# ANDetect: A Third-party Ad Network Libraries Detection Framework for Android Applications

Xinyu Liu[1,2,†], Ze Jin[1], Jiaxi Liu[1,2], Wei Liu[1,2], Xiaoxi Wang[1], Qixu Liu[1,2,*,*†]
[1]Institute of Information Engineering, Chinese Academy of Sciences,
[2]School of Cyber Security, University of Chinese Academy of Sciences

## ABSTRACT

Third-party advertising libraries, which furnish mobile applications with ads, offer a revenue stream for Android application developers. However, the loaded ads potentially expose application users to privacy infringements and security threats. For instance, tracking scripts embedded in third-party ads monitor user behavior and can entice users into downloading malicious files. Therefore, the detection of advertising libraries in mobile applications is crucial for mobile security protection and serves as the foundation for preventing third-party ads from compromising user privacy.

In this paper, we propose ANDetect, a tool specifically designed for identifying advertising libraries in Android applications. Utilizing static analysis of resource characteristics, ANDetect efficiently uncovers advertising libraries embedded in Android applications, thereby addressing the limitation of traditional third-party library detection methods that struggle with encrypted applications. ANDetect leverages a manual collection of 833 unique versions of third-party advertising libraries, combined with profiling and machine learning techniques. This approach utilizes distinctive semantic features in advertising and non-advertising libraries to identify advertising libraries outside of the established ad network database. We conducted an experiment using ANDetect on over 140,000 applications downloaded from Google Play and APPCHINA. Upon manual verification, it was revealed that ANDetect had detected a total of 16 noval advertising libraries, previously unregistered in the database. This underlines ANDetect's potency in enhancing mobile application security by identifying potentially intrusive advertising libraries.

## KEYWORDS

Third-party library, Android, Encryption, Advertising behavior

**ACM Reference Format:**
Xinyu Liu[1,2,†], Ze Jin[1], Jiaxi Liu[1,2], Wei Liu[1,2], Xiaoxi Wang[1], Qixu Liu[1,2,*]. 2023. ANDetect: A Third-party Ad Network Libraries Detection Framework for Android Applications. In *Annual Computer Security Applications Conference (ACSAC '23), December 04–08, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627106.3627182

*† First author: Xinyu Liu.
†* Corresponding author: Qixu Liu.



*ACSAC '23, December 04–08, 2023, Austin, TX, USA*

ACM ISBN 979-8-4007-0886-2/23/12.
https://doi.org/10.1145/3627106.3627182

## 1 INTRODUCTION

Android, an ever-evolving ecosystem, has commanded a substantial 72.51% of the global mobile market as of July 2022 [42]. Within this burgeoning platform, third-party libraries have become a prevalent feature, particularly among developers of free applications. Premier advertising network providers like AdMob [18] and Facebook Audience Network [12], servicing over 34% of the top 500 U.S. apps, underscore their pivotal role in the Android ecosystem. However, their pervasiveness raises concerns, as the presence of malicious advertising libraries could threaten user privacy.

**Ad threats exhibited by real-time bidding.** The rise of real-time bidding (RTB) technology [19], despite its advantages in automating competition for high-value ads and efficient ad serving, has facilitated the circumvention of ad platform oversight. Notwithstanding Google's restrictions [43] on in-app advertising within Google Play (the largest Android app market) [16] applications, RTB technology's inherent limitations facilitate the circumvention of ad platform oversight by ad publishers, thus exposing users to potential privacy breaches and fraud risks. Recent surveys [10, 37, 44, 45] underscore the extent of privacy concerns associated with RTB advertisers. Notably, Sung et al.'s research [45] illustrates how RTB advertisers exploit local storage, index databases, cookies, IP addresses, and geographic locations for continuous, low-cost user tracking, even in the face of user attempts to block such tracking. The sharing of this tracked data with other data brokers can substantially infringe on user privacy. A 2020 ProPrivacy survey [37] revealed that 31% of the UK's most frequented charity websites harbored trackers from RTB platforms, capturing user data, including age, gender, religion, and political affiliation, and shared this with thousands of companies.

**Challenges of identifying ad libraries.** While the identification of ad libraries has undeniable implications, particularly for user privacy, the inherent challenges within this domain continue to be formidable. (1) **Main Challenge**: Modern Android applications are often complex, containing multiple libraries – some of which may be incorporated inadvertently, perhaps imported by other libraries. The typical behavior and absence of malicious code in ad libraries make their large-scale detection difficult. To make matters worse, official platforms such as Google Play do not expose most of the third-party ad libraries they know. As a result, researchers struggle with the static identification of these libraries to figure out the correct attribution of ad fraud. (2) **Technical Challenge**: In the current Android ecosystem, another challenge is the encryption of apps, commonly used by developers to enhance security, protect intellectual property, and deter reverse engineering. For example, as showed in Figure 1, when an app is encrypted by Qihoo [2], its original code, structure and the third-party libraries like "com.qq.e.ads" and "android.supported.v4" are modified to "com.qihoo.util". Unfortunately, the nature of state-of-the-art tools like LibD for identifying the third-party libraries lies in extracting their original code or structure features which are completely lost after encryption. As a result, analyzing the third-party libraries by generating relation pattern or package hash, state-of-the-art tools couldn't recognize any original libraries but only "com.qihoo.util", thus affecting the accuracy of ad libraries detection. Therefore, the encryption nature introduces a significant technical challenge to the accurate detection of libraries.

**Shortcomings in current approaches**. Under the constraints of the above challenges, although prior studies [3, 11, 14, 20, 24, 26, 31, 36, 38] have enriched our understanding of ad fraud detection, their collective efforts have fallen short of overcoming the fundamental challenges, particularly in the realm of ad fraud attribution (the Main Challenge). Innovative dynamic detection frameworks such as FraudDroid, GFD, and FraudDetective [11, 24, 26] have emerged, successfully extracting behavioral patterns of ad fraud through heuristic rules and deep learning. However, they have conspicuously omitted the issue of ad fraud attribution, leaving this critical aspect unresolved. The exploitation of advertising libraries by developers, leading to deceptive advertisements, further highlights the inadequacies of current approaches [9]. Notably, researchers have uncovered adware disguised as a PDF reader, displaying full-screen ads even during inactivity, sourced from well-established libraries such as Facebook and Applovin[5]. While numerous studies [3, 31] have utilized traffic, content, and ad behavior for classification through machine learning algorithms, they have primarily concentrated on detecting malicious behavior, neglecting the crucial task of ad attribution. Pioneering work by Rastogi et al. [38] on malicious advertisement traceability introduced a model for identifying advertising libraries within Android applications, associating detected malicious ads with their initiating libraries. Yet, this model's limitation in attributing malicious ads in encrypted applications further underscores the shortcomings of existing methodologies. Furthermore, conventional detection schemes [14, 20, 36] that rely on whitelist-based matching following module decoupling have shown to be limited in recognizing a broader range of advertising libraries, reflecting an overall failure to effectively address the outlined challenges associated with ad library identification.

To this end, we present ANDetect, an innovative tool designed to accurately identify ad libraries embedded within Android app, the capability persists regardless of app encryption. ANDetect leverages an intricately curated advertising network database, aiming to aid in the accurate attribution of malicious ad activities. Traditional third-party library detection tools, such as LibD [29], LibScout [6], and LibRadar [32], employ clustering or signature methods to discern third-party libraries contained within an app. Despite their impressive accuracy in detecting such libraries in obfuscated apps, they struggle with encrypted apps, often failing to identify the included third-party libraries. ANDetect overcomes this limitation by leveraging resource features within the app, bypassing the encryption issue and successfully identifying the contained ad libraries (see § 3.5 for details). Furthermore, acknowledging the limitations of whitelist-based methods, we extract unique behavioral patterns from ad libraries, then construct an ad network discrimination model to detect more ad libraries beyond those in whitelist.

**Contributions:** Our contributions are outlined as follows:

• We introduced a unique approach that addresses the challenges of identifying ad libraries in encrypted Android applications by using a collection of resource features and matches the accuracy of code semantic-based identification with superior robustness.

• We built an ad network database, a comprehensive database of 833 versions of ad libraries from 162 ad network, ensuring accurate identification across different library versions. Furthermore, we collected over 140,000 apps from popular stores to test ANDetect. Realizing the lack of standard test results, we synthesized detection outcomes from LibD, LibScout, and LibRadar, creating the first open-source test dataset [25] for ad library detection in apps.

• We designed and implemented ANDetect, the first static-analysis based tool capable of identifying ad libraries in encrypted apps. It not only maintains detection efficiency but also extends beyond traditional whitelist-based methods, using unique ad library behaviors and package names for higher recognition accuracy. Additionally, with the assistance of ANDetect, we checked 16 brand-new ad network and 53 unmarked adware. Notably, we open-sourced ANDetect online on our website [25].

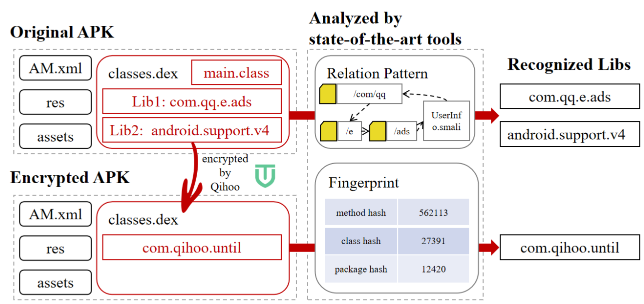



**Figure 1: The technical challenge from state-of-the-art tools.**

## 2 BACKGROUND

In this section, we provide an overview of the operational modes of ANs, encompassing both the traditional in-app advertising model and burgeoning programmatic real-time bidding mode. We elucidate the prevailing methods of code obfuscation, resource obfuscation, and encryption deployed within Android applications.

### 2.1 Advertising Networks

The traditional in-app advertising model is predicated on three elements: advertising publishers, AN platforms, and application developers. Advertising publishers provide the advertising networks (AN) platform with the requisite resources for the advertisement such as images and HTML codes. Concurrently, the application developer is tasked with registering on the AN platform and invoking the interface furnished by the AN library. When placing advertisements, advertising publishers distribute rewards to application developers based on the count of ad clicks within applications.

In recent years, the programmatic mobile advertising model — colloquially known as real-time bidding advertising — has been progressively encroaching upon the market share of the traditional in-app advertising model. Owing to its elevated level of automation, this model has garnered the approbation of both ad publishers and application developers. Application developers merely have to put

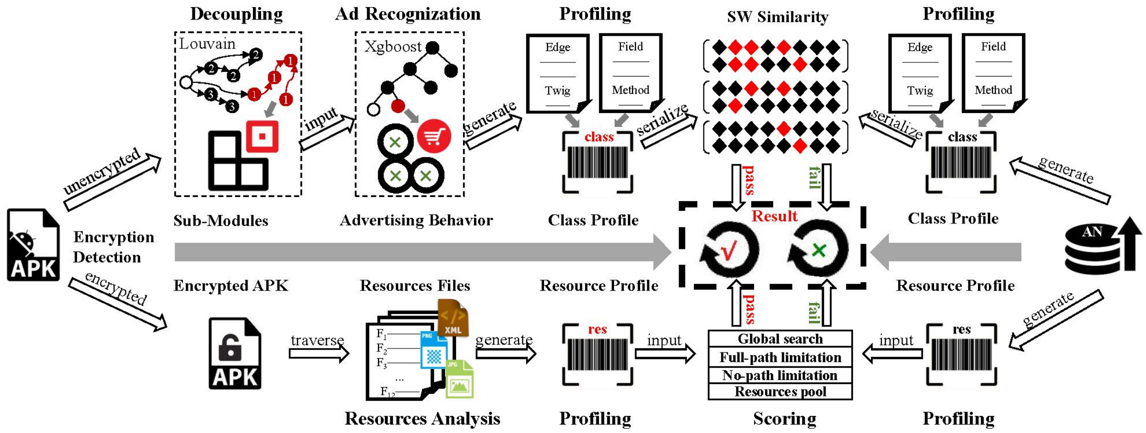



**Figure 2: The workflow of ANDetect.**

up the ad space within their applications for auction, and the AN platform autonomously assists them in procuring advertisements with higher bids, thereby bolstering the developer's profit. In a similar vein, advertising publishers place their ad resources on the AN platform and engage in bidding. The platform, in turn, locates suitable ad positions for them in line with their bids. The programmatic mobile advertising model affords the feasibility of collaboration between disparate AN platforms. To augment the channels for securing ad space, the AN platform may opt to bid its represented ad resources on the advertising exchange market.

