Following Devil's Footprints: Cross-Platform Analysis of Potentially Harmful Libraries on Android and iOS

Kai Chen[1], Xueqiang Wang[2], Yi Chen[1], Peng Wang[2], Yeonjoon Lee[2], XiaoFeng Wang[2]
Bin Ma[1], Aohui Wang[1], Yingjun Zhang[3], Wei Zou[1]
{chenkai, chenyi, mabin, wangaohui, zouwei}@iie.ac.cn, {xw48, pw7, yl52, xw7}@indiana.edu, yjzhang@tca.iscas.ac.cn
[1]State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
[2]Indiana University, Bloomington
[3]Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences

***Abstract*—It is reported recently that legitimate libraries are repackaged for propagating malware. An in-depth analysis of such potentially-harmful libraries (*PhaLibs*) , however, has never been done before, due to the challenges in identifying those libraries whose code can be unavailable online (e.g., removed from the public repositories, spreading underground, etc.). Particularly, for an iOS app, the library it integrates cannot be trivially recovered from its binary code and cannot be analyzed by any publicly available anti-virus (AV) systems. In this paper, we report the first systematic study on PhaLibs across Android and iOS, based upon a key observation that many iOS libraries have Android versions that can potentially be used to understand their behaviors and the relations between the libraries on both sides. To this end, we utilize a methodology that first clusters similar packages from a large number of popular Android apps to identify libraries, and strategically analyze them using AV systems to find PhaLibs. Those libraries are then used to search for their iOS counterparts within Apple apps based upon the invariant features shared cross platforms. On each discovered iOS PhaLib, our approach further identifies its suspicious behaviors that also appear on its Android version and uses the AV system on the Android side to confirm that it is indeed potentially harmful. Running our methodology on 1.3 million Android apps and 140,000 popular iOS apps downloaded from 8 markets, we discovered 117 PhaLibs with 1008 variations on Android and 23 PhaLibs with 706 variations on iOS. Altogether, the Android PhaLibs is found to infect 6.84% of Google Play apps and the iOS libraries are embedded within thousands of iOS apps, 2.94% among those from the official Apple App Store. Looking into the behaviors of the PhaLibs, not only do we discover the recently reported suspicious iOS libraries such as mobiSage, but also their Android counterparts and 6 other back-door libraries never known before. Those libraries are found to contain risky behaviors such as reading from their host apps' keychain, stealthily recording audio and video and even attempting to make phone calls. Our research shows that most Android-side harmful behaviors have been preserved on their corresponding iOS libraries, and further identifies new evidence about libraries repackaging for harmful code propagations on both sides.**

I. Introduction

The prosperity of mobile ecosystems is powered by highly dynamic and ever-expanding markets of mobile applications (*app* for short), which are playing increasingly important roles in our daily life, from entertainment, social networking to serious businesses like finance, health care and home security. Behind such valuable services, however, there could be less than legitimate or even sinister activities, which may cause harm to mobile users. Examples include transferring private user information such as her precise locations and IMEI to unauthorized recipients, sending SMS messages unrelated to an app's functionalities, exploiting known vulnerabilities, installing back-doors, etc. The apps exhibiting such behavior is called *potentially harmful app* (*PHA*), a term Google uses to replace the undefined term "malware" for describing "applications which pose a security risk to users or their data" [1]. A recent study shows that such dangerous activities are found in 7% apps on Google Play [2], most of which are detected by mainstream anti-virus (AV) scanners integrated within VirusTotal. In our research, we consider an app to be a PHA when it acts in a way that can cause potential damage to the user's information assets (as described above). A close look at such apps reveals that the sources of their potentially harmful behavior, oftentimes, are the libraries shared across the apps [2]. Similar observations have also been made on iOS, a platform widely thought to be mostly PHA free: it has been reported recently that iOS apps are infected with malicious code, which comes from either unwitting use of untrusted versions of popular libraries [3] or the methods injected by contaminated XCode toolkits [4].

Challenges in PhaLib analysis. Indeed, given the way that today's apps are developed, which are often built by extensively reusing existing code, it is conceivable that *potentially harmful libraries* (*PhaLib*) could feature prominently in mobile PHA, constituting an important channel for spreading infections when popular legitimate libraries are contaminated. However, a systematic analysis on PhaLibs has never been done before, possibly due to its technical challenges. Specifically, for a legitimate library, what can be found online are just its most recent versions, even when most of its older versions are still in use within a large number of apps. An example is `airpush`, a library we found to have 12 versions distributed across 1,650 popular apps. When it comes to PhaLibs, the situation becomes even more complicated: contaminated libraries are scattered across a variety of sources like public code repositories (e.g., GitHub), online forums, etc. and come and go quickly;

DOI 10.1109/SP.2016.29

dedicated malicious libraries are shared among PHA authors and difficult to come by. The attempt to recover them from apps is nontrivial, due to the presence of different versions of a PhaLib and the customizations made by the app developer. More challenging is the study on iOS PhaLibs: the library integrated into an iOS app is broken down into methods scattered across its binary, which is much more difficult to identify than an Android library that typically stays in a package; further unlike Android for which there are many AV systems for detecting potentially harmful code and behavior, up to our knowledge, no public system exists for finding iOS PHAs. In the absence of such a system, validating the findings of a PHA analysis becomes difficult, as there is no ground truth to confirm that the PhaLibs discovered are indeed harmful.

Cross-platform study. In this paper, we report the first cross-platform analysis on PhaLibs, over 1.3 million Android apps and 140,000 iOS apps, an unprecedented scale compared with all existing research on Android and iOS PHAs. The study is made possible by a methodology designed to overcome the aforementioned technical barriers. More specifically, using a recent technique for scalable comparison of Android methods [5], our approach is able to find similar methods shared by different packages across over a million apps. Clustering the packages within the apps according to their names and code similarity helps us discover 763 libraries and their 4,912 variations. These libraries are then extracted and scanned by VirusTotal to find out those suspicious and their potentially harmful behaviors are further analyzed.

A key idea of our methodology is to leverage the relations between Android and iOS libraries for a cross-platform PhaLib analysis. The interesting thing here is that a significant portion of third-party services to Apple devices are also provided to Android users through libraries: e.g., among the top 38 iOS libraries reported by SourceDNA, 36 have Android versions (see appendix); this enables us to identify and analyze a subset of iOS libraries by leveraging the features they share with their Android counterparts. Our study shows that even though related iOS and Android libraries can be developed independently, the relations between them can still be established using the invariants across the platforms, particular, the constant strings they share such as the URLs for accessing external resources, and the ways the classes involving those strings are connected to other classes (e.g., through method invocation, see Section III-C). Based upon such invariants, we are able to find an iOS PhaLib cross-platform by inspecting iOS apps for the invariants recovered from its Android version and correlated their behavior sequences considered to be potentially harmful by leading AV scanners: our technique detects common action sequences within the Android and iOS libraries of the same service and confirms that they are part of the signatures the scanners use to catch Android PHAs. This enables us to validate reported harmful behavior within iOS apps, when similar activities within Android apps are deemed problematic.

Our findings. Running the methodology over our Android, iOS app sets, we discovered 117 Android PhaLibs (with 1008 variations), which were further mapped to 46 iOS libraries. We manually confirmed that all of them are indeed libraries and 23 of them (706 variations) are potentially harmful. These PhaLibs are found within 98,308 Android apps, and 2,844 iOS apps on the official Apple App Store and 3,998 apps on the third-party Apple stores, including those in North America, Asia and Europe. Altogether 2.94% of the iOS apps (based upon our random samples) on the Apple Store are considered to contain suspicious code, which is surprising given the common belief that the official iOS market is well guarded and unlikely to have PHAs. A further study on the discovered PhaLibs brings to light not only the recently reported iOS PhaLib mobiSage but also its Android version, together with other PhaLib back-doors (adwo, leadbolt, admogo, etc.) never known before. Interesting behaviors discovered include stealthy audio and video recording and picture taking, keychain access within the advertising libraries, command and control, making call attempts, disclosing app list etc., on both Android and iOS.

Further we found that on both platforms, some versions of popular libraries contain the suspicious action sequences their official versions do not have, indicating possible repackaging of such libraries by the adversary to propagate malicious payloads. Particularly, on the Android side, we found that 8 popular libraries, including mappn, jpush, swiftp and etc., have all been repackaged, with the apps using the potentially harmful versions discovered on third-party app markets. Particularly, the contaminated version of a popular Chinese app-market library, mappn was found on GitHub. Also our study shows that within iOS apps, the potentially harmful actions corresponding to those observed in their Android counterparts are often executed through private APIs. Of particular interest here is the strategy both Android and iOS PhaLibs take to perform the operations that need the user's consent, such as collecting precise locations: they typically avoid calling the APIs that need the approval from the user, such as `requestWhenInUseAuthorization`, and instead, read last retrieved location data from the hosting app in background; in other words, the PhaLib is designed to leverage the consent an app already gets from the user (for its legitimate functionality) to execute potentially harmful actions. We are communicating with Apple, Google and other app vendors to report our findings and helping them analyze the apps involving the PhaLibs we discovered. The video demos and other materials related to the research are posted on a private website [6].

Contributions. The contributions of the paper are summarized as follows:

• *Cross-platform study on PhaLibs*. We conducted the first systematic study on potentially harmful libraries, over both Android and iOS. The study is made possible by a suite of innovative techniques, including automatic identification of libraries from Android apps, mapping Android libraries to the code components within iOS apps and determining suspicious action sequences cross-platform. These techniques were evaluated over 1.3 million mainstream Android apps and over 140,000 iOS apps from the official App Store and various

third-party markets, a scale that has never been achieved in the related prior research.

• *New findings*. Our study leads to surprising discoveries about the pervasiveness of Android and iOS PHAs, the critical roles played PhaLibs in these apps, new suspicious activities, contamination of legitimate libraries for spreading potentially harmful code and the unique strategy taken by iOS PhaLibs to remain low-profile. Also interesting is the new understanding about the relations between Android and iOS libraries, which could lead to new techniques for effective detection of PHAs on these platforms, particularly suspicious iOS apps, which have never been systematically investigated before.

II. Background

Mobile libraries. A mobile library is a collection of non-volatile resources (including subroutines, classes and data) that provides a set of functionalities (taking pictures, setting up an SSL connection, etc.) the developers can conveniently integrate into their program. On Android, such a library is typically included in a *package*. A library encapsulates the functionalities it serves with a set of well-defined Application Programming Interfaces (APIs), through which one can easily acquire the service. With such convenience, a vast majority of apps today are built upon different libraries. Examples of popular libraries include unity3d, crashlytics and inMobi for Android and flurry, openfeint and bugsnag for iOS. Also libraries are utilized by advertisers to deliver advertisements (ad) from their servers to the mobile users and collect the users' information useful to targeted advertising.

Although some libraries are maintained by their developers on their official websites, many others are scattered across a variety of sources like public code repositories (e.g., *GitHub*), online forums, etc. Even for those well maintained, typically only their most recent versions are available, which those obsolete ones are most likely still in use within different apps. As a result, a comprehensive analysis of *active* mobile libraries, those still used by popular apps, is nontrivial. In our study, we recovered such libraries from Android and iOS apps through clustering their components using a similar code comparison technique (Section III-B).

Once a potentially harmful or contaminated library is published through online repositories or forums, it can reach a large number of app developers, who by using the PhaLib, unwittingly include harmful code in their programs. The most prominent event related to this threat is the recent XCodeGhost attack, in which XCode, the Apple's programming environment, was repackaged with potentially harmful code and uploaded to online repositories, and all the apps built with the contaminated XCode, including some leading apps like WeChat, were all found to contain potentially harmful libraries [4]. Note that the propagation of potentially harmful code in the attack does not go through shared libraries directly: instead the infection is passed on to apps by the compromised programming environment. In our research, however, new evidence is provided that the library repackaging attack is indeed present on both Android and iOS.

