Computers and Electrical Engineering 121 (2025) 109882

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

MarvelHideDroid: Reliable on-the-fly data anonymization based on Android virtualization

Francesco Pagano [a], Luca Verderame [a], Enrico Russo [a], Alessio Merlo [b,*]

[a] *DIBRIS, University of Genova, Genoa, Italy*
[b] *CASD, School for Advanced Defense Studies, Rome, Italy*

ABSTRACT

Modern mobile applications harvest many user-generated events during execution using proper libraries called *analytic libraries*. The collection of such events allows the app developers to acquire helpful information to further improve the app. The same collected events are likewise an essential source of information for analytic library providers (e.g., Google and Meta) to understand users' preferences. However, the user is not involved in this process. To counteract this problem, some proposals arose from legal (e.g., General Data Protection Regulation (GDPR)) and research perspectives. Concerning the latter point, some research efforts led to the definition of solutions for the Android ecosystem that allow one to limit the gathering of such data before the analytic libraries collect it or give the user control of the process. To this aim, *HideDroid* was the first proposal to allow the user to define different privacy levels for each app installed on the device by leveraging k-anonymity and differential privacy techniques. Subsequently, *VirtualHideDroid* extended HideDroid by taking advantage of the same approach to virtualized Android environments, in which an application (plugin) can run within another application (container). In this scenario, VirtualHideDroid anonymizes user event data running as the container app. However, according to standard threat models regarding virtualized Android environments, assuming that the container app is fully trusted is too optimistic in real deployments.

For this reason, in this paper, we extend the work of the original VirtualHideDroid work by assuming that the same tool may be untrusted, i.e., controlled by an external attacker that has access to the container app, thereby having full access to the user data. To solve this problem, we define a new approach, named *MarvelHideDroid*, which gives reliable anonymization of event data in the Plugin app, even in the event of a malicious/compromised container. Moreover, and differently from VirtualHideDroid, *MarvelHideDroid* relies on LLM to automatically build up the generalizations required by k-anonymity, resulting in an anonymization strategy that is more reliable against modification in the data structure of the events captured by the analytic libraries. We empirically demonstrate the viability and reliability of the proposal by testing an implementation of *MarvelHideDroid* on a set of real Android apps in a virtualized environment.

1. Introduction and context

The exponential growth in mobile device adoption during the last decade led to flourishing research on several aspects of mobile security. The focus primarily concerned defining methodologies and tools to find and exploit application vulnerabilities, harden the Android OS, and detect proliferating mobile malware. However, in recent years, novel concerns about managing and exploiting user data in mobile apps have raised privacy-related issues. In particular, such concerns were exacerbated as the amount of personal data produced on mobile phones increased significantly, primarily due to the spread of social apps, as shown in [1].

To this end, while executing and interacting with the user, mobile applications generate a lot of personal data that is helpful to developers and firms such as Facebook and Google, providing a proper means for developers to monitor user interaction with their applications. Such means include *analytic libraries* that give a set of functionalities that developers can embed into the app they develop, distribute, and configure to recognize and gather specific events generated by the user in the app. This way, analytic libraries allow the developer to collect much data from each app instance installed on users' devices worldwide. This way, the

* Corresponding author.
E-mail address: alessio.merlo@unicasd.it (A. Merlo).

https://doi.org/10.1016/j.compeleceng.2024.109882
Received 15 April 2024; Received in revised form 24 September 2024; Accepted 12 November 2024
Available online 27 November 2024
0045-7906/© 2024 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

developer can monitor the app's execution to detect usage patterns and obtain valuable insights to improve future app versions. The most popular analytics library is Google Analytics [2], which is currently embedded in 58% of the Android apps analyzed in [3].

From a regulatory point of view, in 2018, the EU released the *General Data Protection Regulation* (GDPR) [4] to limit the uncontrolled gathering of personal data by placing obligations on all organizations that collect data related to user data in the European Union. In particular, GDPR establishes that a company must evaluate the risks of data breaches to avoid the need to add privacy-preserving solutions to existing information systems a posteriori. Moreover, 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.

From a research point of view, some works investigated privacy concerns related to analytics libraries. For example, Chen et al. [5] demonstrated that an external adversary could extract sensitive information about the user and the app by exploiting Google Mobile App Analytics and Flurry. Zhang et al. [6] highlighted some privacy problems related to a misconfiguration of analytic services. Furthermore, several works, such as [6], reported how analytical libraries share the same privileges and resources as the hosting application, allowing them to access and collect sensitive information about users and their behavior without proper privacy protection mechanisms. Meng et al. [7] analyze 158 third-party Android SDKs using taint analysis and Large Language Model (LLM) techniques. Their goal is to identify which sensitive user data these SDKs exfiltrate from users' smartphones. Although they employ taint analysis and LLMs to detect data exfiltration, they do not leverage the collected information to propose an anonymization methodology for these types of data. Lu et al. [8] analyze the most popular social-media SDKs and propose a novel methodology to modify the code of social-media SDKs within Android apps to secure the data exfiltration process using dedicated threads. Although this methodology is promising, it does not anonymize the extracted data but only ensures its secure exfiltration. However, the exfiltrated data are still sent in clear to the social-media providers.

Moreover, these libraries must comply with the GDPR for use 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].

Although the GDPR limits some unethical personal data collection procedures, it does not require the user to be actively involved in selecting the data she aims to share. Although this limitation is also relevant in several scenarios such as IoT [11], it is of the utmost importance in the mobile ecosystem, where the previously described protocol, based on analytic libraries, requires the active participation of the user in deciding which data to share to the app developers and the companies providing the analytic libraries.

To this end, some research efforts aim to protect user privacy on the mobile platform. For instance, the authors in [12] 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 [13] 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 [14,15], 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) valid data for assessment. The privacy concerns have also reached Google [16] and Apple [17], 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.

Nonetheless, state-of-the-art solutions have some limitations. First, they – except HideDroid – do not give the user, the data owner, any control over the collected data and the anonymization process. 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: the work in [12] follows an ''all or nothing approach'', supporting only the option to either fully accept or deny all the collection of personal data. At the same time, HideDroid [15] performs an app repackaging step [18] 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.