Irrespective of whether a third-party library furnishes a traditional in-app advertising model or a programmatic advertising model, we collectively designate it as a AL. This serves as the focal point of our detection and security analysis in this study.

## 2.2 Application Obfuscation and Encryption

Obfuscation, implemented to reduce Android app size, employs strategies from basic character renaming to intricate API hiding through Java reflection and dynamic class loading technology, increasing reverse analysis complexity [52]. Common obfuscation tools include Allatori, dashO, DexGuard, DexProtector, ProGuard, Stringer, and PreEmptive [6]. Third-party libraries, however, advise against obfuscation due to potential invocation errors. For instance, Flurry [13] instructs to exclude `com.flurry` classes from Proguard obfuscation. Our profiling method in ANDetect allows for ad library detection even if obfuscated. Resource obfuscation, another technique, is used to further compress app size [40]. Our analysis shows that after obfuscation, resource file names in the *res* directory become meaningless characters, while resource images' names of the third-party library remain unobfuscated.

In the realm of AP encryption, developers possess the ability to encrypt the entirety of the DEX (Dalvik Executable) file. This file is the compiled manifestation of the source code, a crucial component encapsulated within AP, alongside resources, manifests, certificates, and other necessary assets required for the application's successful operation on an Android device. Encryption can be applied to user-defined functions and Android components, such as activities and services. In this context, the protected classes are eliminated from the original *classes.dex* file. This means that, unlike obfuscated APs, the source code of encrypted DEX files remains elusive, inaccessible through conventional reverse engineering tools. The encrypted application incorporates a decryption module. During the application's execution on the Android system, this decryption module decrypts the DEX file, thus reinstating the original one.

# 3 MODEL DESIGN AND IMPLEMENTATION

In this section, we put forward the design of ANDetect, encompassing the construction of the dataset and the identification methods targeting both encrypted and non-encrypted Android applications.

## 3.1 Design Philosophy

Conversely, for non-encrypted apps, we first establish the presence of advertising behavior before conducting ad library matching, thus enabling the identification of a broader range of third-party ad libraries beyond the advertising networks database (AN database). Following this philosophy, we design ANDetect.

The workflow of ANDetect is illustrated in Figure 2. The identification process of ANDetect is mainly divided into two steps: profiling and scoring. Profiling and scoring constitute two pivotal processes in enhancing the efficiency and accuracy of ad library (AL) detection in both encrypted and non-encrypted Android applications (APs). Profiling entails the extraction of critical components from APs, thereby reducing the computational overhead associated with the individual assessment of every AL. Meanwhile, scoring involves determining the similarity between an app and an individual AL, facilitating targeted analysis. Initially, we construct separate resource and class profiles for each ad library, delineating the distinct characteristics pivotal for identification. Subsequently, we categorize APs into encrypted and non-encrypted groups based on variations in package name attributes, which undergo alterations during the encryption process.

For encrypted AP, we generate a resources profile. For non-encrypted AP, we generate a class profile for advertising modules after decoupling. Subsequently, the profile will be matched with the corresponding profile of each ad library and the matching result determines whether the ad library is contained in this AP.

## 3.2 Preprocessing Data

Before delving into the detailed design of ANDetect, we present the data preprocessing phase which aligns with ANDetect's data preprocessing process and will be subsequently discussed in detail.

The package name of the AL is updated to align with its package structure. The original package name data in the dataset is derived from the identification results of LibD [29], LibRadar [32], and LibScout [6], as well as irregular names found on various websites and datasets. However, these names might not accurately correspond to each library's package, necessitating an update. This not only enhances the precision of each AN's identification but also standardizes the outcomes of third-party advertising traceability. The ALs are usually released as `jar` or `aar` files. While `jar` files can be directly decompressed using decompression software, `aar` files require a change of suffix to zip prior to decompression. The decompressed path of the `jar` file is recorded to update the corresponding package name based on its packaging principle. While the package name corresponding to the AL can be directly found in the manifest file obtained after decompressing the `aar`.

## 3.3 Profiling

Prior to the identification of ALs within an AP, we construct both resource and class profiles for each individual AL. In this subsection, we delineate the processes involved in extracting resource and class features, which are then used to formulate these profiles. While code semantic features offer a visual representation of an AL, the application of resource features to represent an AL remains a relatively uncharted territory. Selecting the most suitable features is a critical task. Accordingly, we will provide a comprehensive explanation of the effective utilization of resource features.

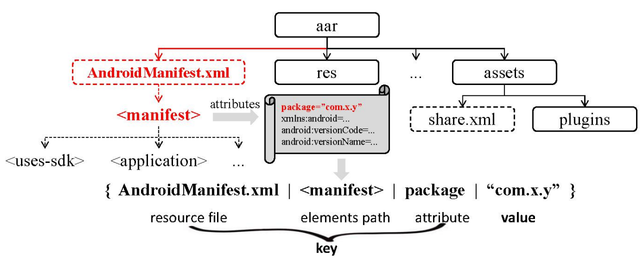



**Figure 3: The mapping of *key-value*.**

**Resources profiling:** Given the comprehensive modification of the original code and structure, identifying third-party libraries in encrypted apps is more effectively achieved through analyzing resource features, which is facilitated by the unique characteristics that advertising libraries exhibit in resource files.

Resource files, which comprise the *AndroidManifest.xml*, as well as files in the *res* and *assets* directories, are essential components of `jar` and `aar` files. When packaging an `aar` file, all resource files are included, whereas for a `jar` file, only *assets* is incorporated. Given the structural discrepancies between `jar` and `aar` files, resource profiles need to be generated through distinct methodologies.

To extract the effective feature set from `aar` files, we establish *key-value* mappings for all resource files with *xml* extensions. In these mappings, the *key* is made up of the resource file path and name, elements path, and attribute, as shown in Figure 3. The corresponding data of the attribute constitutes the *value*. As an excess of features can lead to model overfitting, it is vital to judiciously select the *keys* that constitute the final resource features set. The selection of these *keys* should satisfy the following three conditions:

`Universality`: The *key* should generally exist in different ALs, not unique to a certain AL.

`Differentiation`: The *value* of this *key* should be capable of distinguishing different ALs. We utilize the TF-IDF algorithm [39] to compute the differentiation of each *key*.

`Insensitivity`: The *value* of this *key* should be insensitive for different versions of one AN. We evaluate the sensitivity of all *keys* to different versions of the same ANs.

The computation procedure of universality, differentiation and insensitivity is given in Appendix B. Based on the aforementioned three dimensions, we generate three lists of keys, each sorted in descending order. We then select the intersection of keys present in the top 50 permutations of each list. Following this screening process, we extract a total of 11 keys. These keys collectively comprise the resource features of `aar`, as depicted in Appendix Table 5.

For `jar` files, extracting resource features similar to `aar` is challenging, given that `jar` files do not contain *AndroidManifest.xml* and the *res* directory. However, after extensive investigation of various developer documents, we discovered that when these AN `jar` files are integrated into an AP, they often define values that align with their package name prefixes in the manifest file elements. For instance, the developer documents of Dangbei, a prominent AN in China, instructs the addition of `<receiver android:name="com.dangbei.euthenia.receiver.NetworkChangeReceiver">` in the manifest when integrating this library, while its package name is `com.dangbei.euthenia`. As such, we employ the package name of a `jar` as $F_1$ in Appendix Table 5.

Furthermore, we identified that image-type resources within both `aar` and `jar` files could also serve as a distinctive category of features to denote different advertising networks. To avoid image overlay due to identical names during packaging an AP, ALs typically opt for meaningful names when designating important images. Such significant image names are capable of distinguishing various ALs, as exemplified by `applovin_ic_mediation_applovin.png` in APPLOVIN. Analogous to the process of calculating `distinction`, the significant image names are sifted via the TF-IDF algorithm and build image pool for each AL.

**Class Profiling:** For non-encrypted applications, the unaltered original code and structure facilitate effective *adlib* detection. We decompile the DEX file to obtain the package structure and class files, from which we extract the essential features to construct the class profile. Procuring the structural features of an AP can rapidly pinpoint the AL in the unobfuscated AP. The hierarchical characteristics inherent in the package structure determine its suitability to be represented by a tree, where the parent-child node pair signifies the upper-lower relationship of the package hierarchy directory. Additionally, edges and twigs serve as crucial elements of a tree, enabling rapid location of an app's *adlib* configuration.

`Edges`: In a package structure tree, the set of nodes denoted as $N$ depends on the hierarchy directory, while a pair of parent-child nodes is denoted as $< N_p, N_c >$. The number of parent-child node pairs is the same as all non-leaf nodes. Use edges $E$ to represent the set of all parent-child node pairs in the package structure tree, where $E = \{< N_p, N_c >, p \in \text{Non-Leaves}, c \in \text{Children of } p\}$.

`Twigs`: A twig is defined as a branch within the package structure tree, originating from the root node and extending to the leaf nodes and is denoted as a set of these nodes. The total number of twigs equates to all leaf nodes. Each class profile encapsulates a set of twigs, which we express as $T = \{T_i, i \in \text{Leaves}\}$.

The class file, once decompiled, contains comprehensive code implementation logic. However, generating a class profile directly from unprocessed class files would lead to significant time and storage overheads. Consequently, when creating the class profile, each class must be suitably compressed. The processed classes must retain their functions while also resisting certain obfuscation techniques, such as identifier renaming, string encryption, and dead-code elimination, among others, for the purpose of which, we select the following features. A class file encapsulates numerous vital pieces of information, including the class name, parent class, fields, methods, and called functions. When extracting code features, our focus primarily falls on the fields, methods, and called functions within the class. Given that a called function needs to be implemented within a method, we categorize code features roughly into two groups: field signatures and method signatures.

`Field`: We generate signatures for the fields, which are categorized into static and instance fields. Significant attributes that characterize these fields include their access rights, such as private or public, whether the field is static, and the field type. By concatenating these identifier strings, we obtain the field signature.

`Method`: Given that certain code obfuscation techniques can alter the class name, and such modifications can significantly impact the hash value, we utilize a string concatenation method for method signing in this study. A method signature comprises the method's access rights, whether the method is a static code block, the method's parameter type list, the method's return type, the class of the function invoked in the method, and the function name. The class of the functions within the method can be distinguished as either a system function or a custom function, depending on whether it's defined in the application classes. We classify system functions as those called from the Android SDK and JDK. We retain the name of the invoked function and the order in which it appears in the method, while disregarding the type and specific value of the parameters passed into the function and the return value type.