Mobile PHA detection. Just like traditional desktop systems, Android is also known to be plagued by PHAs. It is reported that potentially harmful apps exist on the Android official market [7] and are rather pervasive on third-party markets [2]. Also prior research shows that such PHAs are mainly introduced through repackaging legitimate apps (e.g., AngryBird), which enables the PHA to free-ride the legitimate app's popularity to reach a large audience. What has been less clear are other avenues the PHA authors can exploit to spread potentially harmful code, particulary repackaging shared libraries, which was studied in our research. Different from Android, iOS is less susceptible to PHA infection, thanks to its more restrictive security control and app vetting process. However, the recent XCodeGhost attack shows that contaminated shared resources could be a realistic threat to iOS security.

Major Anti-Virus (AV) companies are moving towards the mobile market, providing new services to detect mobile PHAs. Particularly, the public AV platform VirusTotal has integrated 54 AV scanners, including the products of all leading AV companies such as Symantec, McAFee, Kaspersky, etc., which all work on Android apps. Those scanners typically inspect the disassembled code of an app, looking for the signatures of known harmful behavior. Also app markets can deploy their own PHA detection mechanisms, such as Google's Bouncer [8].

However, similar AV services do not exist on iOS. Apple is known to be less supportive to third-party AV products [9], [10]. Technically, PHA detection on the Apple platform is hard due to the encryption protection on apps: an app downloaded from the Apple Store is encrypted with keys and analyzing its code needs to first decrypt the app, which cannot be done without the right key. The problem is that such a key cannot be accessed by the user without jail-breaking her device. Therefore, in the absence of the help from Apple, it becomes very difficult for an ordinary user to decrypt the app she installs for an AV scan. In our study, we manually checked more than ten popular online AV services and found that none of them provides a comprehensive PHA detection service for iOS. For example, VirusTotal, the most famous AV platform, only reports the metadata of an iOS app such as configuration information [11]. The only PHA we found it is capable of detecting is "Find and Call", which is known to the public in 2012 [12]. Neither can other services such as VirSCAN [13] capture harmful iOS code. Such a lack of public AV services is also caused by the belief that iOS PHAs are rare: there were just 4 iOS targeted attacks in 2014, compared to 1268 known families of Android PHA this year [14]. Our research, however, reveals that actually a large number of apps on the Apple markets are involved in the activities considered to be potentially harmful when they are performed by Android apps (Section IV).

Code-similarity comparison. To recover the libraries already integrated into an app's code, we have to compare code components (in terms of methods) across a large number of apps (over a million for Android). Such comparison needs to be scalable, accurate and also capable of tolerating some differences between the components, which widely exist due to

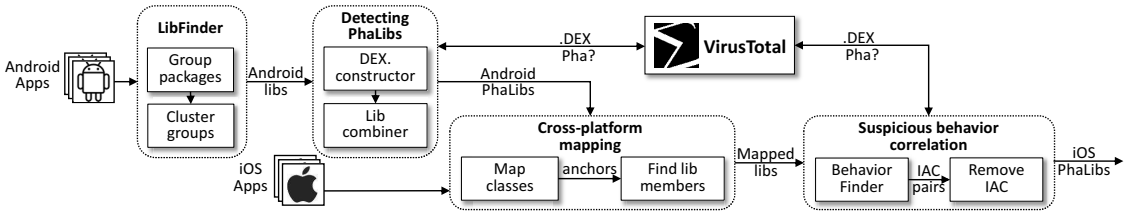

Fig. 1: *Overview of our approach.*

the variations of the same library (different official, customized versions). To this end, we utilized a recently proposed technique called *Centroid* [5] in our research, which extracts a set of features (e.g., loops, branches, etc.) from an app's control-flow graph (CFG), uses such features to convert the program into a high-dimension object (with each feature as a dimension) and then maps the whole program into the geometric center (the centroid) of the object. The centroid, which is a concrete value, is characterized by a monotonicity property: for two program components with similar centroids, their CFGs also come close; for those unrelated to each other, their centroids are also very different. This approach localizes the global comparison across the whole market to a small number of "neighbors", which allows high scalability and accuracy to be achieved at the same time [5]. More specifically, a code component can be easily compared with millions of other components through a similar binary search over their centroids.

Adversary model. We consider the adversary who spreads potentially harmful code through repackaging legitimate Android or iOS libraries, or through distributing dedicated PhaLibs for PHA authors to build attack payloads. As a first step, we only studied the libraries that have not been obfuscated within the app code. Also for the cross-platform analysis, we have to focus on the PhaLibs with both Android and iOS versions. It is important to note that we *did not* make any assumption on what the PHA authors cannot do. Instead, our study is meant to improve our understanding of the scope and magnitude of this type of PHA infections and techniques the adversary uses, by investigating a subset of Android and iOS PhaLibs.

III. Methodology

A. Overview

Idea and key techniques. As mentioned earlier, a study on mobile PhaLibs needs to discover hidden libraries integrated within apps and determine whether they are indeed suspicious, which is nontrivial on Android and even more so on iOS, due to the lack of the ground truth (AV detecting systems for validating whether a library is indeed suspicious). Our solutions are a suite of techniques enabling a unique analysis procedure that correlates Android PhaLibs to their iOS counterparts, using the resources on the Android side to study the suspicious behavior of iOS apps. This procedure is illustrated in Figure 1 and the key techniques involved are highlighted as follows: on Android (Section III-B) (1) hidden library discovery, (2) PhaLib detection, and on iOS (Section III-C) (3) cross-platform mapping of PhaLibs and (4) suspicious behavior correlation.

Specifically, the first step on the Android side is to cluster the packages recovered from the code of over a million apps (including 400,000 from Google Play) to identify "libraries", that is, those extensively reused across apps. The libraries are then extracted and scanned by VirusTotal to detect PhaLibs from them or their variations. After that, the invariants (e.g., constant URL strings) collected from individual PhaLib are utilized to analyze over 140,000 iOS apps, for the purpose of finding related iOS PhaLibs from the apps. Finally, suspicious behavior (in terms of invariant-API-category sequences) in the iOS PhaLibs is identified by correlating each iOS invariant-API-category sequence to the one within the related Android PhaLib, so that it can be confirmed by existing AV systems to be potentially harmful. All these steps are automated. However, we do need manual effort to build a dictionary for the mappings between Android and iOS APIs (Section III-C). Note that this only needs to be done once when the OS for iOS or Android is updated.

Example. Figure 2 presents an example that describes how our methodology works. As we can see from Figure 2-A, the packages found within 13 Android apps share over 65% of methods and are therefore considered to be the variations of a library. They are extracted from the hosting apps and scanned by VirusTotal to detect PhaLibs. A confirmed Android PhaLib is illustrated in Figure 2-B, whose URL sequence is considered stable across platforms (see the figure). Such a stable invariant can also be found from a related iOS Phalib (Figure 2-C), even though it is built upon a different programming language. Further from those PhaLibs, we can identify their corresponding invariant-API-category sequences. The Android-side sequence turns out to be part of the signatures some AV scanners use to detect PHAs, which indicates that the related behavior on the iOS side is also suspicious. In the measurement study we performed using the methodology, the correlations between the PhaLibs on different platforms and their shared behavior were all automatically detected and then manually validated, which confirmed that almost all of them were accurate.

B. Finding PhaLibs on Android

Our analysis on mobile PhaLibs starts from the Android end, since Android apps are easier to study (unencrypted, relatively small size and availability of the ground truth, i.e., VirusTotal) than the programs on iOS. What we did first is to identify the libraries from the code of over a million apps and run VirusTotal to detect those considered to be potentially harmful. This approach could miss the PhaLibs whose behavior has not

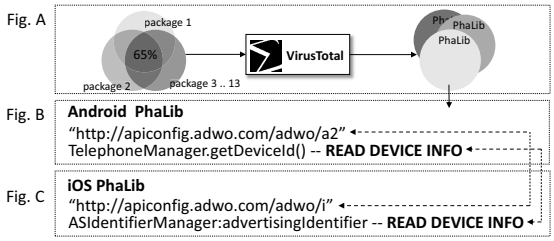

Fig. 2: *An example showing how our methodology works.*

been known (and therefore cannot be found by VirusTotal), but simply attributing known harmful behavior to libraries and analyzing their relations already helps us gain a better understanding about a few important issues never studied before, for example, what is the role a PhaLib plays in potentially harmful activities, how the libraries are exploited as an avenue to propagate harmful code, etc. Following we elaborate our approach (see Figure 1).

Grouping packages. As introduced earlier, an Android library is in the form of a Java package, which implements a set of functionalities and provides services to its hosting app (Section II). It can be built by any developer, who only needs to pack her code and make it available for sharing. There are tens of popular libraries available online [15], which however are only a tip of the iceberg. Many libraries in the wild are only circulated within a small group of people. Examples include those used within an organization and the attack toolkits available to the hacker community. Even for the popular libraries, they tend to have multiple versions introduced by updates and the needs for serving different devices (phone, tablet, etc.) and different markets (North America, Europe, Asia, etc.). As a result, finding these libraries online is highly difficult, and in some cases, even impossible (e.g., an older version that has been replaced with the new one).

Given the challenges in collecting libraries online and tracking their version changes, we have to look at the apps, the ultimate source of libraries, and recover them from the app code. The advantage of doing this is that whatever we found must be the libraries that are still "alive", being used by some apps. The security threats discovered from those apps will have real-world impacts. Specifically, we leverage an observation: a library is typically used as a whole piece and also named and structured in the standard way [16]; also in most cases, different instances of the same library carry the same package name prefix, at least the top domain and organization's domain (e.g., com.android), when they are integrated within different apps. What we can do here is to automatically analyze the disassembled code of Android apps, breaking them into packages and grouping these packages according to their names as appear within their hosting apps. All the packages in the same group are further inspected to find out whether they share a lot of code among them. Those indeed are and also used by the apps from more than 20 different vendors (determined by looking at the certificates used to sign apps, as prior research does [2]) are reported as libraries. This approach helps us avoid the pair-wise code comparison across a large number of apps, which is very computation-intensive. Also note that though grouping packages by their names could miss some potentially harmful libraries that have been obfuscated, this treatment is simple and effective at identifying many PhaLib instances, particulary for the legitimate libraries contaminated with attack code, since they are integrated by legitimate app developers who have no intention to hide their package names.

In our research, we performed this analysis over 1.3 million Android apps collected from the app markets around the world, including over 400,000 popular apps from the Google Play store (see Section IV-A for details). We implemented a tool, called *LibFinder*, to automatically analyze the disassembled apps, which discovered 612,437 packages and further organized the packages into 763 groups according to their names. Particularly our approach utilizes the Root Zone Database [17] to identify multi-level domain names for accurately grouping the packages. Within each group, our approach further clustered all the packages to identify libraries and their variations (different versions or those customized by the third party).

Finding Android libraries. The purpose of clustering is to find out all the related packages, those sharing a large portion of code with others in the same cluster. To this end, we first define a distance, called *package similarity degree (PSD)* based upon Jaccard index, to measure the similarity between two packages: for two packages p_1, p_2 from different vendors, $PSD(p_1, p_2) = n(p_1 \cap p_2)/n(p_1 \cup p_2)$, where $n(p_1 \cap p_2)$ is the number of common methods shared between p_1 and p_2 and $n(p_1 \cup p_2)$ is the number of unique methods in either p_1 or p_2. To compare two packages at the method level, LibFinder utilizes the centroid-based approach [5], which computes the geometric center (centroid) of a model derived from a method's CFG (see Section II) to represent the method: two methods are considered to match each other if their centroids come very close (within the boundary set according to the prior work [5]).

The PSD between two packages describes their similarity. The higher it becomes, the more likely that these packages are variations of the same library. To determine the threshold for classifying the packages into a library, we utilized two training datasets in our research: the first one contains 20 randomly-selected libraries (e.g., `youmi` and `unity3d`) that are unrelated to each other and the second one involves 20 libraries together with their different versions (e.g., updates, patched libraries, etc). Figure 3 shows the distribution of the PSDs between unrelated libraries and that of those related. As we can see from the figure, the unrelated libraries never share more than 13% of their methods, while for those related, they have at least 57% of the methods in common and most of pairs have PSDs above 85%. Given the huge gap between those related and those not, we set the threshold to 35%, right in the middle between 13% and 57%, which easily differentiates these two types of library pairs (see Figure 3).

