurance ID.
- *Quasi-Identifiers (QI).* This category includes attributes that can be combined with other external data sources (e.g., publicly available databases) to identify a user indirectly. Examples of QI include geographic and demographic information, phone numbers, and e-mail IDs.
- *Sensitive Data (SD).* SD are attributes that contain relevant information for the recipient of the microdata set, like, e.g., health diseases, salaries, and eating habits, to cite a few.

The events generated by analytics libraries are multidimensional data. A multidimensional data is presented as a (relational) table, in which each row represents an individual, and each column represents information associated with that individual. Taking a single row, grouping a set of columns to identify the QIs and SDs is possible. Considering the nature and variety

of the information present, this type of data is challenging to anonymize; however, in some cases, it is possible to build hierarchies on the single attribute by varying the granularity of the information carried by the latter, or it is possible to hide some pieces of data or to modify it, to generalize the result.

In general, the choice of anonymization technique depends on the type of data processing, on the level of privacy and utility to be achieved. The two anonymization techniques used for the study will be illustrated below:

- Generalization techniques replace specific values of attributes belonging to the same domain with more generic ones [28]. In a nutshell, given an attribute A belonging to a domain $D_0(A)$, it is possible to define a Domain Generalization Hierarchy (DGH) for a Domain (D) as a set of n anonymization functions $f_h : h = 0, ..., n-1$, such that:

$$D_0 \rightarrow D_1 \rightarrow ... \rightarrow D_n \tag{1}$$

  And:

$$D_0(A) \subseteq D_1(A) \subseteq \cdots \subseteq D_n(A) \tag{2}$$

  It is worth noting that the more generalization functions are invoked on the original data, the higher the resulting privacy value (and the lower the data utility), as heterogeneous data are transformed into a more reduced set of general values. Generalization techniques are suitable for semantically independent data, such as the set of personal data collected by analytics libraries (e.g., the device name or the phone number).

- Differential privacy involves altering the original distribution of a set of interdependent data using a perturbation function [29]. This approach is usually applied in a context where i) the main requirement is the confidentiality of the data exchanged between pairs, and ii) the receivers' identity is unknown a priori. There are two main models for defining DP problems: centralized and local. The centralized model sends the data to a trusted entity that applies DP algorithms and then shares the anonymized dataset with an untrusted third-party client [29]. On the contrary, the local model assumes all external entities and communication channels as untrusted [30], [31]. In such a situation, local DP techniques aim to perform the data perturbation locally before releasing any dataset to an external party. In our scenario, we consider the user the sole owner of its data, and we trust neither the analytics company nor the developer. The local DP model is suitable for

anonymizing the sequences of events logged by analytics libraries. The objective of DP is to transform a sequence of events $(e_1, e_2, \ldots, e_n) \in \mathrm{D}$ in a different sequence of events $(z_1, z_2, \ldots, z_n) \in \mathrm{D}$ through the application of a perturbation function $\mathrm{R} : \mathrm{D} \rightarrow \mathrm{D}$ to each event. This function is commonly a probability distribution, defined a priori: $z_i = R(e_i)$.

3 Related Work

One of the first studies focusing on privacy issues related to mobile analytics services is the one carried out by Chen et al. [7]. In detail, the authors demonstrated how an external, malicious user could extract sensitive information concerning the user and his use of apps by exploiting two mobile analytics services, namely Google Mobile App Analytics and Flurry.

To cope with such concerns, the scientific community proposed several privacy-preserving solutions. For instance, MockDroid [12] is a modified version of the Android operating system, which prevents apps from accessing system resources, including access to the user's personal information. Although this solution guarantees maximum privacy, it cannot preserve any data utility. In addition, MockDroid requires the user to install a modified version of the Android OS, limiting its applicability in real-world scenarios.

Another solution was identified by Zhang et al. with PrivAid [32], a methodology to apply DP techniques to user-generated events collected by mobile apps. The tool replaces the original analytics API with a custom implementation, which collects generated events and applies DP techniques. The anonymization strategy is configured directly by the developer, who can thus reconstruct the distribution of the original events, at least with a good approximation. Privaid exploits local differential privacy techniques to modify the sequence of the events generated by the analytics services. This solution randomly sorts the generated event sequence, disrupting the utility for the end-user. Privaid does not anonymize the content of the analytics request, which contains all the user's personal information that should be appropriately anonymized.

Razaghpanah et al. proposed a solution called Lumen Privacy Monitor. This solution analyzes the device network traffic, searching for requests that carry user personal and device information. The tool notifies the user of its data extraction and allows her to block such requests. The solution does not exploit anonymization techniques to anonymize user data, resulting in a complete loose of utility for the analytics service. The Lumen Privacy Monitor solution intercepts network requests through the Android VPN services [33].

HideDroid [14] proposed as a solution to preserve the privacy-utility balance of data. Specif-

ically, it relies on a VPN that acts as a proxy server between the client, i.e., the app, and the server, i.e., the analytics server, to collect the information and then anonymize it.