The class profile, resilient to identifier renaming, string encryption, and dead-code elimination, is constructed from the set of edges and twigs derived from the package structure, along with the signatures of every class file.

## 3.4 Non-encrypted application detection

As depicted in Figure 2, AL detection strategies can be distinctly categorized based on whether the AP is encrypted or not. Prior to selecting an appropriate detection approach, it's crucial to ascertain the encryption status of the AP. As mentioned in section 2.2, the centerpiece of AP encryption is the encryption of its *dex* file, a process that drastically alters the package structure. Consequently, it becomes impossible to locate the main class directed by the *package* attribute of *AndroidManifest.xml* within an encrypted AP. Leveraging this observation, we subsequently apply suitable detection methodologies. In this section, we focus on the detection strategies for non-encrypted APs.

**Decoupling:** To mitigate the interference from the host application during the decoupling process, we initially identify the host application's main model by selecting the value of the `package` attribute in the manifest. We then segregate the remaining structure and class files into distinct modules — referred to as sub-modules — through Louvain [7] which is a community partition algorithm. Leveraging the weak inter-library and strong intra-library association, Louvain is capable of partitioning different libraries according to the call relationship in an app. For each class, we extract the call and inheritance relationships, enabling the construction of a community network. After partitioning different communities from the whole package structure, we adopt a bottom-up merging strategy to generate sub-modules in accordance with the package structure tree, and the strategy is stated in detail in Appendix A.

**Advertising behavior:** Upon partitioning the package structure into sub-modules, we probe the advertising behavior of each sub-module. Only sub-modules exhibiting advertising behavior proceed to be matched with the Advertising Network (AN) database.

We categorize the Application Programming Interface (API) set from the Android Software Development Kit (SDK) and Java Development Kit (JDK) as system APIs. ANDetect discerns the advertising behavior of sub-modules by statistically analyzing the system APIs' call patterns, thereby abstracting the unique behavioral patterns of advertising libraries. We introduce a binary classification model, constructed with a dataset comprising of $Dataset_{AN}$ (ad modules) and $Dataset_{NAN}$ (non-ad modules) which is mentioned in section 4.1, to differentiate between ad modules and non-ad modules. The number and distribution of system API calls bear substantial behavioral significance. Given that using the entire set of system APIs as a feature set could induce model overfitting, we selected 210 APIs (which have surfaced in libraries from $Dataset_{AN}$) as potential feature sets. Ultimately, three types of features emerge: the number of candidate API calls, the distribution of candidate APIs across different classes, and the distribution of candidate APIs across distinct methods. For each of these feature types, we construct three separate XGBoost models and employ a voting mechanism for the classification results. Any sub-module identified as displaying advertising behavior is subsequently matched against the AN database.

**Smith-Waterman Similarity:** Sub-modules exhibiting advertising behavior are precisely associated with a specific AL in the AN database, or appended to the AN database as a novel AL. In the case of the former, ANDetect identifies a specific library version. Initially, we construct a set of candidate libraries from the AN database to constrict the scope of libraries for association, thereby accelerating the process. This candidate set may include libraries of diverse versions from different ANs. Subsequently, we generate a similarity matrix representing correlations between class profiles from the sub-module and AL. Each matrix value reflects a pair of class sequences' similarity, as determined by the Smith-Waterman algorithm [41] which performs local sequence alignment. Given that the dead-code elimination may delete a part of an unused field or method in class thus changing the original class sequence, the local sequence alignment algorithm is more suitable than the global sequence alignment algorithm. According to this similarity matrix, we maximize the similarity of class sequence pairs, with the AL exhibiting the highest similarity across all class sequence pairs confirmed as the result.

Referring to the definition of node and edge in class profiling, we define the node set in the package tree of a sub-module as $N^s$, the edge set as $E^s$, and the set of twigs as $T^s$. The class set within node $N_i^s$ is denoted as $C_i^s$. All nodes from an AL collectively form the set $N^a$, with the edge set represented as $E^a$ and the twig set as

$T^a$. The class set within node $N_i^a$ is denoted as $C_i^a$. It is crucial to discern whether the sub-module is obfuscated when constructing the candidate set. If it is obfuscated, we consider the possibilities of dead-code elimination and package flattening. The ALs in the candidate set must conform to these constraints:

$$|N^s| \leq |N^a|$$
$$\sum_{i=1}^{|N^s|} |C_i^s| \leq \sum_{i=1}^{|N^a|} |C_i^a| \tag{1}$$
$$|T_i^s| \leq |T_j^a|,\ \forall T_i^s \in T^s,\ \exists T_j^a \in T^a$$

In instances where multiple nodes within the sub-module possess a discernible hierarchical relationship, thereby suggesting the absence of flattening, additional constraints are formulated as follows:

$$\begin{cases} |\text{Children of } N_i^s| \leq |\text{Children of } N_j^a| \\ |C_i^s| \leq |C_j^a| \end{cases} \tag{2}$$
$$\forall N_i^s \in N^s,\ \exists N_j^a \in N^a$$

If the packages of sub-module are not renamed to a meaningless strings, the following constraints are additionally defined:

$$E_i^s = E_j^a, \forall E_i^s \in E^s, \exists E_j^a \in E^a \tag{3}$$

Ad libraries that meet the constraint 1 are added into the candidate set, denoted as $CA$. If $CA$ is an empty set, it indicates that the sub-module cannot match any library in AN database. Under such circumstances, we verify the existence of the library in Maven, and if found, download all versions of the library, subsequently adding them to the AN database. In instances where $CA$ contains at least one AL, ANDetect carries out calculations based on the **S**mith-**W**aterman (SW) similarity.

**Algorithm 1** Local Maximum Similarity Algorithm.

```
Input: Matrix SW ∈ {sw}^{s'×a'}
Output: The sum of local maximum value g(SW)
// Initialize the local maximum value.
g(SW) ← 0;
// Initialize a set to record the matched class in C^a.
Matched ← ⊘;
for (i = 1 → s') ∧ (|Matched| < a') do
    m ← 0;
    match_j ← 0;
    for j = 1 → a' do
        if (sw(i, j) > m) ∧ (j ∉ Matched) then
            m ← sw(i, j);
            match_j ← j;
        end if
    end for
    g(SW) ← g(SW) + m;
    Matched ← match_j;
end for
return g(SW)
```

If the sub-module is not obfuscated, $CA$ should be built subject to constraint 3. To construct a pair of class sequences, ANDetect serializes the field and method signatures in the sub-module's and AL's class profiles of the same name. Here, $sw(i, j)$ indicates the Smith-Waterman similarity of a pair of sequences, derived from class $c_i^s$ and class $c_j^a$. The similarity of sub-module $s$ and AL $a$ is denoted as $Sim_{s,a}$. When facing obfuscated sub-modules, ANDetect generates similarity matrices using the class profiles of the sub-module and all candidate libraries. If the sub-module contains a single node, and this node houses all class files, ANDetect creates a matrix $SW \in \{sw\}^{s' \times a'}$, with $s'$ and $a'$ denoting $|C^s|$ and $|C^a|$ respectively. Define the sum of local maximum similarities of $SW$ as $g(SW)$ and the calculation process is given in Algorithm 1. ANDetect then gets the sum of similarities $Sim_{s,a}$ across all pairs of class sequences, in accordance with $SW$. If the sub-module consists of multiple nodes, numerous similarity matrices are developed in line with constraint 2. Initially, the node $N_*^s$ containing the most classes is selected, along with the node set $N^a$ from the AL that satisfies constraint 2. Similarity matrices $SW^{*,j}$ are constructed based on all pairs of class sequences serialized from $N_*^s$ and $N_j^a$. The node $N_j^a$ associated with $N_*^s$ is determined by the following equation:

$$\arg\max_{N_j^a \in N^a} \{g(SW^{*,j})\} \tag{4}$$

After the exclusion of $N_*^s$ from $N^s$ and $N_j^a$ from $N^a$, ANDetect selects the subsequent $N_*^s$ and $N_j^a$. This iterative process continues until all nodes within the sub-module have undergone similarity computation. After obtaining all pairs of associated nodes, ANDetect maximizes the similarity of all classes belonging to the associated nodes, and then sums to get $Sim_{s,a}$.

In the end, the maximization of total similarity $Sim_{s,a}$ between the sub-module $s$ and the AL $a$ is pursued, thereby identifying the corresponding AL:

$$\max_{a \in CA} Sim_{s,a},\ \text{where } C' = \{C^s \cap C^a\} \tag{5}$$
$$Sim_{s,a} = \begin{cases} \sum_{c' \in C'} sw(c'^{(s)}, c'^{(a)}) & \text{, if satisfy (1) and (3)} \\ \sum_{N_j^a \in N^a} g(SW^{*,j}) & \text{, if satisfy (1) and (2)} \\ g(SW) & \text{, if only satisfy (1)} \end{cases}$$

The name and certain version of AL $a$ is given by ANDetect as the final result in non-encrypted application detection.

## 3.5 Encrypted application detection

Typically, encrypting an Android application does not modify the resource files within it. Thus, even if the resources of the AP are obfuscated, certain resource features persist. We abstract these resource features to create a resource profile, which aids in comprehensive scoring to determine the presence of any AL within an encrypted AP. As described in this section, the scoring methods are categorized into "global search", "full-path method", "no-path method", and "image pool method", each reflecting the characteristic features outlined in Appendix Table 5.

**Global search**: The packaging of an AP entails merging the *<manifest>* elements of the host application and third-party libraries. In this process, only attributes with the highest priority persist. Thus, the *package* attribute of the *<manifest>* element post-merging refers to the host application's entry class, making it unsuitable to match $F_1$ following the original path. Nevertheless, AL developer documents often explicitly require component declarations in the *AndroidManifest.xml*. For instance, *keymob* requires the *android:name* of the *activity* component to be configured as `com.keymob.sdk.core.KeymobActivity` in the manifest file. The values corresponding to the components are prefixed with the package name of the AL which is equivalent to $F_1$ in the resource profile. The package names of ALs are often distinct, even different types of third-party libraries developed by the same company have different package names. As an example, the package name of AL developed by *Yandex* is `com.yandex.mobile.ads` while its map library is named as `com.yandex.maps.mobile`. Therefore, we assert that if an element's *android:name* prefix precisely corresponds to an AL's package name, the AP undeniably incorporates this library. For $F_1$, ANDetect employs a global search approach. Specifically, it compares the *android:name* prefix of all elements in an AP's manifest file against $F_1$ in the resource profile of the AL.