To overcome such limitations, the authors in [19] proposed an extension of HideDroid, called *VirtualHideDroid*, able to execute as a plugin app in a virtualized Android environment. A virtual app comprises two nested applications: the *container app* and the *plugin app*. The container app executes the plugin app by filtering and intercepting all OS requests of the plugin app. VirtualHideDroid executes as a container app, while the app whose data must be anonymized executes as a plugin app. In this way, each data sent by the analytic libraries to their corresponding backends is intercepted by VirtualHideDroid, anonymized on the fly, and then sent to the backends. This way, the app does not need to be modified and repackaged, and the protobuf issue is solved.

Novel Contribution. From a security point of view, the threat model of VirtualHideDroid shares the same limitations of all virtual apps in Android: total trust in the container app. As this intercepts and can modify all traffic from/to the plugin app, it is relatively common to assume that the same is fully trusted. However, since virtualized environments potentially support any container apps (i.e., not only those approved by Google), we argue that a reliable data anonymization solution for Android apps should be robust against untrusted anonymization container apps, as a potentially maliciously crafted and attacker-controlled version of VirtualHideDroid. In this case, the malicious anonymization container app could steal user data from apps. In this paper, we extend the threat model of VirtualHideDroid by assuming that the anonymization container app could be malicious. Then, we propose a novel solution called *MarvelHideDroid* that grants anonymity to plugin app data exported through analytic libraries in case of a malicious anonymizer container app. Moreover, we improved the reliability and scalability of the anonymization techniques adopted by VirtualHideDroid by leveraging LLMs to build up data generalizations automatically. This way, we make the anonymization

strategies robust against modification in the data scheme of analytic libraries. Finally, we empirically assess the reliability of *MarvelHideDroid* through experiments on real apps. We have released *MarvelHideDroid* as open-source software available on GitHub.[1]

Paper Structure. The paper is organized as follows: Section 2 introduces some background knowledge on analytic libraries, data anonymization, and virtualization technologies. Then, Section 3 discusses the threat model of virtualized Android apps; Section 4 describes the methodology based on *MarvelHideDroid*, while Section 5 provides more technical details on its prototype implementation. Section 6 discusses the experimental setup and the experimental results. Finally, Section 8 concludes the article with a summary 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 top-tier applications and features leads to fierce competition among mobile developers who strive to enhance 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 apps. Consequently, developers started integrating analytics libraries into their mobile apps to collect information about the user, the app's usage, and the device, gaining a deeper understanding of market demands. 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 and the integration of the libraries inside the mobile app.

From a technical point of view, the data are collected as *Events* or *User Properties*. The Events allow developers to track users' interaction on an app. They are composed of a name that identifies them and some parameters that give more information about the event. For instance, the *add_to_cart* event comprises parameters such as *currency*, *value*, and *items*. While analytic libraries generally provide a predefined set of standard events, they also support creating custom events. On the other hand, User Properties are attributes that can delineate various segments within the user base. For example, developers can establish properties like `favorite_food` to denote the user's preferred cuisine or `language` to identify the user's primary 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 [20]. Based on statistical insights from appBrain [21], Firebase analytics is the most widely used analytic service in the context of mobile apps. In particular, nearly 70% of the apps on the USA Play Store use Google Firebase Analytics, and close to 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. This library categorizes Events for mobile apps into three distinct types: automatically collected, recommended, and custom events [22]. Automatically collected events include actions such as app removal, *app_remove*, first open after installation, *first_open*, changing screen, *screen_view*, among others. In contrast, developers must manually trigger the other two types within the app's code. Firebase library facilitates this through a method called `logEvent()`, which requires the event name and a bundle containing relevant parameters as input.

The recommended events allow for measuring additional features and behavior and generating more valuable reports. Since these events require additional context to be meaningful, they are not sent automatically. Examples include *add_payment_info* [23], which users should use when submitting their payment information. Custom events enable developers to define event names and parameters based on specific contextual requirements.

Similar to Events, Google Analytics offers custom and automatic user properties. Developers can define custom user properties using the `setUserProperty()` method provided by Google Analytics libraries, which requires specifying a name-value pair, such as *favorite_food* and *pizza*. On the other hand, automatic user properties automatically transmit information regarding the app's usage, user data, and session details, including *first_open_time* and *session_id*. Given their similarity, we use the generic term *events* to refer to both Events and User Properties, only specifying differences when necessary.

[1] https://anonymous.4open.science/r/MarvelHideDroid-5BC4

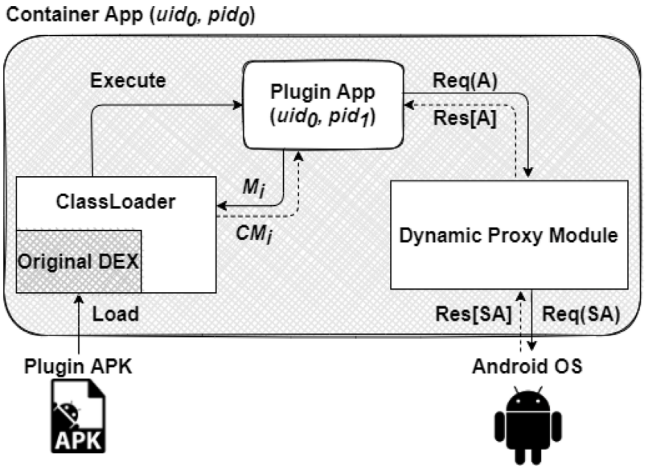

Fig. 1. The internal structure of an Android virtual environment.

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. In this environment, the user can execute several plugins independently from the underlying Android OS and other virtual environments. DroidPlugin [24] and VirtualApp [25] are the two most well-known frameworks that support the generation of virtual environments on Android and share 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's components, hiding the app from the Android OS.

The virtualization technology relies on the *Dynamic Code Loading* (DCL) and *Java dynamic proxy* [26]. 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 us to bypass the 64K reference limit imposed by the `.dex` format. On the other hand, the Java dynamic proxy allows creation of a wrapper object to intercept all method calls towards a specific object instance, eventually adding some functionality.

Fig. 1 shows the internal architecture of an app that generates 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.

ClassLoader loads Java code (`.apk` or `.dex` files) in the Android system. 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 the plugin's classes and methods (i.e., M_i) by returning the appropriate code (CM_i).

Unlike the *User Interface* (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 [27]: 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 point of view, this module relies on Java dynamic proxy, creating a proxy object (known as *hook*) for each Android manager. In the example of Fig. 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 to execute on the actual device. Regarding permissions, the container requires all existing Android permissions to support all possible plugins.

2.3. Static code instrumentation of Android apps

Static code instrumentation of Android apps involves analyzing and manipulating an application's bytecode to gain insights into its behavior or to modify its functionality.