Though this solution is one of the most promising to anonymize user data locally on mobile devices, it presents several technical limitations. First, HideDroid does not allow other VPN services to run on the device if the HideDroid anonymization process is active. HideDroid anonymizes user data based on the analytics request interception. Then, suppose the content of the network traffic generated by analytics libraries is ciphered with additional encryption algorithms. In that case, the HideDroid anonymization pipeline cannot decipher it, making the data anonymization phase impossible. For the same reasons, HideDroid cannot anonymize analytics requests encoded in custom formats, such as Protobuf [34].

Also, HideDroid can only anonymize analytics requests generated from the user-selected apps. Therefore, it does not support the traffic encapsulated through third-party apps (e.g., Google Analytics). Finally, in the most recent Android OS version, HideDroid must apply repackaging techniques to modify the app's configurations to intercept the network traffic, impacting the app's runtime execution [1].

To this aim, this work aims to overcome the limitations introduced by these methodologies, such as the block-or-allow approach, which privileges the privacy or utility without a trade-off, or the VPN approach, which presents compatibility problems in the case of another VPN and will no longer be supported by Android in the near future [36].

4 Methodology

In this section, we describe the methodology to anonymize the user's sensitive data and overcome the limitations of state-of-the-art techniques.

Figure 2 shows a high-level overview: the methodology is built on Android Virtualization to anonymize user data without creating a custom ROM or performing a repackaging. Running an app in a virtual environment allows the container to intercept and modify, as will the data of the plugin app.

The methodology is composed of three main modules, namely, *Installer*, *Interceptor*, and *Anonymizer*. In **Step 1** (of Figure 2), the *Installer* installs the plugin app into the container. This phase is crucial due to extracting the SDK of the analytics library – which is responsible for data collection – from the DEX code (i.e., the `classes*.dex` files). Then, the *Interceptor*

[1]Anti-repackaging techniques applied to the original app (see, e.g., [17, 35]) may negatively affect this process.

module of the container hooks these methods and injects the *Anonymizer* into the plugin app, which acts as a proxy layer between the app code and the analytics library and different methods of the analytics library itself. In particular, the *Interceptor* installs hooks to intercept custom and automatic events. In this scenario, we recall that an *hook* is a function injected from the container to handle a specific method invocation in the plugin app's context. For instance, in Android Virtualization, the container app hooks all Android Managers' function calls to hide the plugin app from the Android OS. After this initial phase, the plugin app is up and running.

At runtime, when the *Anonymizer* module is triggered by the hook (**Step 2.a** in the case of custom events, otherwise **Step 2.b** in case of automatic events), it collects and anonymize (**Step 3**) the events, such as user-specific properties. In **Step 4**, the anonymized data are forwarded to the analytics library, which sends them to its backend (**Step 5**). Finally, the data are available to the developer after backend processing for further analysis.

The three methodology modules will be discussed in more detail in the rest of this Section.

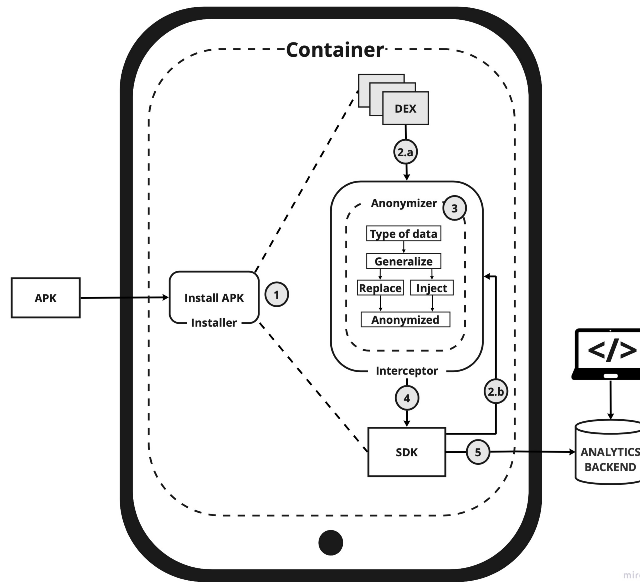


Figure 2: The overall schema of the anonymization methodology.

4.1 Installer

During the installation phase, the container extracts the DEX bytecode of the plugin app from its APK, which also contains the SDK of the analytics libraries. From a technical standpoint,

in the case of custom events, the developer has to explicitly invoke particular library methods (**Step 2.a**), such as `logEvent` for the Google Analytics library. However, the methods responsible for the automatic events are contained in the SDK itself (**Step 2.b**). Thus, the Installer module tries to identify such methods, and if they are present in the code, it installs the Interceptor. It is worth emphasizing that this step is crucial for the rest of the methodology. The anonymization logic cannot be applied if the Installer does not detect the methods or installs the Interceptor into the wrong SDK function. Moreover, this phase is executed only when the plugin app runs for the first time in the virtual environment.

4.2 Interceptor

The Interceptor module intercepts the event creation process (**Step 3**): each time a new event is created (custom or automatic), the container has to anonymize its data. To do so, it behaves like a proxy by intercepting the data flow from the app to the Analytics library's SDK for both the events and the user properties (refer to Section 2.1.1). In detail, it extracts the data in the event and verifies whether they can be anonymized. In such a case, it invokes the Anonymizer, which applies specific techniques based on the data type.