Using the threshold (35%), LibFinder first clusters the packages within the same group (based upon the shared package name prefix) with algorithm DBSCAN [18], and then checks

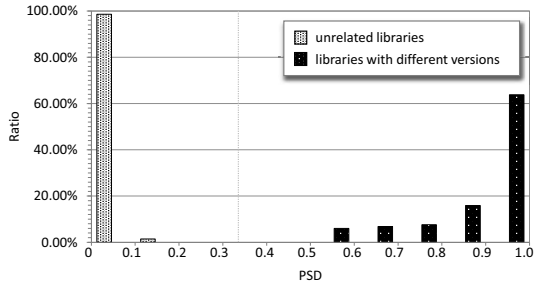

Fig. 3: *Distribution of PSDs between unrelated libraries and that of those related.*

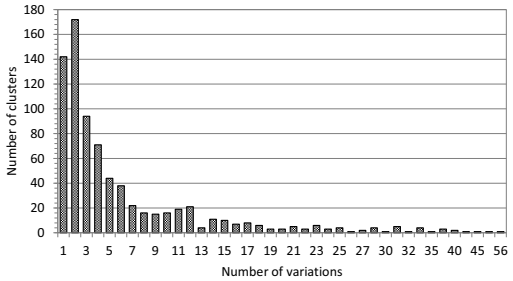

Fig. 4: *Distribution over the number of variations for each library discovered.*

whether the clusters in different groups can be merged (when the members in two clusters share over 35% methods). In our research, we successfully compared over 1.3 million Android apps in this way and discovered 763 clusters, each considered to be a library. Figure 4 shows the distribution over the number of variations for each library discovered in this way.

Detecting PhaLibs. To determine whether a library and its variations are PhaLibs, we scanned them with VirusTotal (Figure 1). A package is flagged as suspicious if at least two scanners report it. A technical challenge here is that we cannot trivially extract the libraries and directly scan them with VirusTotal, which only works on apps, not the library packages. Our solution is to scan the special host app of a library, which carries nothing but the library. For this purpose, we first pick up an app integrating the library, locate the package within the app's `DEX` bytecode and remove all other code to build the new host app. More specifically, our approach automatically discovers the program location of the package from the header of the `DEX` file, including its code and data, before emptying the whole app and converting it to a placeholder for the package. In this way, whatever is discovered by VirusTotal can be attributed to the library.

To scan such an app, VirusTotal has to work in its scan model, running all 54 AV systems on the app. This is much more heavyweight than the caching mode, in which only the checksum of the app is compared against those already scanned. In the scan mode, on average 5 minutes need to be taken to analyze an app. To efficiently handle over 763 libraries and their 4,912 variations discovered in our study (which could take long time to scan if the libraries are processed sequentially), we come up with a technique to analyze multiple libraries together. Specifically, our approach first combines different packages within the same cluster together on a single placeholder app, as many as possible, under the constraint of the file-size limit put by VirusTotal. If the app is reported to be legitimate, we can drop all the libraries involved. Otherwise, we upload and scan each variation one by one. Using this approach, we went through all 763 libraries and their variations found from the 1.3 million apps within only 1,725 scans, total one day with 6 VirusTotal accounts.

C. Analyzing iOS Libs Cross-Platform

Finding iOS PhaLibs is challenging, since recovering libraries from the binary code of iOS apps is hard and no AV systems are publicly available for the Apple platform to validate our discoveries. To address this issue, we utilize a key observation that many iOS libraries actually have Android counterparts. More specifically, in our research, we looked into the top 38 iOS libraries as reported by SourceDNA, an analytics service that profiles the Android and iOS app stores. It turns out that 36 of them, nearly 95%, have Android versions (Table VI in Appendix presents the details of these popular libraries). Since Android is less protected than iOS, there is no reason to believe that once a legitimate library's iOS version is contaminated (e.g., adwo), its Android version will remain intact. Therefore, we decided to map a confirmed Android PhaLib (by VirusTotal) to its iOS counterpart (if exists) and utilize the Android-side ground truth to help validate the potentially harmful code discovered on the iOS side. This approach will certainly miss some iOS PhaLibs. Nevertheless, it serves as a first step towards systematic study of iOS PHAs and provides a baseline for a better understanding of the security risks posed by iOS libraries.

Such a cross-platform mapping, however, is by no means trivial. Android and iOS are two dramatically different systems with totally different frameworks and APIs, and the program languages for developing apps (Java vs. Objective-C). Given the huge gap between the two platforms, it does not come with a surprise that oftentimes, the Android and iOS versions of the same library are actually designed and implemented independently by different developers. As a result, the program structures and logic can be very different across the platforms, even for the same library. For example, a function on one platform can be implemented into multiple ones on the other; APIs on different platforms are hard to align, and even when this can be done, the input arguments of the APIs can also be significantly different (Figure 5). Although prior research studies the relations between the variations of the same program (e.g., one obfuscated while the other is not) [19] or those on different platforms but compiled from the same source code [20], never before has any effort been made to correlate two independently developed programs with the same functionalities across platforms.

With the difficulty in correlating Android and iOS libraries, we believe that there must be some invariant relations between

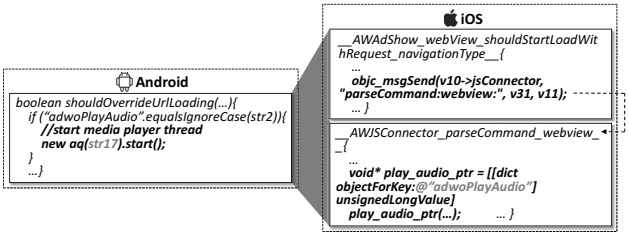

Fig. 5: *Different ways to use audio in Android and iOS platforms. The name of the class (i.e., aq) is obfuscated in Android part. In iOS part, the method is in the form of function pointer and stored in a dictionary.*

them. After all, they are just different versions of the same library, providing identical or very similar services to the users. For example, no matter how different an ad library's Android and iOS versions look like, they have to communicate with the same server or at least the hosts in the same domains (http://apiconfig.adwo.com for adwo), and they need to promote the similar products. In our study, we developed a technique that establishes such a correlation through the invariants shared across the variations of the same library. More specifically, our approach performs a static invariant discovery, using a training set to select from a known list of invariants [21] suitable for bridging the gap between the platforms. Such invariants were later automatically extracted from the Android PhaLibs (packages included in the clusters) as well as over 140,000 iOS apps (decrypted using Clutch [22] and dissembled using capstone [23]) and used to identify the libraries embedded within the iOS apps. Further we compared the suspicious behaviors between the Android and iOS libraries of the same origin to determine whether the activities deemed potentially harmful on the Android side are also there within the iOS counterpart. This is important because Apple is more rigorous in security control than Android. It is possible that some Android-side operations are no longer allowed on iOS (e.g., sending SMS in background). For this purpose, we utilized VirusTotal to indirectly validate the potentially harmful behaviors in iOS apps, even though the AV system cannot directly work on the Apple platform. Following we elaborate our techniques.

Cross-platform invariants. Invariants provide valuable information about a program's operations and data structures, which is a good source for software testing [24], understanding [25] and etc. A typical invariant inference approach usually instruments the source code of a targeted program, runs it on different inputs for several times, and records the values of each variables inside the program for inferring invariable values at specific program points such as procedure entries and exits.

These existing techniques aim at discovering invariants within the same program or between its variations built by the same group of developers. What we are looking for, however, is the connections between libraries on different platforms, which are developed independently, involving different variable/function names, control/data flows and even different API and system calls specific to the platforms (a property used in the prior research as an invariant [21]). Also challenging here is the scalability of the cross-platform analysis: as mentioned earlier, we need to map 763 libraries and their 4,912 variations (discovered from 1.3 million Android apps) to the binary code of 140,000 iOS apps. Therefore, we cannot afford to execute these programs to identify their runtime invariable data, and have to resort to a static invariant analysis.

To find out such cross-platform invariants, we looked into a collection of program points typically used in invariant discovery, as elaborated in the prior study [21], including program entry, program exit and loop header. At different program points, the invariants are different, which affects the outcomes of the mapping. More specifically, procedure entries and exits are widely used as the program points (e.g., by Daikon [21]) where invariants are collected. However, when cross-examining procedures (i.e., methods) over different platforms, we found that rarely they could be mapped to their counterparts. Also unreliable is the code fragment within a method, which cannot be used to link independently developed libraries, as discovered in our study. Further complicating our mission is the difficulty in even identifying libraries within the binary code of an iOS app: unlike an Android library, which is integrated into an app as a separate Java package, an iOS library is typically broken down into methods and mixed together with other libraries and program components. We found in our research that the only reliable program unit for the invariant discovery cross platforms is class. A class is related to a certain object within the program, such as a button, and often designed to handle certain events. At this level, we observe some cross-platform links: e.g., the same `button/webview` shows up on both Android and iOS libraries and a similar `click/load web page` event needs to be handled by the corresponding objects on these platforms. Also, a Java class is easy to find from Android bytecode; on the iOS front, all the methods under a class are named by the class name followed by their individual method names, which allows us to easily group them together.

Based upon this observation, we further studied the cross-platform invariant discovery over a training set with 20 manually paired Android and iOS libraries. These libraries were collected from their official websites. Within each pair, we manually labeled their corresponding classes (those with the same functionalities) whenever possible. Altogether, 126 pairs of classes were identified and labeled. Over those library pairs, we inspected the program points according to the list of candidate invariants [21], using a dictionary (manually constructed) to translate the instructions and APIs cross platforms. More specifically, our approach ran SmartDroid [26] to construct the control-flow graph (CFG) for the methods under each Android class and built the CFG for the iOS methods based on capstone [23]. Then, we looked at the corresponding program points (class construction, class destruction, view appear, view disappear, or methods with sensitive events) within the classes across the platforms to find out those whose values can be determined statically, and are also consistent across the corresponding classes and different between unrelated classes. What has been found was further manually inspected to ensure

that indeed the program features are present within the classes on the both platforms and also reliable. This study shows that constant strings (or substrings) turn out to be the most reliable feature shared by the related libraries on both platforms. Such strings include URLs, JSON Keys, program logs, etc., which are expected to be utilized by the libraries no matter how it is implemented. Table I elaborates the strings and how they are used. For example, within the `AxAdObject` class of the library admixer, strings like "`Load Timeout`" and "`FailedToReceiveAd`" appear on both the Android and iOS side, which are the text the apps use to communicate with the user, and strings like "`&ad_network=`" can also be mapped, which are used to compare the input data from the remote servers to find out its type (e.g., different commands).

How strings are used?	Cross-platform example strings that we found
Keys for JSON or Dictionary	"AppSEC", "mediaURL", "guid_key"
Resources	"offerwall-flow.html", "webview_bar_back.png"
Developers' information	"partners2@adsmogo.com"
Scheme	"adwo://", "wgtroot://", "mraid://", "redir://"
Cyphertext text/code	"DUBu6wJ27y6xs7VWmNDw67DD" "02e310a99f1640b53e88e9e408295a94"
Program logs	"Load Timeout", "FailedToReceiveAd : %@" "[AdPack] interstitial displayed"
Certain Format	"</HitTable>", "</DocumentElement>" "yyyy-MM-dd'T'HH:mm:ssZZZ"
URL related	"&width=%d", "&ad_network=" "http://track.adwo.com:18088/track/i" "http://www.admarket.mobi/ftad/apiadreq"
Command and JS code	"adc_bridge.fireAppPresenceEvent(%@, false);" "window.mogoview.fireChangeEvent(%@);"

TABLE I: Cross-platform strings and how they are used.