A leading tool utilized for this purpose is Soot [28], along with its successor SootUp [29]. Soot is a sophisticated Java optimization framework designed for static analysis and manipulation of Android `.dex` and `.apk` files. Notably, the framework enables comprehensive program execution flow extraction through representations such as call graphs and control-flow graphs [30]. Moreover, Soot facilitates bytecode manipulation and code injection using the Jimple intermediate representation.

The static code instrumentation of Android apps has been used in numerous studies. For example, authors in [31] used static code instrumentation to analyze security vulnerabilities in Android applications. In contrast, the work in [32] used code instrumentation to calculate code coverage during dynamic analysis of Android applications.

2.4. Data anonymization techniques

In data privacy, the set of attributes in a microdata set [33,34] 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 indirectly identify a user. 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, such as health diseases, salaries, and eating habits.

The events generated by analytics libraries are multidimensional data. 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 and 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 data processing type, the privacy level, and the 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 [35]. 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 $f_h : h = 0, \dots, n-1$, such that:

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

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 (DP) involves altering the original distribution of a set of interdependent data using a perturbation function [36]. 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 [36]. On the contrary, the local model assumes all external entities and communication channels as untrusted [37,38]. 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, the user is 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 $(e_1, e_2, \dots, e_n) \in$ D in a different sequence of events $(z_1, z_2, \dots, z_n) \in$ D through the application of a perturbation function R : D→ D to each event. This function is commonly a probability distribution, defined a priori: $z_i = R(e_i)$.

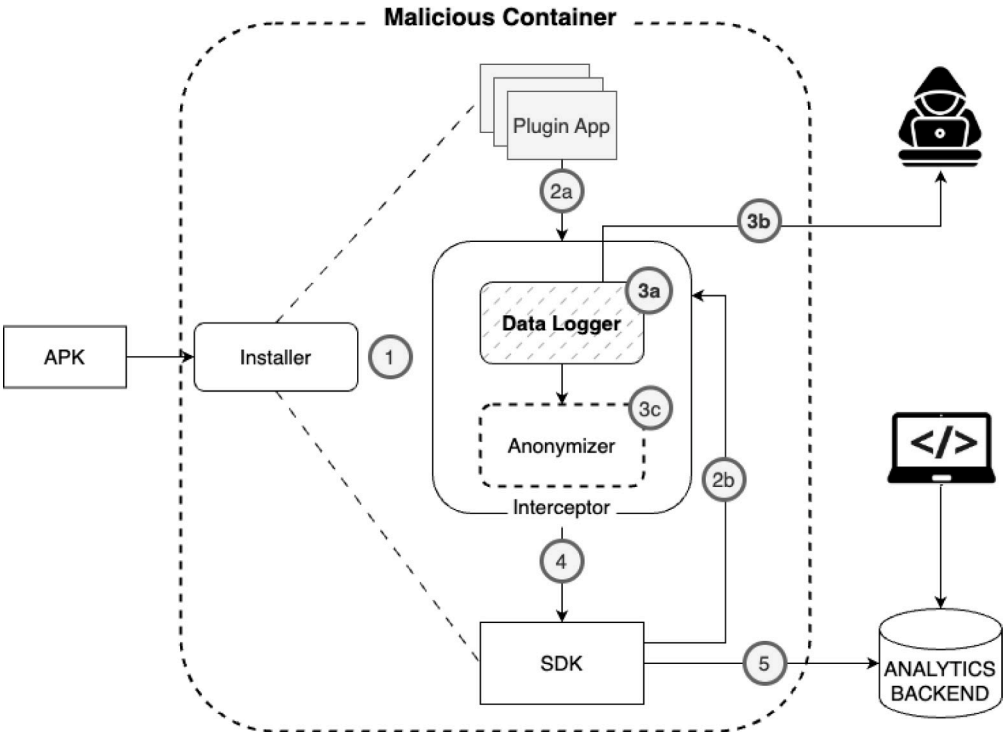

Fig. 2. Threat model of a malicious anonymizing container app.

3. Threat model

In the privacy-related scenario discussed, the attacker aims to steal users' sensitive data from the analytics data generated by the victim's apps. The victim is a general Android user, daily enjoying her mobile device and regularly downloading apps from any app market (e.g., Google Play Store and Amazon Appstore) and using Android virtualization solutions – such as the VirtualHideDroid container [19] – to control the privacy of the data extracted by analytics libraries from her apps.

We argue that previous assumptions are too optimistic. A more reasonable assumption in actual deployments is that the container app is untrusted. In particular, we assume that the attacker could be able to perform a virtualization-based repacking attack [39] to create and redistribute in app stores a malicious version of the VirtualHideDroid container that logs the analytics data generated by the plugin apps before applying the anonymization procedure.

The scenario is illustrated in Fig. 2. The malicious container application has been extended with a *Data Logger* function embedded in its functionality that intercepts data to anonymize. To simplify, we refer to this component (a module, a function, or an API) as Interceptor. The malicious container supports installing Android APK files as plugin apps (**Step 1**). At runtime, after the Interceptor is triggered by the hook on the APIs of the analytic library (**Step 2a** in the case of custom events, otherwise **Step 2b** in the case of automatic events), the *Data Logger* collects the generated events and user-specific properties (**Step 3a**) and forwards them to the attacker without any anonymization (**Step 3b**). Finally, the malicious container continues with the intended workflow of anonymization, and forwards anonymized data to the analytics backend (i.e., **Steps 3c**, **4**, and **5**) to maintain the expected behavior from the user's standpoint.

4. Anonymization methodology

In this section, we describe the methodology for adding an anonymization layer of the user's sensitive data directly in the plugin app to cope with the scenario of untrusted container apps described in the previous threat model. In summary, the methodology consists of injecting the anonymization module directly inside the plugin app and instrumenting all the method invocations contained in the plugin app before sending the user's sensitive data to the analytic libraries.

Fig. 3 shows a high-level overview of the process. First, we build a list of all the analytics library APIs that manage data through events and user properties. To do so, we rely on an offline analysis of the analytics libraries (**Step a**) to retrieve and store the list of APIs (**Step b**). It is worth noticing that such a phase is done on a per-library basis and must be updated only in the case of newer releases on the same library or to support other libraries than Google Play analytics.