4.3 Anonymizer

The anonymization process is performed directly inside the user's device and applies specific techniques according to the type of intercepted data.

The Anonymizer adapts two different techniques to the analytics libraries scenario, i.e., *generalization* and *differential privacy*. The Anonymizer exploits a buffer to store every anonymized event sent to the Analytics back end. For each intercepted event, the Anonymizer module applies generalization to the data contained in the event. It puts the results into a buffer that includes every anonymized event sent to the Analytics back end. If the buffer does not reach a predefined minimum length, the module terminates its execution. The Anonymizer module triggers the differential privacy algorithm if the buffer reaches a predefined minimum length.

The complete data anonymization procedure follows the algorithm described in Listing 1, detailed in the rest of this Section.

4.3.1 Generalization

The Anonymization module exploits generalization techniques to anonymize the data related to the device and the user. Due to the high heterogeneity of data that can be collected, we opted to suppress certain characters in a number dependent on the level of anonymization set within the container.
In detail, the adopted heuristic consists of the following two rules:

- if the attribute is a sequence of **alphanumeric characters**, the generalization replaces the last p characters with the '*' character. The p-value is defined as a percentage of the length of the string, and it is set statically from the container app.
- if the attribute is a **number**, the value is rounded to the first most significant decimal digit to maintain the semantics of the data such that the analysis backend can correctly process it.

4.3.2 Differential Privacy

The algorithm in Listing 1 takes in input five different parameters, namely, the current intercepted event (`interceptedEvent`), a threshold value to determine which procedure to carry out between *replaceEvent* and *injectEvent* (`threshold`), the percentage of characters to generalized in alphanumeric values (`anonymizationPercentage`), the minimum number of events stored inside the event queue needed to apply the differential privacy(`minNumberOfRequests`), and the list of events already anonymized and sent to analytics back end (`sentEvents`), i.e. the buffer of anonymized events.

The algorithm can be divided into several steps:

- **Initialization phase**: this phase is needed to initialize all the parameters for the differential privacy algorithm (lines 1-4). In lines 1 and 2 it is initialized the `sentEventsSize` variable with the size of `sentEvents`, and it is initialized the `anonymizedEvents` list. The `anonymizedEvents` is a temporary list where the events intercepted by the Interceptor are added after being anonymized and emptied whenever the algorithm returns the events. In line 3, the algorithm initializes the Pr_{inj} variable sampling from a uniform probability distribution. This variable contains the probability that it could generate a new event in addition to the one returned from the Interceptor module. Also, in line 4, the algorithm

initializes the Pr_{rep} variable sampling from a uniform probability distribution. This variable contains the probability that the event returned from the Interceptor module could be substituted with another one taken from the buffer.

- **Less than `minNumberOfRequest` inside the event buffer**: in this case, the Anonymizer module checks if the event buffer contains more than `minNumberOfRequest` events (line 5). If so, the Anonymizer module cannot apply differential privacy and generalizes the event (see 4.3.1) returned from the Interceptor before returning it (lines 7-8).

- **Both Pr_{inj} and Pr_{rep} higher than the `threshold`**: in this case, the Anonymizer module applies differential privacy because the number of events inside the queue is more elevated than `minNumberOfRequest`. In line 10, the Anonymizer checks if both the Pr_{inj} and Pr_{rep} values are higher than the threshold. If so, the Anonymizer randomly selects an action between *replaceEvent* and *injectEvent*, with a probability of 50%, by checking if a random number (initialized at line 11), is less than or higher than 0.5 (line 13). If the first condition is verified, the Anonymizer module performs the *injectEvent* action. In this case, two events are sent: the generalized intercepted event (lines 14-15) and a generalized event taken from the buffer (line 12). Otherwise, the Anonymizer module performs the *replaceEvent* action, returning (line 18) only the anonymized event extracted from the event queue (line 12).

- **Pr_{inj} higher than the `threshold`**: in this case, the Anonymizer module performs the *injectEvent* action (line 22). It generalizes the event coming from Interceptor (line 21), saves it in the temporary list (line 22), extracts another generalized event from the buffer (line 23), and puts it in the queue (line 24). The anonymizer returns both events (line 25).

- **Pr_{rep} higher than the `threshold`**: in this case, the Anonymizer module performs the *replaceEvent* action (line 27). The Anonymizer module extracts a generalized event from the event queue (line 28). It adds the event to the temporary list (line 29). It returns it instead of the event returned from the Interceptor module (line 30).

- **None of Pr_{inj} and Pr_{rep} are higher than the `threshold`**: If any of the Pr_{inj} and Pr_{rep} are higher than the threshold, the Anonymizer module generalizes the event returned from the Interceptor module (line 32) and return it (line 34).