**Algorithm 2** Relaxation Matching Algorithm.

**Input:** Features $F_P$ from AP, $F_L$ from AL, Package name $n$ of AL
**Output:** The similarity $sim$ between $F_P$ and $F_L$

```
if prefix(F_N) == n then
    P ← delete n in P_N and split P_N with ".";
    L ← delete n in P_L and split P_L with ".";
    let D^{(|L|+1)×(|P|+1)} be a matrix;
    sim ← 1;
    for i → [0, |L|] do
        for j → [0, |P|] do
            if i == 0 then
                D[i][j] ← 2^{|P|-j} / Σ_{k=1}^{|P|} 2^{|P|-k} ;
            else if j == 0 then
                D[i][j] ← 2^{|L|-i} / Σ_{k=1}^{|L|} 2^{|L|-k} ;
            else
                ρ ← (P[j] ∈ AN list) ? 0 : 1
                d ← (L[i] == P[j]) ? 0 : 1;
                del ← D[i][j-1] + ρ · 2^{|P|-j} / Σ_{k=1}^{|P|} 2^{|P|-k};
                ins ← D[i-1][j] + 2^{|P|-j-1} / Σ_{k=1}^{|P|} 2^{|P|-k};
                rep ← D[i-1][j-1] + dρ · 2^{|P|-j} / Σ_{k=1}^{|P|} 2^{|P|-k};
                D[i][j] ← min(del, ins, rep);
            end if
        end for
    end for
    sim ← sim - D[|L|][|P|];
else
    sim ← 0
end if
return sim
```

**Full-path method:** The paths, attributes, and values of $F_2$, $F_3$, $F_4$, $F_5$, $F_6$, and $F_7$ remain unaltered during packaging, enabling their identical counterparts in the merged and original library manifests to be identified. Consequently, ANDetect applies a full-path approach to match $F_2$ through $F_7$, where 'path' denotes the file path and elements path in the key of each feature. These features are further divided into stable and variable features based on their characteristics.

Stable features including $F_2$, $F_3$, and $F_4$, maintain consistent values despite version changes. Although these features are not exclusively unique to individual ALs, and may recur across various libraries, their combination can serve as a robust identification reference. $F_2$ is a feature that represents permissions including custom permissions, like `com.goole.android.gms.permission.AD_ID`, named by the developer. $F_3$ is a URI string comprising *applicationId* and a custom name, indicating the user of *FileProvider*. $F_4$ corresponds to the action of *intent*, which includes both system-defined and custom actions. These three features may be added or removed across different library versions, yet their string characteristics typically persist. For these stable features, we perform a complete feature comparison.

Variable features, such as $F_5$, $F_6$ and $F_7$, undergo partial string modifications with version changes. Given the possible alterations in features values, a comprehensive comparison is unfeasible. Instead, ANDetect employs a relaxed matching approach. A variable feature value is defined as a prefix and individual strings separated by ".", where the prefix refers to the package name. The method assumes that version changes do not affect these feature prefixes, and any changes to individual strings are minimal and highly correlated with the AN. We maintain a list of AN related words, such as ad(s), network(s), adview, etc. We propose a weighted relaxation matching algorithm outlined in Algorithm 2, to calculate the similarity between each variable feature in the AP and the resource profile from AL.

**No-path method:** For non-manifest resource files, the Android Asset Package Tool assigns each non-assets resource an ID during the xml file compilation before storing it to the R.java file, and generates *resources.arsc* to index resources assigned ID. The original paths of $F_8$, $F_9$, $F_{10}$, and $F_{11}$ from AL are blurred during the packaging process. These features are merged into the *resources.arsc* based on their attributes, hence the feature path is disregarded during the matching process. After parsing *resources.arsc*, we identify the same values corresponding to the same attribute in the AL, employing a method known as "no-path". Given that certain features exhibit weak correlation with the corresponding AL — for instance, $F_9$ of `com.ironsource.sdk.mediation` is *app_name* — it becomes crucial to weight features using equation (8).

**Image pool method**: When a developer incorporates a third-party library, the images from the *assets* and *res* directories are relocated to their respective directories within the AP. The image pool method involves verifying if the path in the AP encompasses the same image name with the images present in the image pool of an AL.

**Comprehensive scoring:** Each of the aforementioned four matching methods is assigned a specific scoring method to estimate the similarity between the resource profiles of the AP and the AL. This process allows the determination of whether the AP includes a particular AL, and if so, which one. Here, $F_n$ represents the set of all values correlating to feature *n* in the AL, while $V_n$ symbolizes the set of all values within the AP that satisfy the matching rules of $F_n$. The computation is detailed in Appendix C. The correlation between individual features and the ad library, alongside the possibility of feature elimination during the AP packaging process, allows us to classify $F_1$ to $F_{11}$ and $PN$ into strong association features and weak association features. Strong association features ($F_1$, and $F_5$ to $F_7$) have the capacity to uniquely distinguish ad libraries. On the other hand, weak association features ($F_2$ to $F_4$, $F_8$ to $F_{11}$, and $PN$) typically require a collective application to differentiate libraries. Accordingly, we assign the weight $\omega_1$ to the strong association features and $\omega_2$ to the weak association features. Upon scoring these features according to their respective rules, ANDetect computes the final similarity score between an AP and an AL via a process of weighted summation and normalization.

Finally, ANDetect computes the similarity between the resource profile of the AP and that of the AL. A threshold $\theta$ is established with the comprehensive score exceeding $\theta$ indicative of the AP containing the ad library. The selection of optimal weights $\omega_1$ and $\omega_2$, along with the threshold $\theta$, is subsequently validated in § 4.2.

## 3.6 Implementation

When generating resource profiles for all ad libraries in AN database, after decompressing the `aar` file, find all resource files in the *AndroidManifest.xml*, *res* and *assets* directories if existing, and extract features $F_1$ to $F_{11}$ and build the image pool. After decompressing the jar file, generate feature $F_1$ according to the package name where the core class is located, find the resource files located in *assets* and build the image pool. When generating class profile, find the classes.jar file in an aar file, and compile the classes.jar to *dex* file with d8 [17], one of the Android Build Tools. For jar file, we skip the decompression step and compile it to *dex* file directly. And then use baksmali [21] to decompile the *dex* file into smali files, ANDetect analyzes the structure and code features of each advertising library to generate class profile. ANDetect analyzes Android applications by using baksmali to decompile the classes.dex obtained after decompression.

ANDetect implements the advertising behavior discrimination model described in § 3.4 using xgboost4j [33] when identifying the advertising behavior of non-encrypted applications. Train XGBoost model based on $Dataset_{AN}$ and $Dataset_{NAN}$ mentioned in § 4.1, save to *json* file. When ANDetect detects the advertising behavior of the sub-module, the XGBoost model saved in json is read and binary classification is performed to generate recognition results.

# 4 EVALUATION

In this section, we conduct a comprehensive evaluation of ANDetect across multiple dimensions. Specifically, we scrutinize the appropriateness of the hyperparameter settings within ANDetect, the efficacy of ANDetect in detecting ad libraries in both experimental and real-world environments as well as the capacity of ANDetect to identify novel advertising libraries beyond those in the AN database. Additionally, we verify the robustness of ANDetect when handling encrypted applications in Appendix 4.4. To accurately quantify ANDetect's performance, we have established the following evaluation metrics:

- Fine-grained True Positive (TP-FG): For Android applications with advertising libraries, fine-grained true positive indicates the probability that the predicted ALs can completely cover the real ALs.
- Coarse-grained True Positive (TP-CG): For all Android applications, coarse-grained true positive suggests the probability that successfully predicts there are advertising libraries in an Android application as absence.
- Precision: Precision presents the probability that the predicted result is correct when a certain advertising library is predicted to exist.
- Recall: Recall means the probability that the predicted result is correct among all advertising libraries existing in an application.

Among the above metrics, TP-FG, precision and recall are fine-grained and accurate to every ad library while TP-CG is coarse-grained and accurate to every application.

## 4.1 Dataset construction

In this section, we introduce the construction process of the datasets used in this paper and the different application approaches.

**Android Application Dataset:** We constructed the Android App (AP) dataset from over 140,000 applications sourced from AndroZoo [4], Google Play, and APPCHINA. 20,000 of these apps released post-2016 and labeled as adware by MadDroid [31] were downloaded from the AndroZoo repository and the set was labeled as $AP_{\text{adware}}$. The remaining 125,401 unique applications, also released after 2016, were collected from Google Play and APPCHINA to form $AP_{\text{truth}}$. To facilitate an extensive analysis, we classified these datasets into two categories based on the confidence in the labeling:

•*Low-confidence labeled datasets*: Crafted through the utilization of tools such as LibD, LibScout, and LibRadar to pinpoint and label the ad libraries in the $AP_{\text{adware}}$ and $AP_{\text{truth}}$ sets. Subsequently, we amalgamated the results obtained from each tool to frame the labels for each sample in the $AP^{*}_{\text{adware}}$ and $AP^{*}_{\text{truth}}$ datasets.

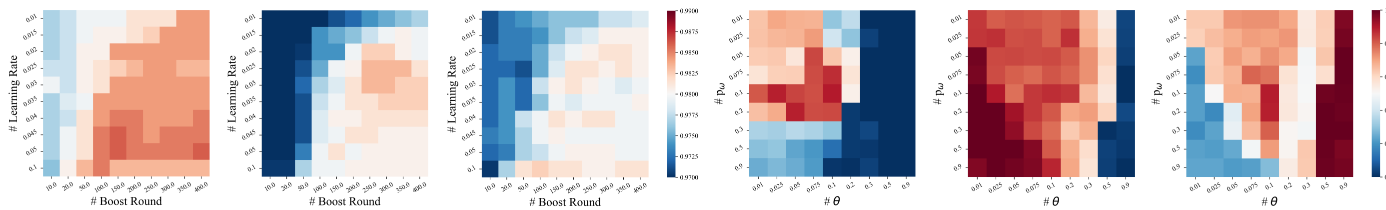
•*High-confidence labeled datasets*: Created by meticulously annotating 300 applications from $AP_{\text{adware}}$ and $AP_{\text{truth}}$ respectively, of which 200 were non-encrypted and 100 were encrypted. The annotation process involved: (1) Undertaking advertising UI tests, (2) Employing Frida [35] to access the function call stacks and trace them to third-party libraries, and (3) Recognizing them as advertising libraries. We referred to these datasets as $AP^{200}_{\text{adware}}$, $AP^{200}_{\text{truth}}$, $AP^{100}_{\text{adware}}$, and $AP^{100}_{\text{truth}}$, respectively.

**Ad Networks Dataset:** As the ALs obtained from open-source platforms often contain noise, we manually collected AL information from various platforms such as Maven Central and Google Play SDK Index. However, these platforms only provide a limited number of popular ALs. To find more ALs, we used popular open-source third-party library detection tools, namely LibScout, LibD, and LibRadar, to detect $AP_{adware}$ and $AP_{truth}$. We then ranked and filtered the detected third-party libraries and scripted a web crawler to automatically search for third-party library information. The collected SDK information was then matched with ad network-related keywords. Following manual verification, we compiled 833 different versions of ALs provided by 162 AN platforms into a third-party AL dataset, denoted as $Dataset_{AN}$. To learn the unique behavior patterns of advertising libraries, we collected 2758 non-advertising libraries, classified as Testing Frameworks & Tools, Android Packages, Logging Frameworks, and others from Maven Central. This collection is denoted as $Dataset_{NAN}$.

## 4.2 Hyperparameter setting

The areas in ANDetect that involve hyperparameter settings include the XGBoost model constructed to detect advertising behavior in § 3.4 and the weights of strongly and weakly correlated features and the thresholds of composite scores in § 3.5. We next describe the evaluation of the hyperparameter settings in each of these two sections and the integration of the most appropriate parameters in ANDetect.