Then, we apply the anonymization layer on each plugin app using a four-step approach. First, the methodology extracts the app's *Control Flow Graph* (CFG) related to the main app code and the analytics library code (**Step 1**). The resulting CFG evaluates

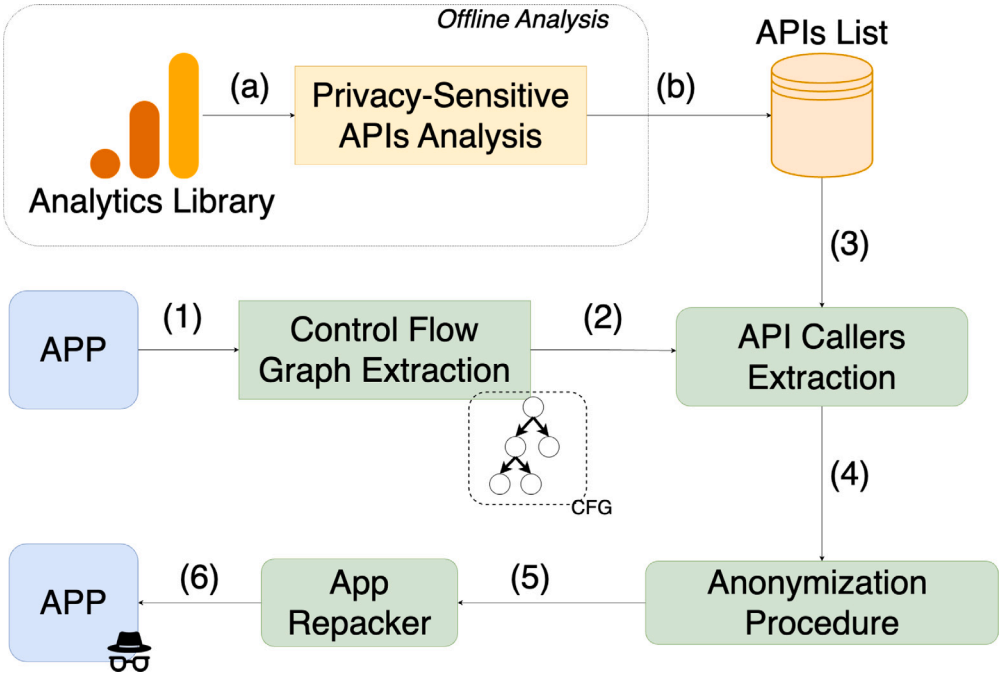

Fig. 3. Overview of the anonymization process of plugin apps.

all the method calls invoking the analytics libraries' APIs. To do so, the `APIs Caller Extraction` phase analyzes the CFG (**Step 2**), fetches the list of signatures of the methods of the analytics library that receive the user's sensitive data (**Step 3**), and retrieves from the CFG of the app the list of methods that needs to be modified (**Step 4**). The `Anonymization Procedure` instruments the app binary by (i) adding an anonymization module that implements differential privacy and data generalization as described in Section 2.4, and (ii) modifying the body of all the methods identified in Step 4 to invoke the anonymization process before invocation of the analytics API. The local and runtime execution of the data anonymization process prevents the container module from hooking inside the API invocation and, thus, accessing the users' data. Finally, the modified app is built and signed in the `App Repacker` phase (**Step 5**) to generate a valid plugin app installed inside a virtual container and executed by the container app (**Step 6**).

The rest of this section details the anonymization strategies exploited by the anonymization module and the process related to the app modification.

4.1. Notes on the anonymization procedure

The anonymization procedure involves a set of compiled classes that adapt two different techniques to the analytics libraries scenario, i.e., *generalization* and *differential privacy*. The anonymization procedure exploits a buffer to store every anonymized event sent to the Analytics backend. For each intercepted event, it applies generalization to the data contained in the event, putting the results into a buffer that includes every anonymized event sent to the analytic library backends. The anonymization is not applied if the buffer does not reach a predefined minimum length. The last step of the anonymization procedure involves triggering 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.

4.1.1. Generalization

The anonymization procedure 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 three 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.
- 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 process it.

- if the attribute is the name of an **event**, the generalization leverages an LLM [40] to extract a more generalized version of the event name without losing the original name's semantics. It is worth noticing that this kind of generalization is just applied to the event name and not to the other types of attributes, as the event name is usually the only attribute that always appears in every analytics request.

4.1.2. 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 backend (`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 initialized the `sentEventsSize` variable with the size of `sentEvents`, and it initialized the `anonymizedEvents` list. The `anonymizedEvents` is a temporary list where the intercepted events 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 a new event could be generated. Also, in line 4, the algorithm initializes the Pr_{rep} variable sampling from a uniform probability distribution. This variable contains the probability that the intercepted event could be substituted with another one taken from the buffer.
- **Less than `minNumberOfRequest` inside the event buffer**: in this case, the anonymization algorithm checks if the event buffer contains more than `minNumberOfRequest` events (line 5). If so, it cannot apply differential privacy and generalizes the intercepted event (see Section 4.1.1) before returning it (lines 7–8).
- **Both Pr_{inj} and Pr_{rep} higher than the `threshold`**: in this case, the algorithm applies differential privacy as the number of events inside the queue is more elevated than `minNumberOfRequest`. In line 10, the algorithm checks if both the Pr_{inj} and Pr_{rep} values are higher than the threshold. If so, it 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 algorithm 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 algorithm 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 algorithm executes the *injectEvent* action (line 22). It generalizes the intercepted event (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). Then, it returns both events (line 25).
- Pr_{rep} higher than the `threshold`: here, the algorithm performs the *replaceEvent* action (line 27): it extracts a generalized event from the event queue (line 28), adds the event to the temporary list (line 29), and returns the same instead of the intercepted event (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 algorithm generalizes the intercepted event (line 32) and returns it (line 34).

5. Methodology implementation

We implemented the methodology in a prototype tool called ***MarvelHideDroid*** for the Android OS that anonymizes the traffic generated by analytic libraries directly inside the app's code, preventing the stealing of sensitive information from a malicious container app, as described in Section 3. To assess the reliability 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.

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 *MarvelHideDroid*, underlying the technique used to instrument the app's code statically, and the adopted anonymization algorithms.

5.1. Anatomy of google analytics

To understand how the Google Analytics library collects the user's data and information about the device and creates the events, we performed a manual static analysis task leveraging the *jadx* [41] tool. For the activity, we considered the version 21 of the library, downloaded from the mavern official repository.[2]

The main goal of this analysis is to identify the privacy-sensitive API method(s) that process automatic and custom events. It is worth noting that this phase directly impacts the methodology's overhead because it determines, for instance, the number of instrumented methods. For example, injecting the anonymization procedure into a recursive method would trigger the procedure each time the function is invoked, resulting in a significant resource overhead at runtime.