Listing 1 Data Anonymization Pipeline

Input: *interceptedEvent*,*threashold*,*anonymizationPercentage*, $minNumberOfRequests$,*sentEvents*
Output: *anonymizedEvents*

```
1: Initialize sentEventsSize ← sentEvents.size()
2: Initialize anonymizedEvents ← list()
3: Pr_inj ← rand()
4: Pr_rep ← rand()
5: if sentEventsSize < minNumberOfRequests then
6:     newGeneralizeEvent ← generateNewGeneralizeEvent(interceptedEvent.parameters,
   anonymizationPercentage)
7:     anonymizedEvents.add(NewGeneralizeEvent)
8:     return anonymizedEvents
9: else
10:    if Pr_rep > threshold AND Pr_inj > threshold then
11:        randomNum ← rand()
12:        fromRealmAnonymizedEvent ← injectEventFromRealm(sentEvents,sentEventsSize)
13:        if randomNum < 1/2 then
14:            newGeneralizeEvent ← generateNewGeneralizeEvent(interceptedEvent.parameters,
   anonymizationPercentage)
15:            anonymizedEvents.add(newGeneralizeEvent)
16:        end if
17:        anonymizedEvents.add(fromRealmAnonymizedEvent)
18:        return anonymizedEvents
19:    end if
20:    if Pr_inj > threshold then
21:        newGeneralizeEvent ← generateNewGeneralizeEvent(interceptedEvent.parameters,
   anonymizationPercentage)
22:        anonymizedEvents.add(newGeneralizeEvent)
23:        fromRealmAnonymizedEvent ← injectEventFromRealm(sentEvents,sentEventsSize)
24:        anonymizedEvents.add(fromRealmAnonymizedEvent)
25:        return anonymizedEvents
26:    end if
27:    if Pr_rep > threshold then
28:        fromRealmAnonymizedEvent ← injectEventFromRealm(sentEvents,sentEventsSize)
29:        anonymizedEvents.add(fromRealmAnonymizedEvent)
30:        return anonymizedEvents
31:    end if
32:    newGeneralizeEvent ← generateNewGeneralizeEvent(interceptedEvent.parameters,
   anonymizationPercentage)
33:    anonymizedEvents.add(NewGeneralizeEvent)
34:    return newAnonymizedEvents
35: end if
```

5 Implementation

We implemented the methodology in a prototype tool called ***VirtualHideDroid*** for the Android OS that anonymizes the traffic generated by analytic libraries of the apps executed inside its virtual environment. To prove the effectiveness of our methodology, we explicitly target Google Firebase Analytics since state-of-the-art solutions cannot apply proper anonymization techniques to the requests generated by this library, as explained in Section 3.

In the rest of this Section, we present the internal aspects of the Google Analytics library

to understand how analytics events are generated. Then, we detail the implementation of *VirtualHideDroid*, the use of AV, and the adopted anonymization algorithms.

5.1 Anatomy of Google Analytics

To understand how this library collects the user's data and information about the device and creates the events, we perform a manual static analysis task leveraging the *jadx* [37] tool. The main goal of this analysis is to understand the data flow to identify the method(s) where both automatic and custom events are processed. It is worth noticing that this phase directly impacts the overhead the methodology introduces because it determines, for instance, the number of methods to hook. For example, installing a hook into a recursive method triggers the procedure each time the function is invoked, resulting in a large resource overhead.

To reduce the impact of overhead, we tried to hook as few functions as possible, focusing on methods that can be used for both custom and automatic event calls.

In the case of Events, we identified a constructor method belonging to the package `com.google.android.gms.measurement.internal` that creates a wrapper object for the Event itself, assigning it a name (of type `java.lang.String`), a type (e.g., automatic or custom), and a `android.os.Bundle`[2] which contains the event's data. By hooking this method, it is possible to have complete control over all the components and attributes of an event, both in the custom and automatic case. Similarly, we hooked a constructor method of the package `com.google.android.gms.measurement.internal` responsible for building User Properties.

Then, the focus shifted to data related to the device. In particular, the Build[38] class was found to be responsible for setting values related to the device. In detail, the operating system version, the SDK version, and the device model were considered. The values we were interested in were enclosed within the static fields of this class. Through reflection [39], it was possible to modify the fields by replacing the original values with anonymized ones, according to the generalization procedure explained in section 4.3.1.

5.2 *VirtualHideDroid*

To hook the analytics library and perform the anonymization task, *VirtualHideDroid* leveraged the feature of Android Virtualization. In particular, it extended the VirtualApp framework [23] to develop a container that enforces anonymization. Thus, the main task of the container app

[2] Android class which creates a mapping from `String` keys to various `Parcelable` values.

is to give rise to the Installer and Interceptor modules of our methodology: it detects and intercepts the call to the target methods, invoking the Anonymizer module to perform the anonymization task.

In the first stage, the container aims to detect the target methods (and classes) in the plugin; thus, it extracts the DEX bytecode of the plugin from its APK (i.e., the `classes*.dex` file(s)) and loads them into the ClassLoader of the container. At runtime, the `loadClass` method of the ClassLoader is in charge of resolving classes and methods for its app. In the case of Android Virtualization, it is responsible for loading classes and methods for both the container and the plugin.

Since we were interested in intercepting and modifying the target methods of the analytics library, the tool intercepts when they were loaded into the plugin app's memory and replaces them with some custom code. To do so, we created a `Custom ClassLoader` (from now is CCL). It is an instance of the `dalvik.system.DexClassLoader`[40], which overloads the `loadClass` method. In particular, the CCL i) resolves the requested class, ii) checks if it contains target methods, and iii) in case of target methods, it injects the custom code (i.e., Interceptor module).