Figure 6 illustrates the results of using common strings as invariants to pair libraries over a test set with 126 pairs of classes and 20 pairs of libraries. As we can see, when the number of matched constant strings (at least 5 letters long and at different program points) within a pair of classes goes above 8, the false detection rate (FDR, the ratio of incorrectly mapped classes among all those paired across the platforms) becomes only 1%. Although the coverage is 40% in this step, we perform an extension on these pairs and find much more classes in the next step. Further, when the number of matched class pairs (through the constant strings) exceeds 3, an Android library is almost certain to be mapped to its iOS counterpart. Therefore, these two thresholds (8 for pairing classes and 3 for pairing libraries) were used in our research to discover Android-related PhaLibs from iOS apps. Running this technique on 140,000 iOS apps, we successfully mapped 46 Android PhaLibs to the iOS libraries integrated within 17680 apps. All these matched libraries were manually validated, by inspecting their corresponding functionalities, and we found that the relations between the identified libraries (pairs of Android packages and iOS apps) were all correct (Section IV-A).

Finding library members. As discussed above, across the platforms, we can correlate an Android library to its counterpart integrated within an iOS app. Particularly, the classes within the app that have been mapped to their Android counterparts based upon shared strings are all considered to be within the same library. However, these libraries, which we call *anchors*, are the only members of the library we can find. Other library

Fig. 6: *False detection rate and coverage when using different number of strings to map classes.*

members also need to be discovered from the iOS app for analyzing the library's behaviors. This is nontrivial, due to the way iOS libraries are integrated within the app: all the methods are mixed together, and even though we can still group them using their names into classes, there is no straightforward way to link different classes together to find a library. Our solution is to statically analyze the code within the anchors to identify their relations with other classes (e.g., an anchor's method is invoked by another class) and use such relations to find other members within the same library. Note that a direct invocation from a class B to the anchor A does not necessarily mean that they are inside the same library: for example, B could be a function within the app that makes a call to the library where A stays.

Specifically, in our research, we developed a technique that automatically explores the anchors' relations with other classes to discover other members inside a library. Our approach is based upon three kinds of inter-class relations: *Call*, *Inherit* and *Refer*. When a method in Class A calls a method in Class B, their relation is denoted by $A \rightarrow_{Call} B$. When A is inherited from B, the relation is $A \rightarrow_{Inherit} B$. When an object of B is used inside A, we describe the relation as $A \rightarrow_{Ref} B$. Note that the relations here are directional: e.g., $A \rightarrow_{Call} B$ does not imply $B \rightarrow_{Call} A$.

Our approach uses the following rules to discover new library members:

• For any class A inside a library, we consider another class B also inside the same class if B is not a framework class, which is determined using a manually constructed list with 972 system classes (on Apple SDK 8.3), and also $A \rightarrow_{Call} B$ or $A \rightarrow_{Inherit} B$ or $A \rightarrow_{Ref} B$. In most cases, if a method calls, inherits or refers to the object within another non-system class, the latter should also be part of the library.

• For any *anchor* class A inside a library, we consider a non-framework class B part of the library if $B \rightarrow_{Call} A$ or $B \rightarrow_{Inherit} A$ or $B \rightarrow_{Ref} A$, and there exists another Android class B' that has the same relation with A's counterpart A', and B and B' share at least k common strings. In this case, B is also labeled as an anchor.

Here k is set below 8, the threshold for selecting an anchor without considering its relations with other classes. This is because B's relation with A, an anchor, is an additional feature that can help classify B, and therefore its affiliation can be

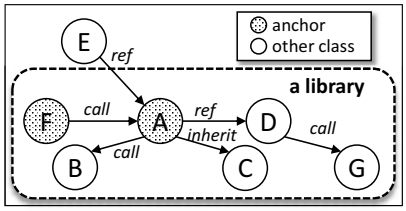

Fig. 7: *An example showing how to extend classes using anchors in a mapped library.*

determined without solely relying on the common strings. Indeed, in our research, we analyzed 222 such classes and found that when $k \geq 3$, B can always be classified into A's library, without any false positive.

Given a set of anchors identified from an iOS app solely based upon common strings, the above two rules are then applied iteratively to other classes within the app until no new class can be added into the library. Figure 7 presents an example for this cross-platform library identification. In the figure, the class A is an anchor and all other classes (B, C, D in the figure) that it calls, inherits or refers should be put into the library (due to the first rule). Further, G is classified into the library because of its relation with D. Also, though $E \rightarrow_{ref} A$ and $F \rightarrow_{call} A$, we only set F as an anchor and a member of the library since it shares at least 3 common strings with its Android counterpart while E does not.

In our research, we utilized a test set with 20 mapped library pairs to evaluate the effectiveness of the above rules for discovering iOS classes within a library. For each library pair, the two rules were applied to extend the iOS library that initially only contains anchors. Then by comparing each newly added class with the official iOS library downloaded from the web, we manually checked whether the class should be included in the library. If not, a false positive is recorded. Once the iOS libraries was fully extended, we manually checked all the classes our approach identified. The FDR was 0.5%. A limitation of this technique is that it cannot cover the independent classes with no relation with other classes. To measure how many classes were missed, we checked the official libraries and found that the false negative rate is 28.84%.

Potentially harmful behavior. Successful mapping of an Android PhaLib to an iOS library does not necessarily mean that the latter is potentially harmful. Due to Apple's strict security control, including its app vetting and security protection at the OS level, some potentially harmful behaviors within the Android library could be dropped from its iOS counterpart. Further, it is possible that an Android PhaLib is a repackaged legitimate library, while its corresponding iOS library recovered from apps has not been contaminated. As mentioned earlier, confirming potentially-harmful activities within an iOS app is hard, due to the lack of ground truth (no public AV system working on iOS apps). In our research, we came up with a novel technique that leverages VirusTotal to determine the presence of suspicious behavior within an iOS app. More specifically, our approach is designed to find a corresponding behavior between the Android and iOS versions of a library, and further determine whether such a behavior is considered to be potentially harmful by VirusTotal: if so, then we get the evidence that indeed the iOS library behaves in a way that VirusTotal would consider to be potentially harmful when the same behavior is observed from an Android app.

To this end, we first need to model a library's behaviors across the platforms. Conventionally, a program's behavior is described by its API sequences, which has been extensively used in PHA detection [27]. However, a direct application of the model to serve our purpose faces a significant challenge. To see where the problem is, let us look at Figure 8, which presents an API sequence within an Android library for adwo, and another sequence in an iOS app that does the same thing. The trouble here, as we can see from the figure, is that not only a dictionary is needed to map the APIs from one platform to the other, but some semantic knowledge should be there to help understand that the operations performed by one API on one platform (e.g., `CTTelephonyNetworkInfo: subscriberCellularProvider` in the figure) may need to be handled by multiple APIs on the other platform (e.g., `ConnectivityManager.getActiveNetworkInfo()` and `NetworkInfo.getTypeName()` for `ACCESS_NETWORK_INFO`). Precise mapping of such a relation (one to many APIs for a specific set of operations) is hard and cannot be easily done using a dictionary.

An observation from the figure is that once we generalize the sequences, replacing each API with its category (e.g, `CTTelephonyNetworkInfo: subscriberCellularProvider` is replaced by `ACCESS_NETWORK_INFO`) and further removing the consecutive occurrences of the same category (e.g., dropping the subsequence of repeated `ACCESS_NETWORK_INFO` and keeping only one), the two sequences on the Android and iOS fronts look very much alike, given the translations between their API categories: e.g., the Android API `TelephonyManager.getDeviceId()` is mapped to `READ_DEVICE_INFO` and also `ASIdentifierManager:advertisingIdentifier`. In the figure, we show the part of the dictionary for such API category translations. It is conceivable that this generalization makes the dictionary construction easier and the comparison across different API sequences feasible. In the meantime, the treatment is also found to be sufficient specific for modeling an app's behavior. Our research shows that comparing two sequences at this API category level is accurate, introducing 3.3% FDR according to our manual validation. Intuitively, replacing an API with its category avoids the trouble introduced by multiple APIs with similar functionalities. Further, we found that oftentimes, before an API is invoked, several other APIs first need to be called to prepare the parameters for such an invocation: for example, to call methods in *NSURLConnection*, methods in *Reachability* first need to be triggered to check whether network service is available on iOS. These APIs (for preparing for the call) are often within the same category and therefore replacing them

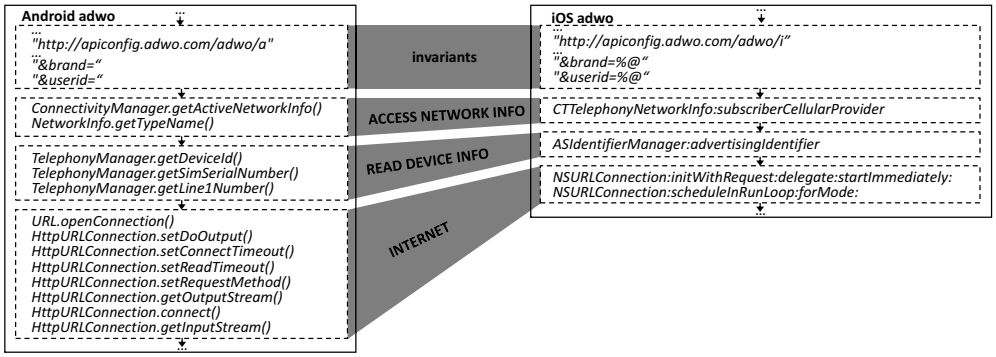

Fig. 8: *The APIs used by the same behavior are quite diverse in two platforms. If category is used instead of a concrete API, the two behavior sequences can be mapped.*

with a single category hides the diversity of such a preparation step across different platforms.

Specifically, in our research, we define a behavior as a sequence of API categories discovered from a program's CFG, together with the occurrences of the invariants (i.e., the constant strings) used for mapping libraries cross platforms. In Figure 8, the behaviors within the Android and iOS libraries are described as such sequences (illustrated in the figure). To compare the behaviors between 46 matched PhaLib pairs, we first created a dictionary that maps 21 Android framework classes and 39 iOS framework classes (i.e., schemes), including over 500 APIs on each side, to 19 categories[1]. To map an Android class to a category, we leveraged the permissions used by APIs in the class. Specifically, such permissions were automatically discovered using PScout [28] and then manually inspected before placing the class requiring them into a specific category. For example, a class asking for `android.permission.SEND_SMS` and `android.permission.READ_SMS` are put in the category "*SMS*". On the iOS side, since there is no such API-permission mapping, we had to manually find out each API's semantics to determine its affiliation with a specific category. Note that Android developers may use Intents instead of framework APIs to perform some operations: for example, they may run "content://com.android.contacts" to read contacts. Therefore, we also included 14 similar Intents in the dictionary. On top of the dictionary, we further implemented an automatic analysis tool called *BehaviorFinder*, which starts from the methods not called by any other methods inside a library (e.g., entry point, event handlers) and conducts a standard static inter-procedural analysis [29] to generate the invariant-API-category (IAC) sequences within both Android packages and iOS apps. For this purpose, we ran SmartDroid [26] to build the cross-procedure CFG for the Android package and construct the CFG over the classes identified within an iOS app based on capstone [23]. Along the CFGs, invariants and APIs are automatically discovered and translated using the dictionary, until IAC sequences are fully constructed.

Given the behaviors (the IAC sequences) from an Android PhaLib and its iOS counterpart, BehaviorFinder further carries out a pair-wise comparison between the sequences from the two platforms. Whenever the two sequences are found to share a common subsequence at least 80% of either sequence in length, we consider that these two sequences are matched and the behaviors they represent are shared across the two libraries over different platforms. To understand the accuracy of this approach, we randomly selected 90 mapped IAC pairs in our research and manually inspected the corresponding code on both platforms to find out whether they really describe the same behaviors. This validation process shows that the FDR introduced by our approach is only 3.3%. Running BehaviorFinder on the PhaLab pairs discovered, we successfully extracted 9,259 IAC sequences from 46 iOS libraries and 16,762 from 46 Android libraries, and identified 2,891 common behaviors.