[2] https://mvnrepository.com/artifact/com.google.firebase/firebase-analytics/21.0.0

Listing 1 Data Anonymization Pipeline.

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

Initialize $sentEventsSize \leftarrow sentEvents.size()$
Initialize $anonymizedEvents \leftarrow list()$
$Pr_{inj} \leftarrow$ rand()
$Pr_{rep} \leftarrow$ rand()
if sentEventsSize < minNumberOfRequests **then**
    $newGeneralizeEvent \leftarrow$ generateNewGeneralizeEvent($interceptedEvent.parameters$, $anonymizationPercentage$)
    $anonymizedEvents$.add($NewGeneralizeEvent$)
    return $anonymizedEvents$
else
    if $Pr_{rep} > threshold$ AND $Pr_{inj} > threshold$ **then**
        $randomNum \leftarrow$ rand()
        $fromRealmAnonymizedEvent \leftarrow$ injectEventFromRealm($sentEvents$,$sentEventsSize$)
        if $randomNum < 1/2$ **then**
            $newGeneralizeEvent \leftarrow$ generateNewGeneralizeEvent($interceptedEvent.parameters$, $anonymizationPercentage$)
            $anonymizedEvents$.add($newGeneralizeEvent$)
        end if
        $anonymizedEvents$.add($fromRealmAnonymizedEvent$)
        return $anonymizedEvents$
    end if
    if $Pr_{inj} > threshold$ **then**
        $newGeneralizeEvent \leftarrow$ generateNewGeneralizeEvent($interceptedEvent.parameters$, $anonymizationPercentage$)
        $anonymizedEvents$.add($newGeneralizeEvent$)
        $fromRealmAnonymizedEvent \leftarrow$ injectEventFromRealm($sentEvents$,$sentEventsSize$)
        $anonymizedEvents$.add($fromRealmAnonymizedEvent$)
        return $anonymizedEvents$
    end if
    if $Pr_{rep} > threshold$ **then**
        $fromRealmAnonymizedEvent \leftarrow$ injectEventFromRealm($sentEvents$,$sentEventsSize$)
        $anonymizedEvents$.add($fromRealmAnonymizedEvent$)
        return $anonymizedEvents$
    end if
    $newGeneralizeEvent \leftarrow$ generateNewGeneralizeEvent($interceptedEvent.parameters$, $anonymizationPercentage$)
    $anonymizedEvents$.add($NewGeneralizeEvent$)
    return $newAnonymizedEvents$
end if

During the activity, we identified three different API methods related to the management of analytics events, namely, the `logEvent` method (located in the `com.google.firebase.analytics.FirebaseAnalytics` class), the `<init>` method of the class `com.google.android.gms.measurement.internal`, and the `logEvent` method belonging to the `com.google.firebase.analytics.connector.AnalyticsConnector` Java interface. All the methods accept a set of parameters that include the name of the collected event (of type `java.lang.String`), whose value could either be custom or selected from a list of predefined events provided by the library and a `android.os.Bundle`,[3] which contains all the information related to the event.

We similarly identified the `setUserProperty` method belonging to the `com.google.firebase.analytics.FirebaseAnalytics` class, responsible for sending User Properties information to the analytics backend. This method is the standard public method that is exposed to the developer from the Google Analytics library to dump this information. It receives the name of the property as a parameter (of type `java.lang.String`), whose value could be decided by the developer or selected from a list of predefined user properties provided by the library, and a `java.lang.String`, which contain the property extracted from the user and linked to the corresponding property name decided from the developer. Similarly, we identified a constructor method of the package `com.google.android.gms.measurement.internal` responsible for building User Properties.

5.2. Marvelhidedroid

MarvelHideDroid is a standalone tool implemented in Java 8 that uses the Soot framework [28] to instrument Android application packages.

First, *MarvelHideDroid* leverage Soot to compute the app's control flow graph to identify the position of the API methods identified in Section 5.1 inside the app's code and all the methods that directly call them (i.e., the APIs caller list).

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

For each method in the list, *MarvelHideDroid* runs a custom Soot Transformer that converts the compiled body of the function into a Jimple representation. Then, the tool generates the Jimple representation of the anonymization code and injects it inside the caller's body before the function invokes the analytics library's hooked API.

The anonymization code takes the original parameter values and anonymizes them according to the `Differential Privacy` and `Generalization` algorithms described above. The `Differential Privacy` and `Generalization` algorithms are implemented in Java code. In particular, the `Differential Privacy` algorithm's implementation leverages an SQLite database to store all the events previously generated from the app to be potentially used in case the generated event has to be replaced. Concerning the `Generalization` algorithm's implementation, *MarvelHideDroid* leverages the ChatGPT LLM [42] API to generalize the event's name. In detail, the Generalization module calls the ChatGPT API[4] through a network request sending the prompt alongside the event name. In detail, the generalization procedure asks the LLM to generalize the event's name with just one English word.

After injecting the anonymization code to anonymize the parameter values of the methods we identified, *MarvelHideDroid* converts the Jimple representation of the caller methods to their bytecode representation to be packed inside the statically instrumented app. Finally, *MarvelHideDroid* builds and signs the `.apk`.

6. Results

We tested the feasibility and the effectiveness of our methodology enforced by *MarvelHideDroid*. 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 campaign.

6.1. Experimental setup

In the current implementation, *MarvelHideDroid* targets the latest version of Google Analytics library (version 21.0.0).

To test the prototype, we collected a dataset of the first 80 top apps on Google Play in February 2024 that use Google Analytics library version 21.0.0 and are compatible with the Android Virtualization environment. The search for apps with such a specific prerequisite involved the execution of a pre-processing script where apps were automatically downloaded from the store, unpacked, scrutinized whether they contained the signature methods identified in Section 5.1, and installed on a clean version of Android Virtual App [25].