We recall that the Interceptor module is responsible for hooking the target function and executing some custom code. When the Interceptor is triggered, it checks if the event's data can be anonymized, and, in this case, it invokes the Anonymizer module.

The tool implements the anonymization algorithm to cope with the hooked methods of `com.google.android.gms.measurement.internal`, maintaining the same input/output format so as not to disrupt the analytics library logic. In this tool version, the threshold and the anonymization percentage can be defined statically in a configuration file. This initialization step enables the container to control the rate of utility/privacy of the data. To keep track of the events anonymized and sent to analytics, *VirtualHideDroid* leveraged a Realm Database [41].

It is worth noticing that even if Android Virtualization is built on top of the Java Dynamic Proxy, it presents several limitations that cannot match the Interceptor goal. In particular, Java Dynamic Proxy can hook only object instances of classes that implement at least one interface[24]. To overcome such limitations, *VirtualHideDroid* leveraged ART instrumentation. In detail, ART Instrumentation allows the modification of the Android classes directed into the memory. Thanks to the ART instrumentation, the container can override the pointer to a class method (i.e., `entryPointFromQuickCompiledCode` attribute of the ART class) with a pointer to the custom method of a class that is already loaded. To do so, the custom method must

have the same signature as the target one. In the current implementation of *VirtualHideDroid*, we rely on YAHFA [42] to implement the ART Instrumentation. YAHFA provides an efficient way for Java method hooking and replacement.

6 Results

We tested the feasibility and the effectiveness of our methodology enforced by *VirtualHideDroid*. In particular, our tests aim to quantify the utility of the data preserved after anonymization. This section describes the setup and the result obtained by the experimental champing.

6.1 Experimental Setup

In the current implementation, *VirtualHideDroid* targets the latest version of Google Analytics library (version 21.0.0). To test the prototype, we collected a dataset of the 300 top apps on Google Play in August 2023 *VirtualHideDroid* is built on top of the open-source version of the VirtualApp framework. Unfortunately, this version does not support all Android APIs, and only 74 apps are compatible with that version. It is worth noticing that this limitation is not directly related to the methodology but only to the open-source version of the VirtualApp framework.

To test our methodology, we ran *VirtualHideDroid* in an Emulator of a Pixel 3 XL with four cores and 1536 MB of RAM with the Google Play services hosted on a PopOS machine with 16 GB of RAM and an Intel i7 6700.

6.2 Testing pipeline

The pipeline aims to evaluate the effectiveness of *VirtualHideDroid* regarding the anonymization impact on the data's usability. To do so, we executed each app of the dataset through *VirtualHideDroid*, logging the result of the anonymization process (i.e., original and anonymized events).

To automatically interact with Android apps, we customized ARES [43], a black-box tool that leverages Deep Reinforcement Learning to explore the app dynamically. In particular, the extended version of ARES (from now on ARES++) can i) execute several plugin apps in the *VirtualHideDroid* container and ii) log the anonymization results. Thus, the testing procedure consists of the following steps:

1. **Container App installation.** The app is installed through the Android Debug Bridge (ADB) tool.

2. **Plugin App assessment.** For each plugin app (app in the dataset), ARES++ installs and executes it for 5 minutes and spawns a separate process to monitor the logcat leveraging ADB. To do so, ARES++ pushes the plugin app into the SD card and grants permission to access the storage [3] thanks to ADB.

3. **Post-processing analysis.** Finally, we evaluated the impact of the anonymization and usage values of the anonymized data.

6.3 Experimental Results

The experimental campaign allowed the evaluation of the impact of the anonymization of *VirtualHideDroid* on the users' data by checking how its utility is impacted. We recall that the anonymization result depends on two variables: `threshold` and `anonymizationPercentage` of Listing 1. The former determines the amount of data to anonymize. This value, in the prototype, is assigned to 55%. In contrast, the `threshold` determines the probability of injecting or replacing events in the queue, affecting the final sequence. Due to this value affecting a pseudo-random value, we executed the testing pipeline with two different `threshold` values: 40% and 60%.

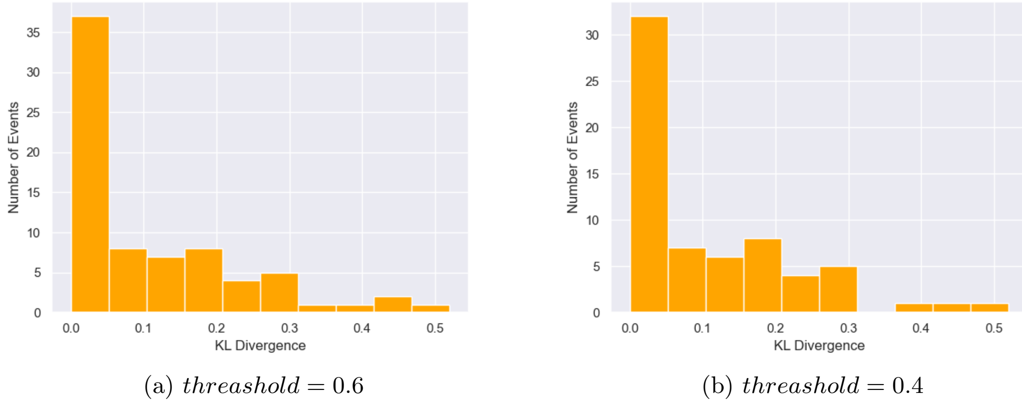
Generalization. As mentioned in Section 4.3.1, the generalization technique used involves the suppression of the `anonymizationPercentage`% of characters. In the current implementation, *VirtualHideDroid* generalizes numeric values up to the nearest ten and 55% of the alphanumeric data. From a usability standpoint, we reduce the sets of the final data (i.e., the cardinality of the possible queries) by factors 0.1 or 0.55, respectively, for the two data types. For instance, the object