The hyperparameters of the XGBoost model include boost round, learning rate, the maximum depth of the tree, and the proportion of randomly sampled features. Here we consider that boost round and learning rate are decisive for the performance of XGBoost in detecting advertising behaviors. To evaluate the two hyperparameters, we define accuracy as the metric and construct test datasets from $Dataset_{AN}$ and $Dataset_{NAN}$, dividing the training and test sets in a 7:3 ratio. As we remarked in § 3.4, based on three feature sets, three XGBoost models were constructed separately. The model evaluation on feature sets generated by the number of candidate API calls (API count), the distribution of these candidate APIs across different classes (DIST across classes) and the distribution of candidate APIs across different methods (DIST across methods) are shown in Figure 4a, 4b and 4c respectively. It is not requisite for



(a) API count (b) DIST across classes (c) DIST across methods (d) TP-FG (e) TP-CG (f) TN-FG

**Figure 4: The influence of different hyperparameter setting in advertising behavior identification and resources profile scoring. Based on three different feature sets, the accuracy of advertising behavior identification is shown in (a), (b) and (c). The three metrics of evaluating encrypted applications detection are shown in (d), (e) and (f).**

**Table 1: The performance of ANDetect compared with other three tools in the experimental environment.**

| Tools | TP-CG | TP-FG | Precision | Recall | Time(s)/AP |
|---|---|---|---|---|---|
| LibD | 46.08% | 78.31% | 99.72% | 78.31% | 58.14 |
| LibRadar | 48.04% | 77.86% | 99.50% | 77.86% | 29.37 |
| LibScout | 25.49% | 72.64% | 99.54% | 72.64% | 74.54 |
| ANDetect | 95.10% | 95.97% | 95.84% | 95.97% | 34.07 |

**Table 2: The performance of ANDetect compared with other three tools in the real-world environment.**

| Tools | TP-CG | TP-FG | Precision | Recall | Time(s)/AP |
|---|---|---|---|---|---|
| LibD | 47.40% | 62.67% | 98.73% | 62.67% | 153.45 |
| LibRadar | 43.35% | 59.69% | 98.26% | 59.69% | 65.11 |
| LibScout | 32.37% | 53.47% | 99.79% | 53.47% | 246.51 |
| ANDetect | 93.64% | 93.08% | 97.29% | 93.08% | 69.24 |

the hyperparameters to be identical. By selecting the optimal hyperparameters, detailed in Appendix Table 6, for different models, ANDetect achieves outstanding performance on the test samples, boasting an accuracy rate of 98.75%.

In detecting adware in encrypted applications, we need to set the weight $\omega_1$ for strong association features and $\omega_2$ for weak association features, and satisfy $\omega_1 > \omega_2$, set the threshold value $\theta$ for the composite score, and when the composite score is higher than $\theta$ the adware is detected. Here, we randomly select 1000 tagged samples of non-encrypted applications $AP^{1000}_{adware}$ from $AP^{*}_{adware}$, and detect these non-encrypted samples using ANDetect's encrypted application detection module mentioned in § 3.5. Considering that the composite score is normalized for the scores obtained from the four categories of methods, setting the ratio of $\omega_2$ to $\omega_1$, denoted as $p_\omega$, has a substantial impact. In order to choose the appropriate $p_\omega$ and $\theta$, we use the method of control variables to restrict $p_\omega$ and $\theta$ to belong to (0,1), and draw a heat map to find the most appropriate $p_\omega$ and $\theta$. Here, we define a new indicator Fine-grained True Negative (TN-FG) which evaluates ANDetect's detection accuracy for encrypted apps without ad libraries to prevent over-detecting. TP-FG, TP-CG and TN-FG are utilized as evaluation metrics to comprehensively assess the two hyperparameters and the result is shown in Figure 4d, 4e, 4f. As a result, given the variation of these metrics based on $AP^{1000}_{adware}$ in an integrated way, it achieves the optimal solution when $p_\omega$ as well as $\theta$ reach 0.1.

## 4.3 Performance of ANDetect

In this section, we assess ANDetect's performance against LibD, LibRadar, and LibScout, under both experimental and real-world conditions, delineated in § 4.1 based on the dataset origins. The experimental analyses employed subsets of $AP_{adware}$, while the real-world evaluations utilized subsets of $AP_{truth}$. An eShard study [47] informs that 39% of Google Play Store apps employ obfuscation and encryption for security. Leveraging a 2:1 mix of non-encrypted and encrypted applications, ANDetect outperformed its peers, a phenomenon detailed subsequently.

In the controlled setting, we integrated $AP^{100}_{adware}$ and $AP^{200}_{adware}$ — comprising 100 encrypted and 200 non-encrypted APs, respectively. Table 1 presents the performance metrics of ANDetect and other tools on the experimental dataset. Conversely, the real-world assessment involved amalgamating $AP^{100}_{truth}$ and $AP^{200}_{truth}$, with results depicted in Table 2. Following meticulous manual annotations, we ensured high reliability for the datasets used in the evaluations.

We employed five metrics: TP-CG, TP-FG, precision, recall, and processing time, facilitating a comprehensive evaluation. First four metrics are described before, and the "time" metric reflects the average processing duration for an app, recorded in seconds.

As observed from Table 1 and Table 2, ANDetect surpasses its competitors in both coarse and fine-grained evaluations due to its proficiency in detecting profit avenues from advertisements and accurately identifying the corresponding AL. While it trails slightly in precision, an offset owing to its broader scope encompassing emerging ALs and leading to potential false positives, it boasts a significantly higher recall, particularly in real-world settings. This is partly due to the competitors' misclassification of `com.google.android.gms.ads`, a prevalent library in the wild. The rival tools, inherently focusing on the entire libraries, missing advertising sub-modules, impacting course-grained results adversely. Despite ANDetect's comprehensive approach cause a slight delay compared to LibRadar, innovative strategies like profile extraction enable ANDetect to outspeed LibD and LibScout, presenting a beneficial balance between speed and precision.

## 4.4 Robustness in encrypted AP detection

To assess the robustness of ANDetect in detecting encrypted applications, we assembled evaluation datasets $AP'_{adware}$ and $AP'_{truth}$ by randomly selecting 100 labeled non-encrypted applications each from $AP^{*}_{adware}$ and $AP^{*}_{truth}$ that undergo resource obfuscation. As delineated in Section 4.1, each tag in $AP^{*}_{adware}$ and $AP^{*}_{truth}$ derives from the combined outputs of LibScout, LibD, and LibRadar.

Following the direct unzipping of the Android application, it is deemed to have resource confusion if it meets any of the following properties:

**Table 3: The result of encrypted application detection in resource confused applications.**

| Dataset | size | TP-CG | TP-FG | Precision | Recall |
|---|---|---|---|---|---|
| $AP'_{adware}$ | 100 | 94.00% | 86.24% | 81.84% | 86.24% |
| $AP'_{truth}$ | 100 | 94.74% | 91.50% | 72.27% | 91.50% |

**Table 4: The candidates of malicious ALs.**

| Candidate ALs | VirusTotal's Label | Relevance |
|---|---|---|
| com.adsmogo | Adware.ADWARE/ANDR.AdMogo | 0.50 |
| net.youmi.android | Android.Adware.Youmi.A | 0.47 |
| com.mopub.mobileads | AdLibrary:MoPub | 0.45 |
| com.inmobi.ads | AdWare.AndroidOS.Inmobi | 0.80 |
| com.nd.dianjin | Android.Dianjin.A (PUP) | 0.50 |
| cn.smartmad | Android.Adw.SmartMad | 0.12 |
| cn.domob.android | Android.Domob.A (AdWare) | 0.26 |
| com.kuguo.ad | Andr.Adware.Kuguo-4 | 1.00 |
| com.jirbo.adcolony | Adware/AdColony!Android | 0.50 |
| com.sixth.adwoad.mraid | ADWARE/ANDR.AdsWo.FAN.Gen | 0.43 |
| org.iqiyi.video | AdWare.AndroidOS.IqiAd.a | 1.00 |
| com.revmob | Adware.Revmob.1.origin | 0.20 |

- The *res* directory is absent in the application.
- While the *res* directory is present, over half of the file names, excluding suffixes, comprise fewer than 5 characters.
- The *color* or *xml* directories are found under the *res* directory, and more than half of the file names within them, disregarding suffixes, contain less than 5 characters.

Subsequently, the advertising libraries within the applications are identified exclusively through ANDetect's encrypted application detection module. The effectiveness of this module, scrutinized using four metrics on $AP'_{adware}$ and $AP'_{truth}$, is documented in Table 3. Clearly, ANDetect demonstrates commendable proficiency in identifying the ad libraries pinpointed by LibScout, LibD, and LibRadar.

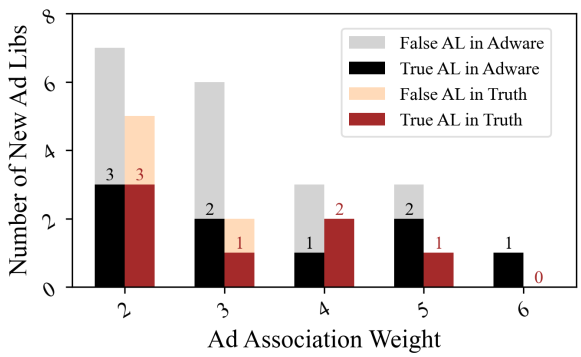



**Figure 5: The novel ad libraries detected by ANDetect. The number of genuine ALs and false positives from $AP_{adware}$ and $AP_{truth}$ are given respectively.**

## 4.5 Novel ad libraries and adware

ANDetect's non-encrypted application detection module is able to explore more novel advertising libraries that are unregistered in the AN database. ANDetect detects applications in $AP_{adware}$ and $AP_{truth}$ separately, detecting 20 novel ad libraries from $AP_{adware}$, and 10 from $AP_{truth}$. Actually, due to the potential for false positives generated by ANDetect, not all newly identified ad libraries exhibit advertising characteristics or function as ad network platforms. In order to discern the true ad libraries from original detection result, we define the ad association weight to quantify the probability that the library is a real ad library as expected and uses the library name as keywords to search ad association words in Google to generate weight. Eventually, we proofread the authenticity of whether this library is indeed an advertising library manually and the contradistinction between the novel ad libraries recognized by ANDetect and true ad libraries is shown in Figure 5. Although ANDetect boasts a 98.75% accuracy rate in recognizing advertising behaviors in the test dataset, as described in section 4.2, it is not exempt from producing false positives in real-world environments. We allow for false positives to identify more novel ad libraries and ANDetect indeed helped us identify a total of 16 novel ad libraries.

VirusTotal [1] is a well-recognized platform aggregating security analysis results from various vendors for applications and URLs. In our study, we labeled all apps in $AP_{adware}$ using VirusTotal results, further analyzing these labels statistically to explore the relationship between advertising libraries and adware. Utilizing VirusTotal's labeling, we identified potential malicious ALs through the correlation between "adware" labels and AL package names, detailed in Table 4. The last column in the same table indicates the relevance between each candidate and its label, derived from the ratio of labeled apps containing the candidate to the total number of apps featuring it, all sourced from $AP_{adware}$. This approach allowed us to highlight apps incorrectly classified as benign by VirusTotal, facilitating a deeper exploration into real-world adware. We examined 3000 randomly selected APs from $AP_{truth}$, eliminating those without candidate ALs and those labeled "malicious" by VirusTotal, resulting in 212 remaining applications. We then manually assessed each for aggressive advertising behavior, following four properties to designate an app as adware, thereby addressing the prevalence of false negatives in VirusTotal's classifications. The properties are as follows:

- Ads within an AP cannot be closed.
- A substantial ad area hinders the normal usage of the software's functionalities.
- Ads automatically trigger a file download interface, even without the user clicking on the ad's download button.
- The user is forced to view an ad for a prolonged period (exceeding three seconds) before being able to use the AP normally.