6.2. Testing pipeline

The pipeline aims to evaluate the effectiveness of *MarvelHideDroid* regarding the anonymization impact on the data utility. It also seeks to assess the robustness of *MarvelHideDroid* against the proposed threat model. This requires the app to be statically instrumented through *MarvelHideDroid* before being executed from a container app implemented through the Virtual App framework [25]. We leverage the VirtualHideDroid [19] tool as our container app since it implements anonymization strategies to anonymize user data. After instrumenting each app with *MarvelHideDroid*, we upload them to VirtualHideDroid. We executed each instrumented app of the dataset through VirtualHideDroid, logging the anonymization process's results (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 container app is installed using the Android Debug Bridge (ADB) tool.
2. **Plugin App installation.** For each app of the dataset, ARES++ installs it as a plugin inside the container. To do so, ARES++ pushes the plugin app into the SD card and triggers the installation procedure of the VirtualHideDroid container, thereby giving it the permission to access the external storage[5] thanks to ADB.
3. **Plugin App assessment.** ARES++ executes each plugin app for 10 min, spawning a separate process to monitor the system logcat.

For the pipeline, we used a Mac OS machine with 64 GB of RAM and an Intel i9 processor for the instrumentation phase and an Emulator of a Pixel 8 Pro with four cores and 1536 MB of RAM with the Google Play services to evaluate the runtime behavior of the plugin apps instrumented with *MarvelHideDroid*.

[4] https://api.openai.com/v1/chat/completions

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

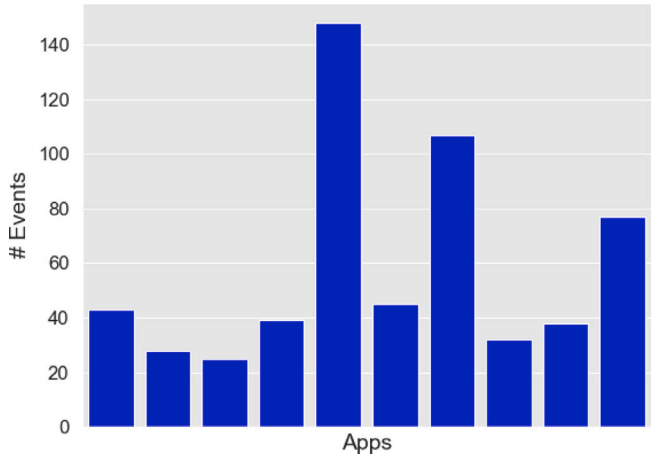

Fig. 4. Number of events generated from the first 10 top apps in the dataset.

6.3. Experimental results

The experimental campaign allowed the evaluation of the impact of the anonymization of *MarvelHideDroid* 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. We executed the testing pipeline with the `threshold` value at 60%.

The experimental activities allowed the logging of 774 events collected during the stimulation of the plugin apps of the dataset. Fig. 4 details the number of events generated from each app during the testing campaign.

To determine the impact of the anonymization process, we computed several statistical metrics to evaluate the difference between the original and anonymized events distributions and the content of the anonymized events.

The first measure is $KL_{divergence}$ (Kullback–Leibler divergence) [44] which evaluates the difference between two PDFs, i.e., probability distribution functions, $P(x)$ and $Q(x)$. The $KL_{divergence}$ is computed with the following formula:

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

In our case, $P(x)$ represents the probability that an event of type x will occur in the anonymized event list, while $Q(x)$ refers to the original list of events. The closer a KL divergence value is to zero, the more $P(x)$ and $Q(x)$ are similar.

Then, we focused on evaluating whether *MarvelHideDroid* preserves data utility by maintaining statistical similarity between the original and anonymized events distributions. First, we applied the Shapiro–Wilk test to assess the normality of the distributions. If the distributions were Gaussian, we performed the ANOVA test and calculated Cohen's d effect size, classifying it as Negligible (N), Small (S), Medium (M), or Large (L) using the standard thresholds of 0.2, 0.5, and 0.8, respectively. For non-Gaussian distributions, instead, we used the Wilcoxon signed-rank non-parametric test and computed the Vargha-Delaney effect size, with thresholds (applied to $2 \cdot |\ eff\_size\ - 0.5|$) of 0.147, 0.33, and 0.474, using the same classification labels as those for the ANOVA test.

Finally, we computed the *Edit distance* (or Levenshtein distance) [45], which can be used to evaluate the content of the generated events that are anonymized using the generalization procedure explained in Section 4.1.1. In particular, this metric evaluates the minimum number of insertion, deletion, and substitution operations needed to make a single parameter of two distinct events equal. In this scenario, a high value of Edit distance means that the parameter and its anonymized version are substantially different.

Fig. 5 shows the KL-Divergence values based on the event distributions and their anonymized versions for each app in the testing dataset. Table 1 shows some statistics related to the KL-Divergence metric of the event's distributions generated from the apps. In detail, the value of the KL-Divergence on the tested apps is 0.30 on average, which means that the original event distributions and their respective anonymized versions are different, in agreement with the `Differential Privacy`'s threshold value we initialized for the testing campaign. Also, the standard deviation value of 0.28 ensures that the anonymization task is practical regardless of the app under test.

The Wilcoxon-ANOVA tests revealed that 12 out of 80 apps exhibited a statistically significant difference. In detail, Table 2 shows the number of statistically significant original-anonymized event pairs per app, categorized by their effect-size level. These results indicate that compared to the original distributions, the number of apps with altered behavior in the anonymized event distributions is relatively low. This finding is consistent with the results obtained using the KL-Divergence metric.

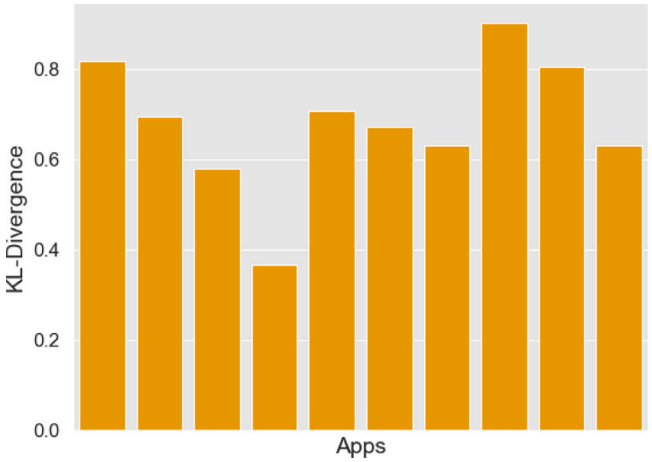

Fig. 5. KL-Divergence for the first 10 top apps in the dataset.

Table 1
Statistics on values of KL-Divergence.