$$\{"id" : 194, "name" : "Example"\}$$

contains two attributes: an integer (*id*) and a String (*name*). The result of the generalization is

$$\{"id" : 190, "name" : "Exa****"\}$$

[3] The `WRITE_EXTERNAL_STORAGE` allows to both read and write in external storage in Android



(a) $threashold = 0.6$ (b) $threashold = 0.4$

Figure 3: $KL_{divergence}$ distribution

Differential privacy. To determine the impact of the anonymization process, we exploited two metrics, namely $KL_{divergence}$ (Kullback–Leibler divergence) [44] and *Hamming distance* [45].

The former is a statistical distance, which measures the difference between two pdfs – i.e., distribution of probability functions – $P(x)$ and $Q(x)$. The $KL_{divergence}$ is computed with the following formula:

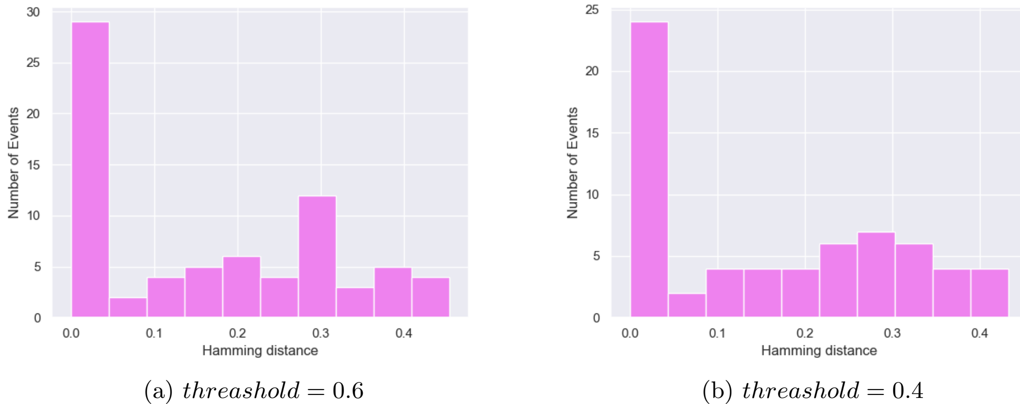
$$KL_{divergence}(P, Q) = \sum_{x} P(x) log(\frac{P(x)}{Q(x)}),$$
$$P(x)|Q(x) = \frac{\#\ occurrences\ of\ x}{\#\ total} \tag{3}$$

In our case, $P(x)$ represents the pdf to have an event of type x in the anonymized event list, while the $Q(x)$ refers to the original list of events. Thanks to this metric, we can understand the loss of events, not the loss of sequence. Figure 3 shows the distribution over the dataset apps for the two different threshold values. As depicted in Figure 3, the distribution is heavily biased (38,5% of apps) toward 0, meaning the two pdfs are semantically identical. In other words, this result means that anonymization does not affect the utility of those apps. However, the distributions with different thresholds are very similar to each other. Some particular apps (4,6%) have a $KL_{divergence}$ higher than 0.3 with a peak of 0.5 (according to Table 2): the utility of the data decreases up to 50%.

The $KL_{divergence}$ extracts only the information about the loss of events in the chain without considering the sequence itself (i.e., the order of the events). For this purpose, we used the

threshold	0.6	0.4
Mean	0.1048	0.1036
Median	0.0519	0.05814
St. dev.	0.1271	0.1211
Max	0.5199	0.5199
Min	0.0	0.0

Table 1: Statistics on values of $KL_{divergence}$



(a) $threashold = 0.6$ (b) $threashold = 0.4$

Figure 4: Hamming distance distribution

Hamming distance. The Hamming distance is the measurement of the minimum number of substitutions you have to perform to make two sequences with the same length equal. In this scenario, a high Hamming distance means that the sequence is completely different even if the proportions between different types of events are maintained (i.e., low $KL_{divergence}$). In terms of usability, a high Hamming distance means a significant loss. Due to the list of events having different lengths for each app, we compute a measure of Hamming distance normalized on the number of events performed by the app under test.

These results (Figure 4 and Table 2) reinforce the results obtained with the $KL_{divergence}$. Figure 4 shows that most apps (77%) have a Hamming distance lower than 0.3, ensuring usability (for the sequence) higher than 0.7. However, the distribution is less condensed near 0 compared to the $KL_{divergence}$ (e.g., the median has increased by a factor of 10), showing a slight decrease in usability. From the developer's point of view, this result highlights how the sequence order is essential and is affected by differential privacy.