Ultimately, after omitting apps hindered by network irregularities or system incompatibilities, we identified 53 brand new adware instances, all labeled "benign" by VirusTotal. As illustrated in Appendix Table 7, a notable revelation is that each of these newly discovered adware in the real-world environment contains one or both of the following: "com.inmobi.ads" or "com.mopub.mobileads." This phenomenon could potentially be attributed to the pronounced and intrusive advertising behavior manifest in these two ALs, which, notably, bypasses VirusTotal detection.

# 5 DISCUSSION

In section 4, we compare the performance of ANDetect with three well-known third-party library detection tools in terms of advertising library detection. Despite some applications being encrypted, ANDetect can detect some of these advertising libraries. For non-encrypted applications, ANDetect outperforms other third-party library detection tools. One reason is that the ad behavior detection method in ANDetect helps us identify additional ad libraries beyond AN database, thus breaking through the limitations of traditional whitelist-based detection.

**Limitation**. Although ANDetect performs well on the test dataset, it still has certain constraints. Focusing on encrypted application detection, even though ANDetect is robust to APKs after resource obfuscation, the resource features of the ad library are restricted. Resource features do not have a clear differentiation effect like class features, which is reflected in different versions of ad libraries. Most ad network platforms do not update resource files when updating ad libraries, which makes resource features resilient and impossible to precisely match to different versions of ad libraries. In addition, not all advertising libraries have meaningful resource features.

To counteract the potential for malicious libraries to rename themselves and thereby evade detection, we routinely download the latest versions of ALs from the advertising platform. This process reflects the typical approach undertaken by developers when incorporating ad libraries: they log in to the ad platform to retrieve the necessary library, a procedure that is emulated within our dataset maintenance regimen. Consequently, should a malicious library undergo renaming to sidestep detection, our dataset is equipped to capture its revised attributes through systematic downloads. In our ongoing commitment to bolster detection efficacy, we pledge to persistently update $Dataset_{AN}$ with newly released ad libraries.

An approach based on static analysis can quickly identify an advertising library in an application that has the potential to fraud and track users with bypassing regulatory. Future work could focus on attribution of third-party ad fraud and tracking, attributing problems in third-party advertising to specific ad network platforms, and contributing to the strict regulation of third-party advertising.

## 6 RELATED WORK

Third-party library detection for Android applications has been a high-profile research point for a long time. Most of the prior research is based on whitelist with other techniques, including clustering, machine learning, signatures, etc. Li et al. collected 240 different versions of ad libraries and compare the similarity with 1500k applications from GooglePlay to confirm the presence of ad libraries [28]. Wukong [48] is a classic clustering-based third-party library detection tool that detects suspicious third-party libraries based on static semantic features. However, these researches are based on the assumption that third-party libraries included in applications maintain their original package names, which ignore the possibility of application obfuscation and is no longer applicable in most current applications.

Same as [38], AMdLens's [22] implementation is based on 164 advertising network collected manually by the author, extracts the feature vectors of calling APIs in third-party libraries, and uses Jaccard coefficients to map the decoupled sub-modules of the application to different advertising libraries. Nevertheless, both [38] and [22] utilize a set of API vectors as a mapping of third-party libraries, and these detection models fail when dead-code elimination applications are encountered. AdDetect [34] and PEDAL [30] extract bytecode features, including Android components, permissions, and APIs to build feature vectors and perform binary classification with Support Vector Machines (SVM), which can achieve high recognition accuracy, but are also limited by dead-code elimination applications.

LibD [29], LibRadar [32], LibScout [6], LibDetect [15], LibDX [46], LibID [53], LibPecker [54] and ORLIS [50] all extract class dependencies and sign the bytecode to speed up the matching of semantic features and resist a certain degree of application obfuscation. However, these tools are not resistant to packet flattening attacks, resulting in susceptibility to false negatives [52]. LibRoad [51] improves LibPecker by using the package name matching method for non-obfuscated applications and the signature method for obfuscated applications, thus speeding up the third-party library identification process on top of the original one. However, these methods are based on identifying non-encrypted applications and do not contribute to recognizing third-party libraries in encrypted applications. Whitelist-based methods cannot escape from the limitations imposed by it when detecting ad libraries in applications, which means they cannot detect other ad libraries outside the whitelist. In Section 4, we demonstrate that ANDetect can not only escape the limitations of whitelisting and identifying more advertising libraries that exist outside the database, but also be able to resist application encryption attacks to some extent.

Privacy leakage risks in third-party libraries have been analyzed, with evidence suggesting that most ad libraries collect private information [20]. Techniques have been developed to assess this risk by extracting sensitive APIs from apps [23] and detecting covert data collection by various libraries [49]. Malicious third-party library variants can be identified to spot repackaged applications, which often contain non-original libraries. Methods used include Jaccard coefficients to identify malicious libraries variants based on shared code [8], feature fingerprinting to detect repackaged applications [55], and the multi-level framework SimiDroid for Android application comparison [27].

## 7 CONCLUSION

We introduced ANDetect, a pioneering tool designed to efficiently identify ad libraries in Android applications, overcoming the limitations of existing methods especially when dealing with encrypted apps. Through leveraging a comprehensive collection of 833 unique versions of ad libraries and using advanced profiling and novel machine-learning techniques, ANDetect has proven effective in uncovering previously undetected ad libraries. The experiment of ANDetect to over 140,000 apps has resulted in the detection of 16 novel ad libraries, substantiating its contribution to improving mobile application security. These advancements mark a significant step towards ensuring user privacy and security in the face of rapidly evolving ad libraries.

## ACKNOWLEDGMENTS

This work is supported by the Strategic Priority Research Program of Chinese Academy of Sciences (No. XDC02040100), the Key Laboratory of Network Assessment Technology at Chinese Academy of Sciences and Beijing Key Laboratory of Network security and Protection Technology.

# REFERENCES


[1] 2023. *VIRUSTOTAL.* https://www.virustotal.com/gui/home/upload

[2] 360JiaGuBao. 2023. *Android Application Hardening.* https://jiagu.360.cn/#/global/details/app

[3] Mohammed M. Alani and Ali Ismail Awad. 2022. AdStop: Efficient flow-based mobile adware detection using machine learning. *Comput. Secur.* 117 (2022), 102718. https://doi.org/10.1016/j.cose.2022.102718

[4] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[5] Applovin. 2023. *Integration.* https://dash.applovin.com/documentation/mediation/android/getting-started/integration

[6] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* 356–367. https://doi.org/10.1145/2976749.2978333

[7] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[8] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016.* 357–376. https://doi.org/10.1109/SP.2016.29

[9] Nathan Collier. 2022. *Adware found on Google Play — PDF Reader serving up full screen ads.* https://www.malwarebytes.com/blog/news/2022/08/adware-found-on-google-play-pdf-reader-servicing-up-full-screen-ads

[10] Erdong Deng, Huajun Zhang, Peilin Wu, Fei Guo, Zhen Liu, Haojin Zhu, and Zhenfu Cao. 2019. Pri-RTB: Privacy-preserving real-time bidding for securing mobile advertisement in ubiquitous computing. *Inf. Sci.* 504 (2019), 354–371. https://doi.org/10.1016/j.ins.2019.07.034

[11] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F. Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. 2018. FraudDroid: automated ad fraud detection for Android apps. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018.* 257–268. https://doi.org/10.1145/3236024.3236045

[12] Facebook. 2023. Facebook Audience Network. https://developers.facebook.com/products/audience-network/ Accessed: 2023-05-25.

[13] flurry. 2023. *Manual Flurry Android SDK Integration.* https://developer.yahoo.com/flurry/docs/integrateflurry/android-manual/

[14] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. 2013. AdRob: examining the landscape and impact of android application plagiarism. In *The 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'13, Taipei, Taiwan, June 25-28, 2013.* 431–444. https://doi.org/10.1145/2462456.2464461

[15] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. 2017. CodeMatch: obfuscation won't conceal your repackaged app. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017.* 638–648. https://doi.org/10.1145/3106237.3106305

[16] Google. 2023. Android Apps on Google Play. https://play.google.com/store Accessed: 2023-05-25.

[17] Google. 2023. *d8.* https://developer.android.com/tools/d8

[18] Google. 2023. Mobile App Monetization - Google AdMob. https://admob.google.com/home/ Accessed: 2023-05-25.

[19] Google. 2023. Real-time Bidding | Google for Developers. https://developers.google.com/authorized-buyers/rtb/start Accessed: 2023-05-25.

[20] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks, WISEC 2012, Tucson, AZ, USA, April 16-18, 2012.* 101–112. https://doi.org/10.1145/2185448.2185464

[21] Ben Gruver. 2021. *smali/baksmali.* Retrieved March 3, 2021 from https://github.com/JesusFreke/smali

[22] Boyuan He, Haitao Xu, Ling Jin, Guanyu Guo, Yan Chen, and Guangyao Weng. 2018. An Investigation into Android In-App Ad Practice: Implications for App Developers. In *2018 IEEE Conference on Computer Communications, INFOCOM 2018, Honolulu, HI, USA, April 16-19, 2018.* 2465–2473. https://doi.org/10.1109/INFOCOM.2018.8486010

[23] Yongzhong He, Xuejun Yang, Binghui Hu, and Wei Wang. 2019. Dynamic privacy leakage analysis of Android third-party libraries. *J. Inf. Secur. Appl.* 46 (2019), 259–270. https://doi.org/10.1016/j.jisa.2019.03.014

[24] Jinlong Hu, Tenghui Li, Yi Zhuang, Song Huang, and Shoubin Dong. 2020. GFD: A Weighted Heterogeneous Graph Embedding Based Approach for Fraud Detection in Mobile Advertising. *Secur. Commun. Networks* 2020 (2020), 8810817:1–8810817:12. https://doi.org/10.1155/2020/8810817

[25] ImCaviar. 2023. *ANDetect.* https://sites.google.com/view/andetect

[26] Joongyum Kim, Junghwan Park, and Sooel Son. 2021. The Abuser Inside Apps: Finding the Culprit Committing Mobile Ad Fraud. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021.* https://www.ndss-symposium.org/ndss-paper/the-abuser-inside-apps-finding-the-culprit-committing-mobile-ad-fraud/

[27] Li Li, Tegawendé F. Bissyandé, and Jacques Klein. 2017. SimiDroid: Identifying and Explaining Similarities in Android Apps. In *2017 IEEE Trustcom/BigDataSE/ICESS, Sydney, Australia, August 1-4, 2017.* 136–143. https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.230

[28] Li Li, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. An Investigation into the Use of Common Libraries in Android Apps. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1.* 403–414. https://doi.org/10.1109/SANER.2016.52

[29] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: scalable and precise third-party library detection in android markets. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017.* 335–346. https://doi.org/10.1109/ICSE.2017.38

[30] Bin Liu, Hongxia Jin, and Ramesh Govindan. 2015. Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys 2015, Florence, Italy, May 19-22, 2015.* 89–103. https://doi.org/10.1145/2742647.2742668

[31] Tianming Liu, Haoyu Wang, Li Li, Xiapu Luo, Feng Dong, Yao Guo, Liu Wang, Tegawendé F. Bissyandé, and Jacques Klein. 2020. MadDroid: Characterizing and Detecting Devious Ad Contents for Android Apps. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020.* 1715–1726. https://doi.org/10.1145/3366423.3380242

[32] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: fast and accurate detection of third-party libraries in Android apps. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume.* 653–656. https://doi.org/10.1145/2889160.2889178