Measure	Value
Mean	0.30
Median	0.20
St. dev.	0.28
Max	0.9
Min	0.0

Table 2
Number of apps where the original event sequence differs from the corresponding anonymized sequence. Effect sizes are reported only when statistically significant (N = Negligible; S = Small; M = Medium; L = Large).

Measure	N	S	M	L
Effect Size	0	3	6	3

Table 3
Statistics on values of Edit Distance.

Measure	Value
Mean	3.22
Median	3.62
St. dev.	1.4
Max	5.0625
Min	0.88

Fig. 6 shows the Edit Distance values between the user's data and their respective anonymized versions. For our experimental activity, we computed the *Edit distance* of all parameters in each logged event by pairing the original and anonymized version and computing the Edit distance value. It is worth noticing that we excluded from this computation the events that are replaced by the `Differential Privacy` procedure since we may not find a correspondence between the parameter in the original event and the substituted one. After obtaining the edit distances between all the pairs of parameters, we computed the average value associated with the entire plugin app.

Table 3 summarizes the results of the Edit distance values on the apps of the dataset. The table shows that the average Edit Distance across the app's dataset is 3.22, which means that an attacker needs a minimum of four operations (among insertions, replacement, or deletion) to transform the anonymized parameter into the original one. The Edit distance's standard deviation across the dataset is equal to 1.4, which means the variability of the Edit distances across the apps is low. It implies that the `Generalization` procedure injected by *MarvelHideDroid* is effective regardless of the app under analysis.

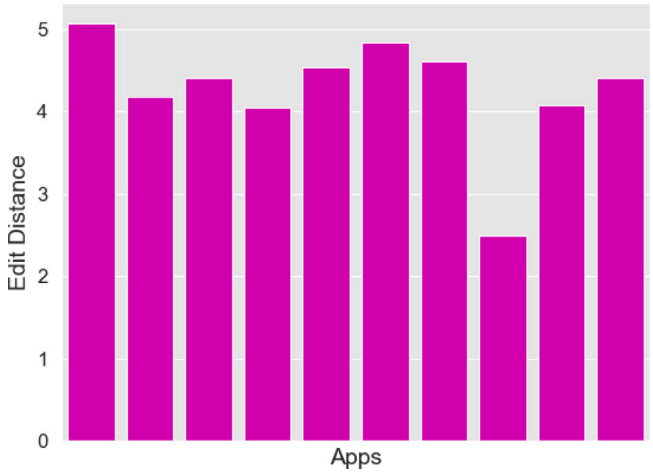

Fig. 6. Edit distance for the first 10 top apps in the dataset.

Table 4
Original and generalized event names.

Original event	Generalized event
screen_view	Display
login	Access
app_open	Opening
search	lookup
add_to_cart	Shopping
view_item	Browse
purchase	Buy
level_start	beginning
add_payment_info	Transaction
share	Exchange

Finally, we measured the time overhead the anonymization process imposes on each app's execution. The experimental results show the anonymization procedure imposes an average overhead of 0.92 s for each event generated from the apps, which is reasonable considering the execution time required from the Differential Privacy algorithm and the execution of a network request to get the generalized version of the event's name that labels each event.

7. Discussion on LLM generalization

We exploited LLM technologies to generalize the names of the events generated from the target app. In this study, we chose to study the application of an LLM-based generalization only on the event's name parameter because it is the only parameter present in every analytic request. Moreover, we also chose to use an LLM-based anonymization approach only on the event's name parameter because most of the events generated from the apps share a similar semantic, which means we were able to tune the prompt to be given to the LLM model we used for event names generalization. Using LLM-based generalization on the event's name demonstrated encouraging results regarding the original semantic preservation of the event's names. Table 4 shows the mapping between the most generated event names and their anonymized version.

The results we achieved during the testing campaign's execution highlight that the LLM-based generalization approach can reasonably anonymize the event's names. However, extending the study to support other kinds of data in the anonymization process is an exciting challenge outside this paper's scope.

8. Conclusion and future works

In this paper, we discussed a novel methodology for the on-the-fly anonymization of user data captured by an analytics library in Android virtualized environments. More specifically, we extended VirtualHideDroid [19] to deal with attackers able to successfully tweak the VirtualHideDroid tool, run as a container app, and steal plain user data. The proposed solution allows the plugin app running inside VirtualHideDroid to apply some form of anonymization to make data robust against theft from a malicious VirtualHideDroid instance. We implemented the proposed approach in a tool called *MarvelHideDroid*, which we plan to make fully available to the research community soon. The experimental campaign with *MarvelHideDroid* suggests the approach is viable and reliable. We also explored using LLM to support specific tasks in the anonymization procedure (e.g., the generalization of event

names). The preliminary results encourage using such technologies to overcome the limitations of existing approaches, e.g., the need to define data generalization hierarchies for multidimensional data manually.