The last step of the analysis is to find out whether a common behavior is indeed potentially harmful. To this end, we ran a script to remove the behavior's IAC from all the packages within the same cluster and then submitted the placeholder app of this cluster (i.e., the fake app that integrates the packages within the cluster; see Section III-B) to VirusTotal. When the scan report comes back, our approach inspects the number of scanners (in VirusTotal) flag the placeholder app as potentially harmful: if some scanners before removing the IAC, marks the placeholder as potentially harmful but stops doing so after the IAC sequence is gone, we have reason to believe that the sequence is considered to be part of the signatures used by those scanners[2]. This indicates that indeed the sequence is part of potentially-harmful activities. In our research, our approach recovered 838 such confirmed harmful behaviors out of the 2,891 common behaviors across Android and iOS PhaLibs.

IV. Findings and Measurement

In this section, we report the discoveries made by running our cross-platform methodology over 1.3 million Android apps and 140,000 iOS apps. Our research brought to light a large number of PhaLibs, both on Android and on iOS, and their significant impacts. Also highlighted in our study is the observation that repackaging third-party libraries likely already becomes an

[1]We plan to release the dictionary at www.appomicsec.com.

[2]Note that the placeholder is a fake app and all the packages it includes have been modified when extracted from their host apps, and therefore, the scanners flagging the app are certainly not using checksums.

Market	# PhaLibs	# of total apps studied	# Infected apps
Google Play	77	400000	27353 (6.84%)
Asia Market	113	800000	67108 (8.39%)
Other Markets	25	100000	3847 (3.85%)

TABLE II: PhaLibs in Android markets

important avenue for propagating mobile PHAs. Further we revealed the correlations between Android and iOS PhaLibs.

A. Settings

Apps. We collected 1.3 millions Android apps from over 30 markets (Google Play and other markets in America, Europe and Asia), and 96,579 iOS apps from the official Apple Store and 45,966 apps from third-party markets (2 American, 3 Asian and 2 European markets, as illustrated in Table II & III). One of these third-party markets install Apple apps to un-jailbroken devices and others serving jailbroken devices. From the third-party stores, we randomly selected apps to download, while from the official stores (i.e., Google Play and Apple Store), we first picked up the most popular apps in each category (top 500 for Google Play and top 100 for Apple Store) and then randomly chose the targets from the rest. Also, we removed duplicated apps according to their MD5.

Computing environment. Three powerful servers were used to analyze all the apps. One has 20 cores at 2.4GHz CPU, 128GB memory and 56TB hard drivers. The other two servers each have 12 cores at 2.1Hz CPU, 64GB memory and 20TB hard drivers. The operating systems are Red Hat Linux.

Validation. All the findings made by our methodology were thoroughly validated to ensure their accuracy. Specifically, 100 randomly sampled clusters from the total 763 clusters (for Android library discovery) were analyzed manually: from each sampled cluster, 10 packages were randomly picked out to determine whether they include similar methods, provide similar services and should be considered as the variations of the same library. Further, from all the libraries mapped across the platforms, we randomly selected 50 pairs (an Android package and an iOS app found to contain related methods) and again manually analyzed their code to determine whether indeed they are different versions of the same library (for the iOS app, the library it integrates). Finally, we randomly sampled 90 pairs of IAC sequences matched between Android and iOS PhaLibs to find out whether they are related to the same activities (e.g., read contacts and send them out) and whether there is any reason to believe that the behavior is potentially harmful.

This validation process shows that both the library clustering and cross-platform library mapping are highly accurate: we did not find any false positive. On the other hand, the behavior paired across the platforms did cause some mismatches, introducing a FDR below 3.3%. In other words, the vast majority of the matching we found are correct.

B. Landscape

PhaLibs on Android. From the 1.3 millions Android apps in our dataset, we found 763 libraries with totally 4,912 variations (different official versions, third-party customizations, etc.). Running VirusTotal [30] on these libraries (Section III-B), we discovered that 1008 variations of 117 libraries are potentially harmful (flagged by at least two scanners in VirusTotal). Figure 9 illustrates the distribution of PhaLibs over the number of scanners. As we can see from the figure, 386 variations of 32 libraries were alarmed by at least 10 out of the 54 scanners. 77 of the 117 libraries were downloaded from Google Play. By comparison, third-party markets are more susceptible to PhaLibs, in general, about 7.88% of their apps infected with these libraries.

To understand the impacts of the PhaLibs discovered, for each library, we added together the total installs of each Google-Play app that integrates it. Figure 10 illustrates the distribution of the numbers of PhaLibs over the sum of the installs from all the apps using them. The figure shows that some PhaLibs (e.g., `jirbo`) have been installed over 279 million times, and altogether, each PhaLib has 11 million installs on average. This indicates that the impacts of such libraries are indeed significant.

PhaLibs on iOS. We found that among all 1008 variations of 117 PhaLibs, 46 PhaLibs were mapped to their counterparts integrated within iOS apps. Among them, 23 PhaLibs (706 variations) were shown to have potentially harmful behaviors related to their Android versions. Compared with Android, the total number of PhaLibs on iOS is relatively low, which is understandable due to the fact that our study focuses on the iOS libraries with Android versions. Still, such iOS PhaLibs have a significant impact, affecting over 6,842 iOS apps, and also 2,844 of them show up on the Apple App Store. Even with its rigorous vetting protection, the apps integrating those libraries still fall through the cracks. As an example, an ad library `adwo` we found has infected at least 61 apps on the Apple Store, performing harmful activities such as taking pictures, recording audio and uploading the recordings and the user's contacts and precise locations to a remote server. Also we found that some versions of a library contain potentially harmful behaviors while others do not (details are illustrated in appendix: Table VIII for Android and Table VII for iOS). On iOS, this raises the concern that potentially a new version of a library may not receive a full security inspection if its old version (without harmful behaviors) passed the vetting process before.

We also found 19 PhaLibs in third-party app stores, both Android and iOS. Table III elaborates our findings. Particularly, 16 PhaLibs infecting 2,985 apps were discovered on the 91 market [31], a Chinese Apple app market that enables the iPhone users to install its apps without jailbreaking their devices (possibly through enterprise certificates or other techniques [32]). The harmful behaviors of these libraries include collecting accurate location data, voice recording, etc. Also, our study reveals 18 PhaLibs in jailbreak app stores, which perform such activities as uploading installed application list, install application with private APIs, and taking photos. Just as expected, more PhaLibs are on the third-party stores than the official Apple Store.

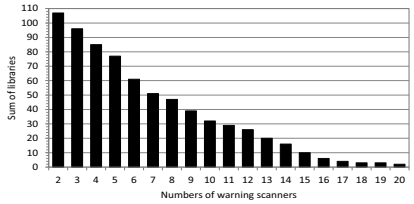

Fig. 9: *Distribution of PhaLibs over the number of scanners.*

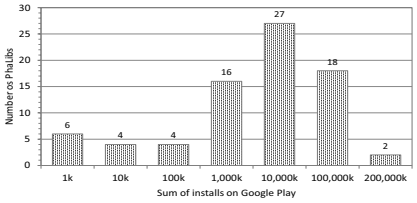

Fig. 10: *Distribution of PhaLibs over the sum of installs from all the apps using them.*

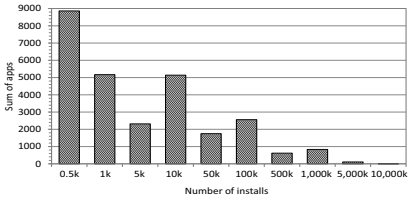

Fig. 11: *Distribution of apps over their total installs.*

Among all 23 iOS PhaLibs, 15 of them are ad libraries while the other 8 are toolkits, such as umpay, lotuseed and mobclick. Some of them are available on the public repositories, such as GitHub and Google Code. Using their names and some invariant strings within the libraries, we searched Google and found 6 of them are publicly available, while the other 17 PhaLibs are not online. Among these PhaLibs could be those distributed in the underground community. Also, some of these libraries have both their official versions and third-party customized versions. Later we show that this could be indicative of the presence of library repackaging attacks (Section IV and Section IV-D).

Infected apps. As mentioned earlier, our study shows that 27,353 apps (6.84%) in Google Play and 70,955 apps (7.88%) in third-party markets are infected through libraries. Table II shows the detailed findings. As we can see from the table, the Asia markets are more likely to host infected apps. On the other hand, still many PHAs are on the Google Play store. Of particular interest here is that compared with what are reported by the prior research, which discovers that 7% of the apps on the Play store are PHAs, apparently, PhaLibs could be behind most of these suspicious apps. This indicates that libraries could be an important (and also largely overlooked) channel for disseminating potentially harmful code. Further we analyzed the popularity of these PHAs. Figure 11 shows the distribution of those apps over their total installs. About 4,123 have been installed over 50,000 times each. Among them are popular apps such *Unblock Me Free* and *Clash of Clans*, which have been installed for over 50 million times each.

On the iOS side, we were surprised to find that 2.94% of the apps from the official Apple Store are infected through the PhaLibs (706 variations). It is widely believed that very few PHAs exist on Apple's official market. As far as we know, the number of disclosed PHAs is very few till now [12]. However, by mapping Android PhaLibs to iOS apps, we were able to identify 2,844 apps (out of 96,579) to be PHAs. Some of them are quite popular, for example, the popular game called "2048". We have reported those PhaLibs and apps to the Apple security team. Also, we note that 50.45% of these PHAs were actually uploaded this and last year, which indicates that suspicious apps are more likely to be among the new apps on the Apple Store (Figure 13 shows the distribution of such PHAs over the time they stay on the Apple Store). One possible explanation is that PHA authors are increasingly moving their attentions to the Apple platform. In the meantime, we found that 1,346 apps have been on the store for quite a long time (24 months).

On the third-party Apple stores, our research shows that 8.7% of the apps there are PHAs (about 3 times of those on the official Apple Store), which is comparable with the Android markets [2]. Altogether, 3,998 out of 45,966 apps there are infected through PhaLibs. Table III presents the findings across the official and third-party markets.

Market	Area	Type	# of PhaLibs	# of apps studied	# infected apps
Apple Store	Global	Official	23	96,579	2,844 (2.94%)
91	China	3rd party**	16	34,338	2,985 (8.69%)
51 ipa	China	Jailbreak	10	2,594	159 (6.13%)
Baidu*	China	Jailbreak	16	5,393	306 (5.67%)
Vshare	US	Jailbreak	10	2,163	389 (17.98%)
PandaApp	US	Jailbreak	7	1,292	148 (11.46%)
iDownloads	Russia	Jailbreak	3	186	11 (5.91)%

TABLE III: PhaLibs in iOS markets
(*Apps are from Baidu cloud disk, uploaded by multiple users. **3rd party apps for non-jailbreaking iPhones.)

Behavior. We analyzed all 117 Android PhaLibs to understand their potentially harmful behaviors, focusing primarily on the activities matching those within their iOS counterparts (through the IAC sequences as described in Section III-C). Besides common activities such as tracking users' fine location and sending out private data (e.g., IMEI, app list, phone number, etc.), our study leads to the discovery of a set of highly suspicious activities never reported by prior research. For example, some ad libraries (e.g, `adwo`) were found to contain back-doors that execute the commands received from a remote server, taking photos/videos/audios and sending them out to any IP address given by the server. Further under the server's control, the libraries can read or even add items to the user' contact list. Such operations may allow the adversary to modify an existing contact using her information (e.g., replacing a friend's email or website with those under the adversary's control), which opens the door to an impersonation attack. Also interestingly, the ad library `adwo` even operates on the user's photo gallery, with the capability to add, delete and edit photos there. This is very unusual for an ad library, as it is less clear how the capability can legitimately serve advertising purpose. On the other hand, we note that some known memory flaws can be exploited through pictures [33], which can be executed using such a capability. Other identified suspicious behaviors include reading from the user's reminder, calendar and passbook, and changing her todo list, etc., which could have serious consequences (e.g., changing the dosages of the medication on one's todo list). Again, it is certainly surprised to see that such behaviors are performed within an ad library like adwo. Another dangerous operation is to open any URL scheme given by a remote party, which may cause the device

to download any app indicated by the adversary or running any app on the device. Finally we found that some PhaLibs (mostly ad libraries) even prompt the user to send an SMS message or email or make a phone call. All these behaviors were successfully triggered in our research through a (manual) dynamic analysis, indicating that the threats they pose are indeed real. Table IV summarizes our findings.