[33] ml.dmlc. 2023. *Xgboost4j.* https://mvnrepository.com/artifact/ml.dmlc/xgboost4j

[34] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. 2014. AdDetect: Automated detection of Android ad libraries using semantic analysis. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, April 21-24, 2014.* 1–6. https://doi.org/10.1109/ISSNIP.2014.6827639

[35] Karl Trygve Kalleberg Yotam Ole André Vadla Ravnås, David Weinstein and NSEcho. 2023. *Frida.* https://github.com/frida

[36] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David A. Wagner. 2012. AdDroid: privilege separation for applications and advertisers in Android. In *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012.* 71–72. https://doi.org/10.1145/2414456.2414498

[37] ProPrivacy. 2020. *Exposing the hidden data ecosystem of the UK's most trusted charities.* https://proprivacy.com/privacy-news/exposing-the-hidden-data-ecosystem-of-the-uks-most-trusted-charities

[38] Vaibhav Rastogi, Rui Shao, Yan Chen, Xiang Pan, Shihong Zou, and Ryan D. Riley. 2016. Are these Ads Safe: Detecting Hidden Attacks through the Mobile App-Web Interfaces. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016.* http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/ads-safe-detecting-hidden-attacks-through-mobile-app-web-interfaces.pdf

[39] G. Salton and Clement T. Yu. 1973. On the Construction of Effective Vocabularies for Information Retrieval. *SIGIR Forum* 9, 3 (nov 1973), 48–60. https://doi.org/10.1145/951761.951766

[40] shwenzhang. 2020. *AndResGuard.* https://github.com/shwenzhang/AndResGuard

[41] T.F. Smith and M.S. Waterman. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* 147, 1 (1981), 195–197. https://doi.org/10.1016/0022-2836(81)90087-5

[42] Statcounter. 2022. *Mobile vendor market share worldwide — statcounter global stats.* https://gs.statcounter.com/vendor-market-share/mobile/worldwide

[43] Android Statistics. 2022. *Ads.* https://support.google.com/googleplay/android-developer/answer/9857753

[44] Suibin Sun, Le Yu, Xiaokuan Zhang, Minhui Xue, Ren Zhou, Haojin Zhu, Shuang Hao, and Xiaodong Lin. 2021. Understanding and Detecting Mobile Ad Fraud Through the Lens of Invalid Traffic. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021.* 287–303. https://doi.org/10.1145/3460120.3484547

[45] Keen Sung, Jianyi Huang, Mark D. Corner, and Brian Neil Levine. 2020. Re-identification of mobile devices using real-time bidding advertising networks. In *MobiCom '20: The 26th Annual International Conference on Mobile Computing and*

*Networking, London, United Kingdom, September 21-25, 2020.* ACM, 48:1–48:13. https://doi.org/10.1145/3372224.3419205

[46] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. 2020. LibDX: A Cross-Platform and Accurate System to Detect Third-Party Libraries in Binary Code. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020.* 104–115. https://doi.org/10.1109/SANER48275.2020.9054845

[47] Hugues Thiebeauld. 2021. *Mobile App Shielding Market Intelligence.* https://eshard.com/posts/mobile-app-shielding

[48] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. WuKong: a scalable and accurate two-phase approach to Android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, Michal Young and Tao Xie (Eds.). ACM, 71–82. https://doi.org/10.1145/2771783.2771795

[49] Jice Wang, Yue Xiao, Xueqiang Wang, Yuhong Nan, Luyi Xing, Xiaojing Liao, Jinwei Dong, Nicolás Serrano, Haoran Lu, XiaoFeng Wang, and Yuqing Zhang. 2021. Understanding Malicious Cross-library Data Harvesting on Android. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021.* 4133–4150. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-jice

[50] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. 2018. ORLIS: obfuscation-resilient library detection for Android. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2018, Gothenburg, Sweden, May 27 - 28, 2018.* 13–23. https://doi.org/10.1145/3197231.3197248

[51] Jian Xu and Qianting Yuan. 2022. LibRoad: Rapid, Online, and Accurate Detection of TPLs on Android. *IEEE Trans. Mob. Comput.* 21, 1 (2022), 167–180. https://doi.org/10.1109/TMC.2020.3003336

[52] Xian Zhan, Tianming Liu, Yepang Liu, Yang Liu, Li Li, Haoyu Wang, and Xiapu Luo. 2022. A Systematic Assessment on Android Third-Party Library Detection Tools. *IEEE Trans. Software Eng.* 48, 11 (2022), 4249–4273. https://doi.org/10.1109/TSE.2021.3115506

[53] Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. 2019. LibID: reliable identification of obfuscated third-party Android libraries. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019.* 55–65. https://doi.org/10.1145/3293882.3330563

[54] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. 2018. Detecting third-party libraries in Android applications with high precision and recall. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018.* 141–152. https://doi.org/10.1109/SANER.2018.8330204

[55] Wu Zhou, Yajin Zhou, Michael C. Grace, Xuxian Jiang, and Shihong Zou. 2013. Fast, scalable detection of "Piggybacked" mobile applications. In *Third ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013.* 185–196. https://doi.org/10.1145/2435349.2435377

# 8 APPENDICES

## A DECOUPLING IN DETAILS

Assuming that both `class_A` and `class_B` are declared in the AP, we define a call relationship as a scenario where a method from `class_A` calls a method from `class_B` and define an inheritance relationship as `class_A` inheriting from `class_B`. Notably, we exclude classes defined in the Android SDK and JDK when constructing the community network to avoid the introduction of irrelevant noise.

Each node within the community is designated by the package name, under which all classes included in the node fall. Edges are drawn connecting one node to another, their direction dictated by the call and inheritance relationships between the classes from the two nodes. The weight of each edge is determined by the number of calls between all classes associated with the two nodes and the existence of an inheritance relationship between any two classes. Referring to the definition of node and edge in class profiling, We designate the weighted directed edge from node $N_i$ to node $N_j$ as $< N_i, N_j, W_{ij} >$, where $W_{ij}$ denotes the edge's weight. Assuming only `class_A` is contained in $N_i$ and `class_B` is contained in $N_j$, and `method_a` from `class_A` calls `method_b` from `class_B` once while `method_c` from `class_A` calls `method_b` twice, the edge from $N_i$ to $N_j$ would be represented as $< N_i, N_j, 3 >$. Extending this, if `class_A` inherits from `class_B`, the edge is represented as $< N_j, N_i, \rho >$, where $\rho$ signifies the weight of the inheritance relationship. To partition this community network, we utilize single-Louvain [7]. Given that the marked nodes in the community are primarily nodes containing class files - mostly leaf nodes in the package structure tree - we adopt a bottom-up merging strategy in accordance with the package structure tree. All nodes in the package structure tree are marked as communities before grouping nodes belonging to the same community into the same module for decoupling. The bottom-up merging strategy follows these rules:

- If the node to be merged has a parent node and the parent node is marked, the community number of the node to be merged is the same as that of the parent node.
- If neither the node to be merged nor its parent node is marked, the community of the node to be merged is determined by the community of its child nodes. If the child nodes belong to the same community, the node to be merged is also marked as the community. Once the child nodes belong to different communities, take the community with the highest probability of occurrence and greater than $\sigma$ as the result. If there is no community whose occurrence probability is greater than $\sigma$, it is considered that the node does not belong to any community and stop merging the branch where the node belongs to. Here, we define $c$ as the set of communities which child nodes belong to and the definition of $\sigma$ is as follows:
$$\sigma = \frac{1}{|c|} \tag{6}$$
- Starting from the leaf node, merge along the parent node to the root node. Until the communities to which all nodes belong no longer change, nodes belonging to the same community are merged as sub-module according to the package structure tree. If all nodes of the same community form a sub-tree, it is regarded as a sub-module. If the nodes of the same community come from different branches of the tree, a new node is created as the common parent of all branches, constituting the sub-module.

## B FEATURES IN RESOURCE PROFILE

The selection of the keys in resources profiling should satisfy the universality, differentiation and insensitivity. Details about how these metrics are calculated are given in the following equations.

We define the universality of *key* $k$ as $U_k$ in which $K$ represents the *key* set of an `aar` file and $l$ represents total of all `aar` files:

$$U_k = \frac{\sum_{i=1}^{l} In_{k,K_i}}{l}, \quad In_{k,K_i} = \begin{cases} 1, & k \in K_i \\ 0, & k \notin K_i \end{cases} \tag{7}$$

We utilize the TF-IDF algorithm [39] to compute the differentiation of each *key*. Initially, we fragment each *value* into individual elements based on punctuation. For instance, "`com.yandex.mobile.ads`" is divided into "`com`", "`yandex`", "`mobile`" and "`ads`". It means that every *value* is regarded as a set of elements. Subsequently, we construct a corpus from these elements. The differentiation $d_{v,i}$ of *value* $v$ in the i-th `aar` is given by the max differentiation of element $e$ belonging to it, where $f_{e,i}$ symbolizes the frequency of $e$ in the i-th `aar` and $\boldsymbol{e}_i$ represents all elements in the i-th `aar`. We denote $M_{k,i}$ as a set of *values* mapped from *key* $k$ in the i-th `aar` and the

**Table 5: The key, universality, differentiation and insensitivity of each feature.**

| Feature | Key | $U_k$ | $D_k$ | $IS_k$ |
|---|---|---|---|---|
| $F_1$ | AndroidManifest.xml_manifest_package | 1.0000 | 0.7294 | 0.7913 |
| $F_2$ | AndroidManifest.xml_manifest uses-permission_android:name | 0.5385 | 0.1122 | 0.3215 |
| $F_3$ | AndroidManifest.xml_manifest application provider_android:authorities | 0.1186 | 0.0517 | 0.7953 |
| $F_4$ | AndroidManifest.xml_manifest application receiver intent-filter action_android:name | 0.1602 | 0.0648 | 0.7222 |
| $F_5$ | AndroidManifest.xml_manifest application activity_android:name | 0.3960 | 0.4886 | 0.4251 |
| $F_6$ | AndroidManifest.xml_manifest application service_android:name | 0.1609 | 0.3541 | 0.5326 |
| $F_7$ | AndroidManifest.xml_manifest application receiver_android:name | 0.1943 | 0.3261 | 0.6667 |
| $F_8$ | res\values\values.xml_resources dimen_name | 0.1350 | 0.1666 | 0.1018 |
| $F_9$ | res\values\values.xml_resources string_name | 0.3770 | 0.2886 | 0.2112 |
| $F_{10}$ | res\values\values.xml_resources declare-styleable attr_name | 0.1302 | 0.3541 | 0.1628 |
| $F_{11}$ | res\values\values.xml_resources style_name | 0.2529 | 0.1062 | 0.3273 |

final differentiation of $k$ symbolized as $D_k$ is given by the average differentiation of *values* in $M_{k,i}$:

$$d_{v,i} = \max_{e \in v}\{\frac{f_{e,i}}{\sum_{e' \in \boldsymbol{e}_i} f_{e',i}} \times \ln \frac{l}{1 + \sum_{j=1}^{l} In_{e,\boldsymbol{e}_j}}\} \tag{8}$$

$$D_k = \frac{\sum_{i=1}^{l} \sum_{v \in M_{k,i}} d_{v,i}}{\sum_{i=1}^{l} |M_{k,i}|} \tag{9}$$

We evaluate the sensitivity of all *keys* to different versions of the same ANs. We define $an_k$ as the set of all ANs who own libraries of different versions and contain *key* $k$ in resource files. The i-th AN in $an_k$ is represented as $an_{k,i}$. We traverse every AL in $an_{k,i}$, find the *values* of $k$ and form the set $v_{k,i}$. The insensitivity of $k$ is defined as $IS_k$:

$$IS_k = \frac{\sum_{i=1}^{|an_k|} \frac{1}{|v_{k,i}|}}{|an_k|} \tag{10}$$

## C COMPREHENSIVE SCORING IN DETAILS

The following methods are utilized to score the features mentioned in Table 5. For different features, we use different methods depending on how they are presented. The score of $F_1$ is given by global search. The scores of $F_2$ to $F_4$ is generated by full-path limitation method.