Future extensions of the work include testing *MarvelHideDroid* on a more comprehensive set of real-world apps and more analytic libraries to assess the reliability of the selected LLM. Indeed, the results obtained in the testing campaign demonstrate that the proposed methodology could be used to anonymize the user data extracted from the Google Analytics library. Still, the approach could be further extended to other analytics libraries. The extension of the approach to other analytics libraries requires the analysis of the library artifact and identifying the procedures that extract the user's sensitive data from the device. It could be interesting to extend this research work with the automation of the sensitive procedure discovery phase to avoid the manual reversing phase of the library. Considering other LLM alternatives is an exciting extension that would make the approach scalable and adaptable in the wild to different analytics and change the syntax and semantics of the event captured during app execution.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] Statista. Distribution of global time spent on mobile in 2020 and 2022, by category. 2023, Online; https://www.statista.com/statistics/435324/share-app-time-category/. [Accessed 24 September 2024].
[2] Google. Google analytics. 2020, Online; https://analytics.google.com/analytics/web/. [Accessed 24 September 2024].
[3] Exodus. Most frequent trackers - google play. 2023, Online; https://reports.exodus-privacy.eu.org/en/trackers/stats/. [Accessed 24 September 2024].
[4] Wolford B. What is GDPR, the eu's new data protection law?. 2020, Online; https://gdpr.eu/what-is-gdpr/. [Accessed 24 September 2024].
[5] Chen T, Ullah I, Kaafar MA, Boreli R. Information leakage through mobile analytics services. In: Proceedings of the 15th workshop on mobile computing systems and applications. 2014, p. 1–6.
[6] Zhang X, Wang X, Slavin R, Breaux T, Niu J. How does misconfiguration of analytic services compromise mobile privacy? In: Proceedings of the ACM/IEEE 42nd international conference on software engineering. 2020, p. 1572–83.
[7] Meng MH, Yan C, Hao Y, Zhang Q, Wang Z, Wang K, et al. A large-scale privacy assessment of android third-party SDKs. 2024, arXiv preprint arXiv:2409.10411.
[8] Lu H, Liu Y, Liao X, Xing L. Towards {Privacy – Preserving}{Social – Media}{SDKs} on android. In: 33rd USeNIX security symposium. 2024, p. 647–64.
[9] Il garante. 2022, https://www.garanteprivacy.it/home/autorita.
[10] CookieScript. Google analytics 4 and GDPR: Is GA4 GDPR compliant?. 2019, https://cookie-script.com/blog/google-analytics-4-and-gdpr.
[11] Caputo D, Verderame L, Ranieri A, Merlo A, Caviglione L. Fine-hearing google home: why silence will not protect your privacy. J Wirel Mob Netw Ubiquitous Comput Dependable Appl 2020;11(1):35–53.
[12] Razaghpanah A, Nithyanand R, Vallina-Rodriguez N, Sundaresan S, Allman M, Kreibich C, et al. Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem. 2018.
[13] Beresford AR, Rice A, Skehin N, Sohan R. MockDroid: Trading privacy for application functionality on smartphones. In: Proceedings of the 12th workshop on mobile computing systems and applications. New York, NY, USA: Association for Computing Machinery; 2011.
[14] Caputo D, Verderame L, Merlo A. Mobhide: App-level runtime data anonymization on mobile. In: International conference on applied cryptography and network security. Springer; 2020, p. 490–507.
[15] Caputo D, Pagano F, Bottino G, Verderame L, Merlo A. You can't always get what you want: Towards user-controlled privacy on android. IEEE Trans Dependable Secure Comput 2022.
[16] Google. Protecting your privacy online. 2020, Online; https://privacysandbox.com/intl/en_us/. [Accessed 24 September 2024].
[17] Apple. App tracking transparency. 2020, Online; https://developer.apple.com/documentation/apptrackingtransparency. [Accessed 24 September 2024].
[18] Merlo A, Ruggia A, Sciolla L, Verderame L. You shall not repackage! demystifying anti-repackaging on android. Comput Secur 2021;103:102181.
[19] Pagano F, Ruggia A, Verderame L, Merlo A. VirtualHideDroid: User data anonymization through virtualization techniques. In: Proc. of the 7th international conference on mobile internet security. 2023.
[20] Firebase. Firebase. 2022, https://firebase.google.com.
[21] AppBrain. Firebase. 2024, Online; https://www.appbrain.com/stats/libraries/details/firebase/firebase. [Accessed 24 September 2024].
[22] Events GA. Google analytics 4 events. 2022, Online; https://support.google.com/analytics/answer/9322688?hl=en. [Accessed 24 September 2024].
[23] Analytics G. Google analytics 4 events. 2022, Online; https://developers.google.com/analytics/devguides/collection/ga4/reference/events. [Accessed 24 September 2024].
[24] Team D. DroidPlugin. 2020, [Online]. Available: https://github.com/DroidPluginTeam/DroidPlugin. [Accessed online 24 September 2024].
[25] Ltd. JLNTC. VirtualApp. 2020, [Online]. Available: https://github.com/asLody/VirtualApp. [Accessed online 24 September 2024].
[26] Oracle. Dynamic proxy classes. Oracle; 2021, [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html. [Accessed online 24 September 2024].
[27] Armando A, Merlo A, Verderame L. 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, p. 176–89.
[28] Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot: A java bytecode optimization framework. In: CASCON first decade high impact papers. 2010, p. 214–24.
[29] Karakaya K, Schott S, Klauke J, Bodden E, Schmidt M, Luo L, et al. SootUp: A Redesign of the soot static analysis framework.
[30] Allen FE. Control flow analysis. ACM Sigplan Notices 1970;5(7):1–19.
[31] Liu J, Wu T, Deng X, Yan J, Zhang J. InsDal: A safe and extensible instrumentation tool on dalvik byte-code for android applications. In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering. 2017.

[32] Romdhana A, Ceccato M, Georgiu GC, Merlo A, Tonella P. Cosmo: Code coverage made easier for android. In: 2021 14th IEEE conference on software testing, verification and validation. IEEE; 2021, p. 417–23.
[33] Navarro-Arribas G, Torra V. Information fusion in data privacy: A survey. Inf Fusion 2012;13(4):235–44.
[34] di Vimercati Sd, Foresti S, Livraga G, Samarati P. Anonymization of statistical data. IT-Inf Technol 2011;53(1):18–25.
[35] Samarati P. Protecting respondents identities in microdata release. IEEE Trans Knowl Data Eng 2001.
[36] Dwork C. Differential privacy: A survey of results. In: International conference on theory and applications of models of computation. 2008.
[37] Cormode G, Jha S, Kulkarni T, Li N, Srivastava D, Wang T. Privacy at scale: Local differential privacy in practice. In: Proceedings of the 2018 international conference on management of data. 2018.
[38] Yang M, Lyu L, Zhao J, Zhu T, Lam K-Y. Local differential privacy and its applications: A comprehensive survey. 2020, arXiv preprint arXiv.
[39] Ruggia A, Losiouk E, Verderame L, Conti M, Merlo A. Repack me if you can: An anti-repackaging solution based on android virtualization. In: Annual computer security applications conference. ACSAC, Association for Computing Machinery; 2021, http://dx.doi.org/10.1145/3485832.3488021.
[40] Chang Y, Wang X, Wang J, Wu Y, Yang L, Zhu K, et al. A survey on evaluation of large language models. ACM Trans Intell Syst Technol 2023.
[41] skylot. Jadx. 2022, [Online]. Available: https://github.com/skylot/jadx. [Accessed online 24 September 2024].
[42] OpenAI. ChatGPT. 2024, https://chat.openai.com/. [Accessed in 24 September 2024].
[43] Romdhana A, Merlo A, Ceccato M, Tonella P. Deep reinforcement learning for black-box testing of android apps. ACM Trans Softw Eng Methodol 2021.
[44] Van Erven T, Harremos P. Rényi divergence and Kullback-Leibler divergence. IEEE Trans Inform Theory 2014;60(7):3797–820.
[45] Ristad E, Yianilos P. Learning string-edit distance. IEEE Trans Pattern Anal Mach Intell 1998;20(5):522–32.