On the iOS front, which is supposed to be securer due to Apple's rigorous security control, we were surprised to find that most of the aforementioned harmful behaviors are preserved within the corresponding iOS PhaLibs, as illustrated in Table IV. From the table, we can see that compared with the Android-side behaviors, the only one missing on the list is reading the contact list, whose Android-side IAC sequences were never matched to any behavior on the iOS side. This capability turns out to be well guarded on iOS, requiring the com.apple.security.personal-information.addressbook entitlement rarely granted to the app without proper justification. An example of such a PhaLib is `adwo`, a library integrated within 111 Android apps and 61 iOS apps. Interestingly, though the behavior (reading the contacts) failed to show up on the library's iOS version, we found that the function interface for this operation is still left there within the library but the body of the function is empty. Further we observed that some Android-side functionalities have been changed to suit iOS, though the relations cross the platforms are still clearly there: for example, the check on whether a device has been rooted has been replaced with a jailbreak check, which could lead to installing apps through private APIs. We also note that some dangerous activities that can be performed stealthily on an Android device (e.g., sending SMS message) have to be done in a more explicit way on iOS: i.e., asking for the user's consent. This indicates that the adversary indeed has to adapt to the more restrictive security control on iOS. On the other hand, our research also reveals some iOS-specific suspicious behaviors (showing up on the IAC sequence within iOS apps): a PhaLib wanpu was found to call `_SecItemCopyMatching` for accessing and operating on its hosting app's keychain, which could lead to the disclosure of the user's password and other private information associated with the app (Figure 16 in appendix).

Behavior	iOS	Android
take a picture and send it out	Y	Y
record/play voice	Y	Y
send text messages	Y	Y
send emails	Y	Y
make a phone call	Y	Y
read/write/detele bookmarks	Y	Y
download apps and install	Y	Y
steal contact list	N	Y
steal user accounts, location, phone number	Y	Y
steal cpu, mem info, ip address, device ID, arch	Y	Y
inject javascript code	Y	Y
jailbreak related	Y	N
access keychain	Y	N

TABLE IV: Comparing Android and iOS PhaLib behaviors

Simultaneously and independently, FireEye recently blogs the discovery of `mobiSage`, an iOS SDK embedded with a back-door that infects thousands of iOS apps [34]. This SDK has also been discovered in our study, together with its

Lib name	System	# Infected apps	# Downloads	Reported?	Behaviors
mobiSage	Android	123	835,000	N	CA,RE, LO, KE
	iOS	32	N/A	Y	
adwo	Android	111	2,500,000	N	CA, RE, LO, CO, SMS
	iOS	61	N/A	N	EM, PH, DE, JA, JS
Leadbolt	Android	1189	399,150	N	DE, LO, JS
	iOS	275	N/A	N	
admogo	Android	102	N/A	N	SMS, LO, DE, JA
	iOS	134	N/A	N	
wanpu	Android	368	N/A	N	JA, DE, IN, LI, KE
	iOS	13	N/A	N	
prime31	Android	7042	3,162,160	N	SMS, LO, DE, PH
	iOS	7	N/A	N	
jirbo	Android	3295	192,075,251	N	LO, DE, PH
	iOS	186	N/A	N	

TABLE V: Example of backdoors on Android and iOS markets

CA: Camera; RE: Record; LO: Location; KE: Keychain (only iOS); CO: Contact; EM: Email; PH: Phone; DE: Device Info; JA: JailBroken (only iOS); JS: Inject Javascript code; IN: Install apps; LI: List apps

(*# downloads* is not available in Apple Store or in third-party Android markets.)

Android version, which has *never* been reported before. Most importantly, the SDK is just one of the back-door PhaLibs we found. Others are presented in Table V. These PhaLibs all exhibit similar behaviors as `adwo` across their Android and iOS versions. Their impacts are described in the table. More details are given in Appendix (Figure 14 and 17).

C. Android PhaLibs

Spread through repackaging. By analyzing different variations of Android PhaLibs, we found that some of them are actually benign, not including any potentially-harmful activities discovered in the other variations of the same library. Most intriguing is that among all 117 PhaLibs recovered from apps, 58 do not have their harmful variations found on Google Play. Actually these variations were all collected from third-party markets. Table II present examples for those libraries and the markets where their potentially harmful variations were discovered. To find out where their potentially harmful code come from, we utilized `Dex2jar` [35] to disassemble these libraries' bytecode into Java source code and compare them to extract the difference that contains potentially harmful code. Typical behaviors of the code snippet is sending devices' information (e.g., `deviceID` and `simSerialNumber`), discovered from the PhaLib `mappn`, an analytics library integrated in the apps such as *CrazyMachines GoldenGears Lite* (at least 500,000 installs). Searching the snippet on Google reveals that the potentially harmful code actually exists on Github. This indicates that highly likely `mappn`, a popular library for a famous Chinese app market, has been repackaged to spread potentially harmful code. We also note that it is very common that a library (sometimes very famous like Facebook) will refer another one to utilize its functionalities. Once the referred library is infected (usually less noticeable by developers), it is very dangerous for developers to include them in their apps (examples are in appendix).

Prior research shows that repackaging popular apps is the main channel for propagating Android PHAs [36], since the PHA authors can free-ride the popularity of the legitimate apps to reach out to a large number of Android users. A limitation of this approach is that the repackaged app usually cannot be uploaded to the market hosting the original version of the app,

particularly for the reputable place like Google Play, since this will cause the potentially harmful app to be easily exposed. By comparison, library repackaging does not have this problem. Very likely multiple apps in the same markets are all integrated with the same PhaLib. Also, such libraries could even be used by highly popular apps, such as *Unblock Me Free* and *Clash of Clans*, as discovered in our research, which unwittingly get infected when incorporating the contaminated versions of legitimate libraries.

Library embedding. We also found that some libraries contains other libraries (embedded libraries) to utilize their functionalities. In this case, once an embedded library is infected, its host library also becomes contaminated: e.g., `Adview` contains 42 libraries like `adwo`. Other examples include famous ad libraries like `admob` and `adsmogo`, each impacting millions of users around the world.

Our research demonstrates that library embedding is likely to be another channel for spreading potentially harmful code. Unlike direct attachment of the code to a library, placing it within a legitimate library integrated into another library is much stealthier. Such indirect contamination could propagate the harmful code even to highly reputable libraries: it is conceivable that the developers of such a library would not directly utilize the code from untrusted sources; however, as long as any libraries down its embedding chain are contaminated by a PhaLib, the library gets infected.

D. iOS PhaLibs

Repackaging. Our research shows that repackaged libraries are also extensively present on the Apple platform. Specifically, we found that 2 iOS libraries have both benign and potentially harmful variations. To better understand the code, we disassembled the library binaries and compared the code of different variations. We found that lots of potentially harmful behaviors disappear from the version of the library integrated within the apps on the official Apple Store. One possibility is that Apple's restrictive vetting process that forces some libraries, such as those for advertising, to become less aggressive. More probably, however, we could imagine that the adversary might infect some legitimate libraries and spread them across the Internet, which are picked up by less prominent developers to build the apps for regional customers (e.g., China). This is likely due to the great firewall of China that has significant impacts on the speed of the Internet for downloading the libraries from the websites outside the country. Interestingly, Chinese developers tend to obtain the toolkits from the domestic repositories, which just gives the adversary an opportunity to upload contaminated versions there. Indeed, on the third-party markets (particularly in the Chinese markets like 91 Market and 51ipa Market), the same libraries tend to contain more suspicious behaviors. An example is appflood, which has been embedded in 61 apps (e.g., *WildPuzzles, 123Karaoke and DotMatch*) on the Apple App Store and 53 apps like *Bulu Monster-1.3.0* on the third-party markets. Comparing the variations of the libraries across the apps on these two markets, it is interesting to see that even though they are clearly the variations of the same library (same names, 95% of identical methods and the same functionalities), the version of *lotuseed* on the 91 Market includes additional code accessing location, while the version used on the Apple Store does not have such behaviors.

Evasion strategy. We found that some PhaLibs apparently are designed to be less noticeable, trying to avoid explicitly asking for permissions from the user. As an example, within library lotuseed, we found that it has a unique strategy to perform the operations that need the user's consent (e.g., collecting fine user locations): the PhaLib does not call the API `requestWhenInUseAuthorization`, which will cause a window to pop up to seek the user's approval, and instead, just read the last retrieved location data from the host app in background, as if the permission had already been granted. In this case, if the app does not have the permission, then nothing will happen and otherwise, the location data will be obtained by the library. Either way, the user will not be notified of the behavior. In other words, the PhaLib leverages the consent an app already gets from the user (for its legitimate functionality) to execute potentially harmful actions. The similar strategy was also found in other PhaLibs.

Interestingly, we found that such behaviors exist on both iOS and Android side. For example, the iOS version of lotuseed has the aforementioned behavior and its Android version also checks the presence of permission android.permission. ACCESS_FINE_LOCATION before accessing a user's location. Only when the permission is granted, does lotuseed request for the user's accurate location and send it out to a remote server. Also the app checks android.permission.RECORD_AUDIO before recording audio (Figure 18).

V. Related Work

Mobile malware detection and prevention. A lot of effort has been made to analyze and detect Android PHAs, most of which aims at detecting the code [37], [38], [39], [40], [41], [42], [43], [44], [45] for performing potentially harmful behaviors (e.g., stealing the user's sensitive information), and protecting the Android system from various attacks [46], [47], [48], [49], [50], [51], [52], [53]. Unlike these prior studies, which mainly work at the app level, our research focuses on discovering and analyzing mobile libraries and their harmful behaviors, which has never been done before.

Compared with the research on Android, little has been done on the iOS front. Among few examples is PiOS [54], the first static tool to detect privacy leaks in iOS apps. When it comes to dynamic analysis, prior research [55] highlights the challenges in analyzing iOS programs. To address the issues, DiOS [56] uses UI automation to drive execution of iOS application, trying to trigger more events. Further, by combining dynamic binary instrumentation (through porting Valgrind [57] to iOS) and static analysis, iRiS [58] analyzes the API calls within iOS apps to find the abuse of private APIs. Other examples include MoCFI [59] that extracts the CFG of a program using PiOS and checks whether the instructions that change an execution flow are valid at runtime, and PSiOS[60] that enforces privacy

protection on top of MoCFI. None of the existing research seriously investigates potentially harmful apps on the Apple platform, largely due to the lack of ground truth.

Invariants inference. Invariants are widely used in finding a program's vulnerabilities [21], [24], [61], [25], which are typically discovered through dynamically analyzing the program [21]. More specifically, the program's runtime information is first collected to derive the invariable features before dynamically analyzing the program; any violation of the invariants detected in the analysis could indicate a program error. All the prior studies on this subject are performed on the same platforms, and most of the time on the same program or its variations. Never before has any attempt been made to identify cross-platform invariants for PHA analysis, which has been done in our research. Particularly, our invariant analysis is fully static, since this is the only way to make the analysis scalable, up to the job of processing 140,000 iOS apps.

Cross-platform analysis. Cross-platform security studies are known to be challenging. Still some work has been done in this area, mostly for vulnerability analysis. A prominent example is the recent work that finds similar bugs cross architectures (X86, MIPS, ARM) within the same program that can be compiled into different executables for those architectures [62]. By comparison, we are looking for potentially harmful code within the programs independently developed by different parties. Linking them together is clearly much more difficult. Also related to our work is the effort to establish a baseline for security comparison between Android and iOS apps, in which manually selected app pairs are analyzed to find out how they access security-sensitive APIs [63]. In our study, we have to develop technique to automatically correlate Android and iOS code and discover the harmful behavior both programs involve.