The scoring result of $F_1$ in the global search is denoted as $Score_g$:

$$Score_g = \begin{cases} 1, \exists\, v \in V_1,\ s.t. \text{ Prefix of } v = F_1 \\ 0, \forall\, v \in V_1,\ s.t. \text{ Prefix of } v \neq F_1 \end{cases} \tag{11}$$

For stable features $F_2$ to $F_4$ in full-path limitation, complete matches are required for all values if a feature contains multiple values. To constrain the score within the range of 0 to 1, we define the activation function as follows:

$$f(x) = \begin{cases} 0, & x = 0 \\ \frac{1}{1+e^{1-x}}, & x \geq 1 \end{cases} \tag{12}$$

The scoring result of stable features is denoted as $Score_s$:

$$Score_s^n = f(\sum_{i \in F_n} s_i),\ \text{where } n \in \{2, 3, 4\} \tag{13}$$

$$s_i = \begin{cases} 1, & \exists\, v \in V_n,\ s.t.\ v = i \\ 0, & \forall\, v \in V_n,\ s.t.\ v \neq i \end{cases}$$

**Table 6: The final hyperparameters of advertising behaviors detection model based on different feature sets.**

| Feature Set | Boost Round | Learning rate |
|---|---|---|
| API calls | 150 | 0.05 |
| distribution across classes | 250 | 0.025 |
| distribution across methods | 100 | 0.1 |

For variable features $F_5$ to $F_7$ in full-path method, we define the relaxation similarity between $V_n$ and $F_n$ as $RS_n$ and denote the scoring result as $Score_v$. And the details about how these metrics were generated

$$Score_v^n = \frac{RS_n}{|F_n|}, \text{where } n \in \{5, 6, 7\} \tag{14}$$

The weight of each value in the feature $F_8$ to $F_{11}$ with no-path method is given by equation (8). The score of each feature is denoted as $Score_o$:

$$Score_o^n = \frac{\sum_{i \in F_n} s_i}{|F_n|},\ \text{where } n \in \{8, 9, 10, 11\} \tag{15}$$

$$s_i = \begin{cases} d_i, & \exists\, v \in V_n,\ s.t.\ v = i \\ 0, & \forall\, v \in V_n,\ s.t.\ v \neq i \end{cases}$$

In the image pool method, we define $PN$ as the collection of all image names present in the ad library and $V_p$ as the equivalent set in the Android application. The score, represented as $Score_r$, adheres to the scoring procedure outlined in equation (13).

$$Score_r = f(\sum_{i \in PN} s_i),\ s_i = \begin{cases} 1,\ \exists\, v \in V_i,\ s.t.\ v = i \\ 0,\ \forall\, v \in V_i,\ s.t.\ v \neq i \end{cases} \tag{16}$$

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

**Table 7: The brand-new adware detected by ANDetect. The sha256 of each adware and malicious ALs it contains are provided.**

| sha256 | Malicious ALs |
|---|---|
| 0002CA163461C26D3EA1D7384DB1833D4002CA679D0BA1906494E1BB24A95896 | com.mopub.mobileads |
| 00120DAD331324ED394118079BCB91D2B3F69C42E3C5135303DA50272FAE45F7 | com.inmobi.ads;com.mopub.mobileads |
| 00154456B10D6BF1F16DA14267D6063DAABEC20FFD28ABAB942D7BBD266D430F | com.mopub.mobileads |
| 0134CE0FC74B7DDD800CF772A1742F7966FA4D5762359D2E6D15BED8728CC779 | com.mopub.mobileads |
| 0120397B85BFF474D554F82BEC39A5AB9827F93B68FD505AF89B935497860BAE | com.mopub.mobileads |
| 0137F1EBE1F012BF454F78FE2452C1D39E0105AD0A00AC9E90F799EC96D09406 | com.mopub.mobileads |
| 7C90D85F5E08639CA32A5473CBDBE1B60334634B5CDB5AA7274FFF7A3B7B9BEA | com.mopub.mobileads |
| 7C98A973971958D54D69D04C504A14EE8BF9CD709882BA544743F6D272C18955 | com.mopub.mobileads |
| 7CC341B217FA38C3C12BFA008166F6F7229ABE3CB765D4AD5455925D5EB20632 | com.mopub.mobileads |
| 7CE87F293FDDCA68C5786976B7F9AF9CF37D2598B383424086066BF41814372F | com.mopub.mobileads |
| 7CF7F9D84FEA8DE117088FDAA4A3148DE301AFB21EA64BF3DEF244F93FF1B637 | com.mopub.mobileads |
| 7CD48EA847234C09C2D5EB60C8950005AE38674DDE24D9F5EBBD6043D533A10D | com.mopub.mobileads |
| 7CC61FA9A5E041957CFA9F5434E540624988D6E139DCC71D85348F793AFEA475 | com.mopub.mobileads |
| A4E9D3649543CF6246EEA45BFE15894C2C0430FDB4CA2787B3DEF2CECDF84521 | com.mopub.mobileads |
| A4FD6B5E2E6F0FE0CBE89230FA9122647FF22D2DB814C247B3F8532614E347B0 | com.mopub.mobileads |
| A552F35988EE74751A650C5AEB74FB1B2EEFD1231851D19AD23DB91923BBC7A8 | com.mopub.mobileads |
| A56A45FBA78F4E785593F747E6FE9A7F3BFF7A94EAF1D08428D67959629C538F | com.inmobi.ads;com.mopub.mobileads |
| A58772EAAA85CB507003199F0129AB918BCEB6120CDD202C6CE4EBAC52A6C5BE | com.mopub.mobileads |
| A5A46FDC7879DC0210ECEF7ECC29D2369A1B00165BBAA668115B6BD863AF600C | com.mopub.mobileads |
| A5CAFB80BCF17912772C72C3DAB41D439B8EA06DD42A174A01C6864491F1F22D | com.mopub.mobileads |
| A5C2D903596DA9A8880BA97B3B9A9783391B1552D3B605FFF9134C3ED4B82E0E | com.mopub.mobileads |
| B33A5CD6AF62E8865DD7C2127CDDB9B96813AE6DE365EC6F0DF135E127C35BE6 | com.inmobi.ads |
| B33949FA901C3227FD46416201AB2E219AEE15C804A8471D8213551E44E1C33A | com.mopub.mobileads |
| B34E6C6178B1FC6A3EAA4BFF12EB9BB6493F7F44E39BAA8ED44C91890BC92B04 | com.mopub.mobileads |
| B3548B219682E7E739E32DE22B35BA1997EF3B6C74D1E7C6218F907059A29771 | com.mopub.mobileads |
| B3C2ED36A1CBC3E921DAC8072E3E907A4E82E25FF223133963A3872A15A9B4DB | com.mopub.mobileads;com.inmobi.ads |
| B40C89446F073C6DCE610714181E4FB891342144E367EF05AE8A0D671ABC520F | com.mopub.mobileads |
| B3E8A1BA25B18226E542814BB066DF89364A09FC8021CA70D5698210A088DF4C | com.mopub.mobileads |
| B3EE48851086FE4F66186AF69461FD8A559D4AC87E8596AA2DF06F877F28005A | com.mopub.mobileads |
| B40C0F97811AE225EAE17BC8DE6ED3188A2F7219652A495ED161C23F35FEE55A | com.mopub.mobileads |
| B4215B5FF02FDC76788062ED7C1CAA69A2C7CCD5B4031E9F51DAC5CB7DEFFEC2 | com.mopub.mobileads |
| B42CD94A973F1A2CC48B324EFF9F7E7648EA01C9B6FA11489EBBC6BA2754A855 | com.mopub.mobileads |
| 7D042F1D52DD9BF078B04353340F63753A14A62D5FFE838AA4B36AF7EAB27E9D | com.inmobi.ads;com.mopub.mobileads |
| 7CE4CB61A91D3A960C14D996E24729965ACCE2BD465B9941C2479186ADF2D170 | com.mopub.mobileads;com.inmobi.ads |
| 7CF0724873BC4464FCF0D8F99D675DCB22F1CBA97017BD8545E44FDDAAE0CD52 | com.mopub.mobileads |
| 7D39817880B7ACAAFA7A05B6864D30CC89CD4DAC16AE64235FDD15DAA82178DE | com.inmobi.ads |
| 7D1247A2391A9A9FB08C1F77682ED16AE8885C1D10385A1D7D7B788AF0A2A2A5 | com.inmobi.ads |
| 10856950A97C189B5458B537BE2DB5555202A50B15345E97F1D733DA97EE21DF | com.mopub.mobileads |
| A5EBC5A1052546C781513698F8BF1E6ADF2F87B0AD203C8282C07F93C4F68F1D | com.mopub.mobileads |
| A5DCBD35B67D47BCA804225A34DCFF3E2AFBD7544BBEFDAB54BBCD64D19A4125 | com.mopub.mobileads |
| A5F5AB473F93203D78F078CAC4580561609FB40D35A39CB5B78DEDCDFDBCE843 | com.mopub.mobileads |
| A624B80E79BDA4535298DB9A6B0BDC6A265BDA521926079687E4C79BDD325CE0 | com.mopub.mobileads |
| B4744416B37E89D94B865BACF3CF8D76CF23B0504F571C89A1E4AEA38A2F7C6D | com.mopub.mobileads |
| 11F0B6F0FC2BE6F5826D53EA5685307201FEDAED7FEB789123D2C07E03D03B68 | com.mopub.mobileads |
| B4BBC4CB843C4501E79DEA682425AED436C2168A1EAC057099E0A8A2A1357729 | com.inmobi.ads;com.mopub.mobileads |
| 11EC680264465AA2E07D0E46DE62E6CDB449208BE53FB281DF39FE5E944D43F8 | com.mopub.mobileads |
| B527EF6798ECA6FBD4C1A43FBB7BACCAC1ACDCEB144F504C314E91B5A7589939 | com.mopub.mobileads |
| B436E30ADFFC7C4FD75FF38FC91014A181978348ADB6B04F2B34389CF6DA5926 | com.mopub.mobileads |
| B491A9266968E3687CE258D0163A7B458D2B0DB5207FD843CD5C71CFAC63BA4C | com.mopub.mobileads;com.inmobi.ads |
| B3FEE4445B1D5F1D7D45FA98E0DF32FBB74ABFE8572E413B0EEE1E5767FB4A4D | com.mopub.mobileads |
| 7CE6103DFC87DC72EB0BDFAAAC872467E7A893C6007D7C378CED6D8F3584F462 | com.mopub.mobileads |
| 7C9C220C986580B07A9A2E2241349E75F8F91FB310CF2F87EB9A100896C4FF22 | com.mopub.mobileads |
| B43E0E61180E5CD03590CFC172D4627F0E321D4A58ED538901ABCE7023968657 | com.mopub.mobileads |