Overall, the results achieved in the experimental campaign demonstrate *VirtualHideDroid* anonymizes user data, maintaining an acceptable utility level.

Threashold	0.6	0.4
Mean	0.1591	0.1610
Median	0.1548	0.16667
St. dev.	0.1506	0.1469
Max	0.4545	0.4333
Min	0.0	0.0

Table 2: Statistics on values of Hamming distance

7 Conclusion and Future Works

This work proposed a solution to efficiently anonymize user data extracted from analytics services. The proof of concept implementation relies on the new Android Virtualization techniques to perform the user data interception and focuses on Google Analytics libraries as the most complex anonymization scenario. The experimental campaign demonstrated that the results obtained from executing the anonymization algorithms on user data are satisfying because the overall utility level is maintained. Indeed, the utility level reached about 77%, regardless of the privacy parameters set, thereby proving that it is possible to grant a high level of privacy to the user without considerably affecting the utility of the data gathered and provided to the developer for their analysis activities.

Concerning the Android Virtualization techniques, the main challenge of using these technologies is that the open-source solutions at the state-of-the-art share several limitations regarding the supported Android SDK version, which is 26 at the time of writing. Using Android Virtualization techniques allows for overcoming the limitations of Android VPN technologies, which will be neglected in future Android releases.

The current implementation only supports version 21.0.0 of Google Firebase analytics. All the versions of Google Analytics libraries are obfuscated in different ways. Since the hooking process of the methodology is based on the method signatures defined in section 5.1, the change in the version of the analytics library could require a manual update of the current hooks defined for the library; we aim to define an automated methodology to extract the signatures of the target methods.

As a final consideration, the growing adoption of analytic libraries in current Android apps that keep profiling the user requires considering the privacy contribution when evaluating a risk profile of an app. So far, the risk analysis of mobile apps has focused on characteristics of the attack surface of an app (i.e., the number of vulnerabilities and their characterization).

Nonetheless, the kind and number of analytic libraries embedded in an app and the type of event profiled must be considered when calculating the risk. Novel risk evaluation methodologies should be proposed and the existing tools [46] updated accordingly. Also, the differences between malware and goodware in the distribution of analytic libraries should be investigated.

Acknowledgement

This work was partially supported by the Curiosity Driven grant "Security Assessment of Cross-domain Application Ecosystems" of the University of Genova funded by the EU-NGEU.

References

[1] BankMyCel, "How-many-phones-are-in-the-world?" https://www.bankmycell.com/blog/how-many-phones-are-in-the-world, 2022, online; accessed December 14, 2023.

[2] B. Wolford, "What is gdpr, the eu's new data protection law?" https://gdpr.eu/what-is-gdpr/, 2020, online; accessed December 14, 2023.

[3] D. Caputo, L. Verderame, A. Ranieri, A. Merlo, and L. Caviglione, "Fine-hearing google home: why silence will not protect your privacy." *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, vol. 11, no. 1, pp. 35–53, 2020.

[4] Statista, "Distribution of global time spent on mobile in 2020 and 2022, by category," https://www.statista.com/statistics/435324/share-app-time-category/, 2023, online; accessed December 14, 2023.

[5] Google, "Google analytics," https://analytics.google.com/analytics/web/, 2020, online; accessed December 14, 2023.

[6] Exodus, "Most frequent trackers - google play," https://reports.exodus-privacy.eu.org/en/trackers/stats/, 2023, online; accessed December 14, 2023.

[7] T. Chen, I. Ullah, M. A. Kaafar, and R. Boreli, "Information leakage through mobile analytics services," in *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications*, 2014, pp. 1–6.

[8] X. Zhang, X. Wang, R. Slavin, T. Breaux, and J. Niu, "How does misconfiguration of analytic services compromise mobile privacy?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1572–1583.

[9] "Il garante," https://www.garanteprivacy.it/home/autorita, 2022.

[10] C. Script, "Google analytics 4 and gdpr: Is ga4 gdpr compliant?" https://cookie-script.com/blog/google-analytics-4-and-gdpr, 2019.

[11] A. Razaghpanah, R. Nithyanand, N. Vallina-Rodriguez, S. Sundaresan, M. Allman, C. Kreibich, and P. Gill, "Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem," 2018.

[12] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: Trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '11. New York, NY, USA: Association for Computing Machinery, 2011.

[13] D. Caputo, L. Verderame, and A. Merlo, "Mobhide: App-level runtime data anonymization on mobile," in *International Conference on Applied Cryptography and Network Security*. Springer, 2020, pp. 490–507.

[14] D. Caputo, F. Pagano, G. Bottino, L. Verderame, and A. Merlo, "You can't always get what you want: Towards user-controlled privacy on android," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[15] Google, "Protecting your privacy online," https://privacysandbox.com/intl/en_us/, 2020, online; accessed December 14, 2023.

[16] Apple, "App tracking transparency," https://developer.apple.com/documentation/apptrackingtransparency, 2020, online; accessed December 14, 2023.