VI. Discussion

Understanding PhaLibs. All the iOS PhaLibs in our study were found from the invariants shared with their Android counterparts. The invariants used in our study, constant strings, were shown to be highly reliable, introducing almost no false positives. On the other hand, this approach does have false negatives, missing the iOS classes that do not share any strings with their Android versions. Even though this limitation has been somewhat made up by our extension technique, which starting from a few anchors, recovers 71.16% of the classes within a library, certainly some classes or even libraries fall through the cracks. A more comprehensive study could rely on the combination of the constant strings and other invariant features across the platforms, e.g., code structure and intermediate variables.

More fundamentally, the methodology of mapping Android PhaLibs to iOS apps misses the libraries built solely for the iOS platform. As an example, among the top 38 iOS libraries given by SourceDNA, 36 do have Android versions. We used this methodology for the purpose of understanding the relations between Android and iOS libraries and also leveraging VirusTotal to identify potentially harmful behaviors within iOS apps. The consequence, however, is that almost certain we underestimate the scope and magnitude of the PHA threat on the Apple platform. For example, our estimate of 2.94% of PHAs on the Apple App Store is very much on the low end. The real percentage could come much closer to that of Google Play. Further research on the iOS PhaLibs may require development of new techniques for a large-scale code similarity search over the iOS apps and comprehensive definitions of potentially harmful behaviors within these apps.

Identifying potentially-harmful behavior. Both the PhaLibs on the Android and iOS sides are identified, directly or indirectly, by VirusTotal. To determine the potential harmful behavior within an iOS library, we have to translate it into that inside an Android PhaLib and utilize a black-box technique to find out whether the behavior is part of a PHA signature. Such a translation and analysis, based upon IAC sequences, are less accurate, introducing 3.3% false positives and certainly missing the harmful activities that do not have the Android counterpart. Future research will explore a more effective way to export Android-side suspicious behaviors to the iOS platform, supporting direct PHA scanning on iOS apps. Also importantly, the techniques not relying on behavior and content signatures, as proposed in the prior research [64], should be developed for detecting the PHAs with previously unknown harmful behavior.

Evading. It is important to note that the techniques we used for PhaLib analysis across platform are *not* meant for PHA detection. As mentioned earlier, our approach only focuses on a subset of PHAs, those detected by VirusTotal on Android and those including the PhaLib with an Android counterpart on iOS. Even for such apps, the possibility is always there for the PHA authors to obfuscate their invariants to deter our cross-platform analysis. Future study is certainly needed to find a more reliable mapping, making such evasion harder. That being said, our study reveals the pervasiveness of PhaLibs and their cross-platform deployment, helping the research community better understand how such harmful code is propagated, a critical step towards ultimately defeating the threat they pose.

VII. Conclusion

This paper reports the first systematic study on mobile PhaLibs across Android and iOS platforms. Our research is made possible by a unique methodology that leverages the relations between the Android and iOS versions of the same libraries, which helps get around the technical challenges in recovering library code from iOS binary code and determining whether it is indeed potentially harmful. Running the methodology on 1.3 million Android apps and 140,000 iOS apps, we discovered 6.84% of PHAs on Google Play and 2.94% of PHAs on the Apple App Store. Looking into their behaviors, we discovered the high-impact back-door PhaLibs on both sides and their relations. Also we found further evidence that library contamination could be an important channel for propagating potentially harmful code. This study made the first step toward understanding mobile PhaLibs across the platforms and PHA detection on iOS.

ACKNOWLEDGEMENT

We thank our shepherd Franziska Roesner and anonymous reviewers for their valuable comments. We also thank VirusTotal for the help in validating suspicious apps in our study. Kai Chen was supported in part by NSFC U1536106, 61100226, Youth Innovation Promotion Association CAS, and strategic priority research program of CAS (XDA06010701). The IU authors are supported in part by the NSF CNS-1223477, 1223495 and 1527141. Yingjun Zhang was supported by National High Technology Research and Development Program of China (863 Program) (No. 2015AA016006) and NSFC 61303248.

REFERENCES

[1] Google, "Google report: Android security 2014 year in review," https://static.googleusercontent.com/media/source.android.com/en/security/reports/Google_Android_Security_2014_Report_Final.pdf, 2014.

[2] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale," in *USENIX Security*, vol. 15, 2015.

[3] SourceDNA, "ios apps caught using private apis," https://sourcedna.com/blog/20151018/ios-apps-using-private-apis.html, 2015.

[4] C. Xiao, "Novel malware xcodeghost modifies xcode, infects apple ios apps and hits app store," http://researchcenter.paloaltonetworks.com/2015/09/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/, Tech. Rep., 2015.

[5] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on android markets," in *ICSE*, 2014.

[6] "A private website," https://sites.google.com/site/phalibscom/, 2015.

[7] F-Secure, "Q1 2014 mobile threat report - f-secure," https://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q1_2014.pdf, Mar 2014.

[8] M. Kassner, "Google play: Android's bouncer can be pwned," http://www.techrepublic.com/blog/it-security/-google-play-androids-bouncer-can-be-pwned/, 2012.

[9] J. Erwin, "Where did virusbarrier ios go?" http://www.intego.com/mac-security-blog/where-did-virusbarrier-ios-go/, 2015.

[10] J. Leyden, "Apple is picking off ios antivirus apps one by one: Who'll be spared?" http://www.theregister.co.uk/2015/03/24/ios_anti_malware_confusion/, 2015.

[11] VirusTotal, "A closer look at mac os x executables and ios apps," http://blog.virustotal.com/2014/12/a-closer-look-at-mac-os-x-executables.html, 2014.

[12] T. iphone wiki, "Malware for ios," https://www.theiphonewiki.com/wiki/Malware_for_iOS, 2015.

[13] VirSCAN, "Virscan.org is a free on-line scan service," http://www.virscan.org/, 2015.

[14] S. Fadilpai, "Android is the biggest target for mobile malware," http://betanews.com/2015/06/26/android-is-the-biggest-target-for-mobile-malware/, 2015.

[15] AppBrain, "Android ad network stats," http://www.appbrain.com/stats/libraries/ad?sort=apps, 2015.

[16] "Admob for android, get started," https://developers.google.com/admob/android/quick-start#load_the_ad_in_the_mainactivity_class, November 2015.

[17] iana, "Root zone database," https://www.iana.org/domains/root/db, 2015.

[18] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *Kdd*, vol. 96, no. 34, 1996, pp. 226–231.

[19] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '12. New York, NY, USA: ACM, 2012, pp. 317–326. [Online]. Available: http://doi.acm.org/10.1145/2133601.2133640

[20] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 709–724. [Online]. Available: http://dx.doi.org/10.1109/SP.2015.49

[21] M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *Software Engineering, IEEE Transactions on*, vol. 27, no. 2, pp. 99–123, Feb 2001.

[22] "Clutch," https://github.com/KJCracks/Clutch.

[23] Capstone, "Capstone is a lightweight multi-platform, multi-architecture disassembly framework," http://www.capstone-engine.org/.

[24] S. Hangal and M. S. Lam, "Tracking down software bugs using automatic anomaly detection," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 291–301. [Online]. Available: http://doi.acm.org/10.1145/581339.581377

[25] Y. Kataoka, D. Notkin, M. D. Ernst, and W. G. Griswold, "Automated support for program refactoring using invariants," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ser. ICSM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 736–. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2001.972794

[26] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12. New York, NY, USA: ACM, 2012, pp. 93–104. [Online]. Available: http://doi.acm.org/10.1145/2381934.2381950

[27] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant, "Semantics-aware malware detection," in *Security and Privacy, 2005 IEEE Symposium on*, May 2005, pp. 32–46.

[28] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 217–228.

[29] E. M. Myers, "A precise inter-procedural data flow algorithm," in *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1981, pp. 219–230.

[30] VirusTotal, "Virustotal - free online virus, malware and url scanner," https://www.virustotal.com/, 2014.

[31] 91, "91 market," http://zs.91.com/, 2015.

[32] Apple, "Distributing apple developer enterprise program apps," https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/DistributingEnterpriseProgramApps/DistributingEnterpriseProgramApps.html, 2015.

[33] Lookout, "What you need to know about the new android vulnerability, stagefright," https://blog.lookout.com/blog/2015/07/28/stagefright/, 2015.

[34] P. G. Y. K. Zhaofeng Chen, Adrian Mettler, "ibackdoor: High-risk code hits ios apps," https://www.fireeye.com/blog/threat-research/2015/11/ibackdoor_high-risk.html, 2015.

[35] Dex2jar, "Tools to work with android .dex and java .class files," https://github.com/pxb1988/dex2jar, 2015.

[36] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *IEEE S&P*, 2012.

[37] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones." in *OSDI*, vol. 10, 2010, pp. 1–6.

[38] L.-K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis." in *USENIX security symposium*, 2012, pp. 569–584.

[39] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets." in *NDSS*, 2012.

[40] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2012, pp. 101–112.

[41] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1354–1365.

[42] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs,"

in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1105–1116.

[43] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1329–1341.

[44] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors," in *Proc. of the Symposium on Network and Distributed System Security (NDSS)*, 2015.

[45] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of android applications in droidsafe," in *Proc. of the Network and Distributed System Security Symposium (NDSS). The Internet Society*, 2015.

[46] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 2010, pp. 328–332.

[47] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 241–252.

[48] M. Dam, G. Le Guernic, and A. Lundblad, "Treedroid: A tree automaton based approach to enforcing data processing policies," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 894–905.

[49] M. Hardt and S. Nath, "Privacy-aware personalization for mobile advertising," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 662–673.

[50] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song, "Contextual policy enforcement in android applications with permission event graphs." in *NDSS*, 2013.

[51] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and fine-grained mandatory access control on android for diverse security and privacy policies." in *Usenix security*, 2013, pp. 131–146.

[52] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang, "Airbag: Boosting smartphone resistance to malware infection," in *NDSS*, 2014.

[53] O. Tripp and J. Rubin, "A bayesian approach to privacy enforcement in smartphones," in *USENIX Security*, 2014.

[54] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications." in *NDSS*, 2011.

[55] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna, "Challenges for dynamic analysis of ios applications," in *Open Problems in Network Security*. Springer, 2012, pp. 65–77.

[56] A. Kurtz, A. Weinlein, C. Settgast, and F. Freiling, "Dios: Dynamic privacy analysis of ios applications," Technical Report CS-2014-03 June, Tech. Rep., 2014.

[57] Valgrind, "Valgrind," http://valgrind.org.

[58] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "iris: Vetting private api abuse in ios applications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 44–56.

[59] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi, "Mocfi: A framework to mitigate control-flow attacks on smartphones." in *NDSS*, 2012.

[60] T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz, "Psios: bring your own privacy & security to ios devices," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, p. 1324.

[61] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, *Bugs as deviant behavior: A general approach to inferring errors in systems code*. ACM, 2001, vol. 35, no. 5.

[62] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *IEEE S&P*, 2015.

[63] J. Han, Q. Yan, D. Gao, J. Zhou, and R. Deng, "Comparing mobile privacy protection through cross-platform applications," in *NDSS*, 2013.

[64] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with jplag," *J. UCS*, vol. 8, no. 11, 2002.

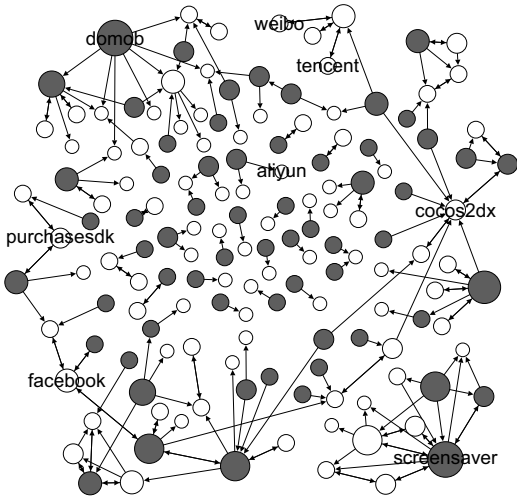

Fig. 12: *The call relation between libraries. Many benign libraries (white node) including those famous ones like Facebook, Coco2dx and Weibo are referring a PhaLib (black node).*

APPENDIX

A. Library Referring

We also note that it is very common that a library (sometimes very famous like Facebook) will refer another one to utilize its functionalities, which enforces a developer to download the referred one that they may not be familiar with. So it is more likely that the developer download a contaminated library. To understand the security implications of such library referring, we analyzed all the PhaLibs discovered in our study and the libraries referring them, as elaborated in Figure 12. In this figure, gray nodes represent the infected referred PhaLibs, the white nodes are the libraries referring the PhaLibs and arrows represent their referring relations. Altogether, we discovered 63 referred PhaLibs, which infect 188 libraries. Among them are highly popular libraries such as Facebook, Purchasesdk, Cocos2dk, Weibo and Tencent. For example, the Facebook library refers a library prime31, which were confirmed to be potentially harmful by VirusTotal and exhibits potentially harmful behaviors such as stealing device information, sending a message, and taking a picture. Also as we can see from the figure, a legitimate library can be infected by more than one referred PhaLibs and a PhaLib can also contaminate multiple legitimate libraries. An example is screensaver, a library blanking the screen or filling it with moving images. It is referred by 6 legitimate libraries (e.g., mobiware, wapsad, entrance, controller) and in the meantime, refers 2 other suspicious libraries (mobiware and wrapper). Further some libraries are found to be infected indirectly, such as sponsorpay.

B. Supporting Tables and Figures

iOS libs	Has android version
Adwhirl	Y
Interactive	Y
AdMob	Y
iAd	N
Flurry	Y
AdColony	Y
Millennial Media	Y
Jumptap	Y
Mopub	Y
Analytics	Y
Tag Manager	Y
App Events	Y
InMobi	Y
Localytics	Y
Unity	Y
Cordova	Y
Corona	Y
Adobe	Y
PhoneGap	Y
Marmalade	Y
Appcelerator Titanium	Y
Crashlytics	Y
Twitter performance metrics	Y
Twitter Beta	Y
Hockey	Y
New Relic	Y
Crittercism	Y
Bugsense	Y
Roboguice	Y
Facebook	Y
WeChat	Y
Pinterest	Y
Sina	Y
Dropbox	Y
MagicalRecord	N
Amazon	Y
Box	Y
Yandex	Y

TABLE VI: Top 38 libs (from SourceDNA) and whether they have Android versions.

Library	# Variations	# Pha variations
mobclick	181	131
inapp	109	109
gfan	97	97
jirbo	84	84
yrkfgo	53	53
admogo	42	42
sharesdk	27	27
pad	25	3
Leadbolt	24	24
adsmogo	20	20
V1_4	20	20
mobiware	19	19
appflood	14	14
adwo	13	13
zkmm	13	13
mobiSage	8	8
wanpu	7	7
zywx	7	7
imgview	5	5
prime31	5	5
clevernet	3	3
lotuseed	1	1
umpay	1	1

TABLE VII: PhaLibs on iOS and their variations

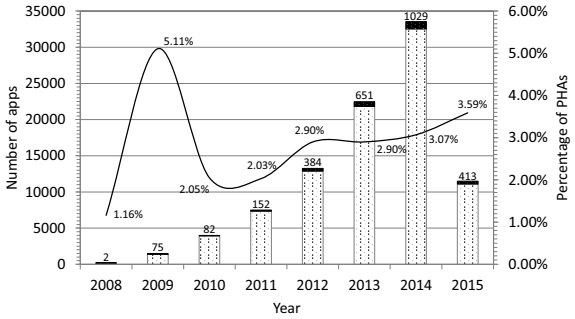

Fig. 13: *The distribution of such PHAs over the time they stay on the Apple Store.*

iOS: install .ipa file using private API

```
// WapsSilentRequest - (int)IPAInstall:(id)
int __cdecl -[WapsSilentRequest IPAInstall:](struct WapsSilentRequest *self,
SEL a2, id a3)
{
 v26 = a3;
 v8 =
dlopen("/System/Library/PrivateFrameworks/MobileInstallation.framew
ork/MobileInstallation", 1);
 if ( v8 )
 {
  v27 = (int)dlsym(v8, "MobileInstallationInstall");
  if ( v27 )
  {
   v25 = objc_msgSend(CFSTR("Install_"), "stringByAppendingString:",
v10);
    v14 = objc_msgSend(
          &OBJC_CLASS___NSDictionary,
          "dictionaryWithObject:forKey:",
          CFSTR("User"),
          CFSTR("ApplicationType"),
          v22);
    v28 = ((int (__fastcall *)(void *, void *, _DWORD, id))v27)(v25, v14, 0,
v26);
    ......
  }
 }
 return result;
}
```

Fig. 14: *Sample Code I (Install apps using private API).*

iOS: get installed app list using private API

```
// WapsRequest - (id)browseApps:(id) id __cdecl -[WapsRequest
browseApps:](struct WapsRequest *self, SEL a2, id a3)
{
  v4 = objc_msgSend(&OBJC_CLASS___NSMutableArray, "new");
  v5 = (struct objc_object *)objc_msgSend(v4, "autorelease");
  v6 =
dlopen("/System/Library/PrivateFrameworks/MobileInstallation.framew
ork/MobileInstallation", 1);
  if ( v6 )
  {
    v7 = dlsym(v6, "MobileInstallationBrowse");
    v8 = CFSTR("Any");
    if ( v3 )
      v8 = (__CFString *)v3;
    v9 = objc_msgSend(&OBJC_CLASS___NSDictionary,
"dictionaryWithObject:forKey:", v8, CFSTR("ApplicationType"));
    ((void (__fastcall *)(_DWORD, _DWORD, _DWORD))v7)(v9, callback,
v5);
  }
  return v5;
}
```

Fig. 15: *Sample Code II (Get installed app list using private API).*

iOS: get identifier related with Apple account

```
v2 = objc_msgSend(
&OBJC_CLASS___NSBundle,
"bundleWithPath:",
CFSTR("/System/Library/PrivateFrameworks/AppleAccount.framework"));
if ( (unsigned int)objc_msgSend(v2, "load") & 0xFF
    && (sprintf(&v9, "%s%s%s", "AA", "Device", "Info"),
        v3 = objc_msgSend(&OBJC_CLASS___NSString,
                          "stringWithUTF8String:", &v9),
        (v4 = (void *)NSClassFromString(v3)) != 0)
   && (v5 = objc_msgSend(&OBJC_CLASS___NSString,
        "stringWithFormat:", CFSTR("appleI%@tIdentifier"),
        CFSTR("DClien")),
        v6 = ((int (*)(void))NSSelectorFromString)(),
       (unsigned int)objc_msgSend(v4, "respondsToSelector:", v6) & 0xFF) )
{
  v7 = NSSelectorFromString(v5);
  result = (id)objc_msgSend(v4, "performSelector:", v7);
}
```

Fig. 17: *Sample Code IV (Get Identifier Related with Apple Account).*

iOS: access app's keychain

```
v7 = objc_msgSend(&OBJC_CLASS___NSMutableDictionary, "alloc");
v8 = objc_msgSend(v7, "init");
*((_DWORD *)v6 + 2) = v8;
objc_msgSend(v8, "setObject:forKey:", kSecClassGenericPassword,
kSecClass);
objc_msgSend(*((void **)v6 + 2), "setObject:forKey:", v4, kSecAttrGeneric);
if ( v5 )
  objc_msgSend(*((void **)v6 + 2), "setObject:forKey:",
v5, kSecAttrAccessGroup);  [identify keychain entry]
objc_msgSend(*((void **)v6 + 2), "setObject:forKey:", kSecMatchLimitOne,
kSecMatchLimit);
objc_msgSend(*((void **)v6 + 2), "setObject:forKey:", kCFBooleanTrue,
kSecReturnAttributes);
v9 = objc_msgSend(&OBJC_CLASS___NSDictionary,
"dictionaryWithDictionary:", *((_DWORD *)v6 + 2));
v12 = 0;
if ( SecItemCopyMatching(v9, &v12) )
{
  objc_msgSend(v6, "resetKeychainItem");
  objc_msgSend(*((void **)v6 + 1), "setObject:forKey:", v4,
kSecAttrGeneric, v12, v13, v14);
  if ( v5 )
    objc_msgSend(*((void **)v6 + 1), "setObject:forKey:", v5,
kSecAttrAccessGroup, v12);
}
```

Fig. 16: *Sample Code III (Accessing app's keychain).*

Android: permission evasion

```
@android.webkit.JavascriptInterface
public void startblow(String paramString)
{
  LogUtil.addLog("js startblow");
  int i
=this.context.checkCallingOrSelfPermission("android.permission.RECORD_
AUDIO") == 0 ? 1 : 0;
  if ((this.oldSdkListener != null) && (i != 0)) {
    this.oldSdkListener.startblow();
  }
}
```

Fig. 18: *Sample Code V (Permission evasion).*

Library	# Variations	# Pha variations	Library	# Variations	# Pha variations	Library	# Variations	# Pha variations
adfeiwo	3	1	gamesalad	8	1	payment	4	1
admob	7	1	gfan	17	3	paypal	2	1
admogo	1	1	giderosmobile	3	1	phonegap	41	1
adpooh	3	1	greenrobot	13	1	pkeg	1	1
adsdk	16	1	http	4	1	platoevolved	10	1
adsense	1	1	igexin	11	3	plugin	17	1
adsmogo	1	1	imadpush	2	1	prime31	100	9
adwo	2	1	imax	1	1	qumi	2	1
adzhidian	6	2	imgview	2	2	rabbit	5	1
airpush	12	3	inapp	2	1	revmob	31	1
androidannotations	2	1	jcraft	9	3	RMjDRvkz	1	1
androidnative	3	1	jirbo	17	1	rrgame	2	2
androidsoft	2	1	joymeng	2	1	screensaver	20	4
ansca	33	9	jpush	25	5	secapk	9	1
appchina	4	2	k99k	3	1	senddroid	7	1
apperhand	11	1	kobjects	12	1	sharesdk	31	1
appflood	11	1	ksoap2	24	1	skplanet	7	1
applovin	8	1	kxml2	2	1	surprise	3	1
Bigfool	3	1	kyview	8	2	swiftp	1	2
bugsnag	2	1	Leadbolt	22	4	tool	9	3
callflakessdk	2	2	letang	9	2	umpay	7	1
cczdt	1	1	lidroid	1	1	unisocial	1	1
cdfg	2	1	livegame	1	1	unity3d	28	1
clevernet	2	1	lotuseed	1	1	uucun	9	2
cnzz	5	2	lthj	5	1	v1_4	3	1
content	11	1	mappn	5	2	vpon	6	2
daoyoudao	3	1	margaritov	5	1	vserv	9	3
dash	1	1	measite	11	1	wanpu	3	1
define	7	1	mobclick	4	1	waps	1	1
demo	2	1	mobiSage	2	1	wapsad	4	1
disneymobile	3	1	mobiware	14	3	widget	3	1
dlnetwork	6	3	mongodb	6	1	winsmedia	3	1
domob	3	1	MoreGames	7	1	wrapper	6	2
esotericsoftware	24	1	neatplug	10	2	yoyogames	14	1
feiwo	3	1	newqm	4	1	yrkfgo	3	1
feiwoone	2	1	ning	2	2	zhuamob	4	1
fivefeiwo	1	1	novell	16	2	zkmm	1	1
flip	2	2	opengl	15	1	zong	7	2
framework	10	1	pad	2	1	zywx	7	1

TABLE VIII: PhaLibs on Android and their variations