[17] A. Merlo, A. Ruggia, L. Sciolla, and L. Verderame, "You shall not repackage! demystifying anti-repackaging on android," *Computers & Security*, vol. 103, p. 102181, 2021.

[18] Firebase, "Firebase," https://firebase.google.com, 2022.

[19] AppBrain, "Firebase," https://www.appbrain.com/stats/libraries/details/firebase/firebase, Online; accessed December 14, 2023.

[20] G. A. Events, "Google analytics 4 events," https://support.google.com/analytics/answer/9322688?hl=en, 2022, online; accessed December 14, 2023.

[21] G. Analytics, "Google analytics 4 events," https://developers.google.com/analytics/devguides/collection/ga4/reference/events, 2022, online; accessed December 14, 2023.

[22] D. Team, "Droidplugin," 2020, accessed online: December 14, 2023. [Online]. Available: https://github.com/DroidPluginTeam/DroidPlugin

[23] J. L. N. T. C. Ltd., "Virtualapp," 2020, accessed online: December 14, 2023. [Online]. Available: https://github.com/asLody/VirtualApp

[24] Oracle, "Dynamic proxy classes," Oracle, 2021, accessed online: December 14, 2023. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html

[25] A. Armando, A. Merlo, and L. Verderame, "An empirical evaluation of the android security framework," in *Security and Privacy Protection in Information Processing Systems: 28th IFIP TC 11 International Conference, SEC 2013, Auckland, New Zealand, July 8-10, 2013. Proceedings 28*. Springer, 2013, pp. 176–189.

[26] G. Navarro-Arribas and V. Torra, "Information fusion in data privacy: A survey," *Information Fusion*, vol. 13, no. 4, pp. 235–244, 2012.

[27] S. d. C. di Vimercati, S. Foresti, G. Livraga, and P. Samarati, "Anonymization of statistical data," *IT-Information Technology*, vol. 53, no. 1, pp. 18–25, 2011.

[28] P. Samarati, "Protecting respondents identities in microdata release," *IEEE transactions on Knowledge and Data Engineering*, 2001.

[29] C. Dwork, "Differential privacy: A survey of results," *International conference on theory and applications of models of computation*, 2008.

[30] G. Cormode, S. Jha, T. Kulkarni, N. Li, D. Srivastava, and T. Wang, "Privacy at scale: Local differential privacy in practice," *Pro- ceedings of the 2018 International Conference on Management of Data*, 2018.

[31] M. Yang, L. Lyu, J. Zhao, T. Zhu, and K.-Y. Lam, "Local differential privacy and its applications: A comprehensive survey," *arXiv preprint arXiv*, 2020.

[32] H. Zhang, S. Latif, R. Bassily, and A. Rountev, "Privaid: Differentially-private event frequency analysis for google analytics in android apps," *Ohio State University, Columbus, Ohio, USA*, 2018.

[33] A. Developers, "Vpn," https://developer.android.com/guide/topics/connectivity/vpn, Accessed in December 14, 2023.

[34] G. Developers, "Protocol buffers," https://developers.google.com/protocol-buffers, Accessed in December 14, 2023.

[35] A. Merlo, A. Ruggia, L. Sciolla, and L. Verderame, "Armand: Anti-repackaging through multi-pattern anti-tampering based on native detection," *Pervasive and Mobile Computing*, vol. 76, p. 101443, 2021.

[36] GooglePlay, "Developer program policy," https://support.google.com/googleplay/android-developer/answer/12253906?hl=en, 2022, online; accessed December 14, 2023.

[37] skylot, "jadx," 2022, accessed online: December 14, 2023. [Online]. Available: https://github.com/skylot/jadx

[38] Android, "Build class," https://developer.android.com/reference/android/os/Build, 2022, online; accessed December 14, 2023.

[39] Java, "Java reflection," https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html, 2022, online; accessed December 14, 2023.

[40] Android, "Dexclassloader," https://developer.android.com/reference/dalvik/system/DexClassLoader, 2023, online; accessed December 14, 2023.

[41] Realm, "Realm database," https://realm.io/, Accessed in December 14, 2023.

[42] PAGalaxyLab, "yahfa," 2022, accessed online: December 14, 2023. [Online]. Available: https://github.com/PAGalaxyLab/YAHFA

[43] A. Romdhana, A. Merlo, M. Ceccato, and P. Tonella, "Deep reinforcement learning for black-box testing of android apps," *ACM Trans. Softw. Eng. Methodol.*, 2021.

[44] T. Van Erven and P. Harremos, "Rényi divergence and kullback-leibler divergence," *IEEE Transactions on Information Theory*, vol. 60, no. 7, pp. 3797–3820, 2014.

[45] A. Bookstein, V. A. Kulyukin, and T. Raita, “Generalized hamming distance,” *Information Retrieval*, vol. 5, pp. 353–375, 2002.

[46] A. Merlo and G. C. Georgiu, *RiskInDroid: Machine Learning-Based Risk Analysis on Android*. Springer International Publishing, 2017, pp. 538–552. [Online]. Available: https://doi.org/10.1007/978-3-319-58469-0_36