12-2014

Android or iOS for Better Privacy Protection?

Jin Han

Qiang Yan

Debin GAO
Singapore Management University, dbgao@smu.edu.sg

Jianying Zhou

Huijie Robert DENG
Singapore Management University, robertdeng@smu.edu.sg

Citation

Han, Jin; Yan, Qiang; GAO, Debin; Zhou, Jianying; and DENG, Huijie Robert. Android or iOS for Better Privacy Protection?. (2014). *International Conference on Secure Knowledge Management in Big-data Era (SKM 2014), 8-9 December*. 1-10. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/2632

Android or iOS for better privacy protection?

Jin Han*, Qiang Yan†, Debin Gao‡, Jianying Zhou§, Robert Deng‡

*Software Engineer
Twitter Product Security Group USA

†Software Engineer
Google Switzerland

‡School of Information Systems
Singapore Management University
{*dbgao, robertdeng*}*@smu.edu.sg*

§Cryptography and Security Department
Institute for Infocomm Research
jyzhou@i2r.a-star.edu.sg

***Abstract*—With the rapid growth of the mobile market, security of mobile platforms is receiving increasing attention from both research community as well as the public. In this paper, we make the first attempt to establish a baseline for security comparison between the two most popular mobile platforms. We investigate applications that run on both Android and iOS and examine the difference in the usage of their security sensitive APIs (SS-APIs). Our analysis over 2,600 applications shows that iOS applications consistently access more SS-APIs than their counterparts on Android. The additional privileges gained on iOS are often associated with accessing private resources such as device ID, camera, and users' contacts.**

A possible explanation for this difference in SS-API usage is that privileges obtained by an application on the current iOS platform are invisible to end users. Our analysis shows that: 1) third-party libraries (specifically advertising and analytic libraries) on iOS invoke more SS-APIs than those on Android; 2) Android application developers avoid requesting unnecessary privileges which will be shown in the permission list during application installation. Considering the fact that an Android application may gain additional privileges with privilege-escalation attacks and iOS provides a more restricted privilege set accessible by third-party applications, our results do not necessarily imply that Android provides better privacy protection than iOS. However, our evidence suggests that Apple's application vetting process may not be as effective as Android's privilege notification mechanism, particularly in protecting sensitive resources from third-party applications.

I. Introduction

The current intensive competition among mobile platforms has sparked a heated debate on which platform has a better architecture for security and privacy protection. Discussions usually focus on Google's Android and Apple's iOS, which are the top two players in terms of user base [1], [2]. Some claim that Android is better since it makes the complete permission list visible to users and it takes an open-source approach [2]. Some argue that iOS is better because 1) Apple screens all applications before releasing them to the iTunes App Store (aka. Apple's vetting process); 2) Apple has complete control of its hardware so that OS patches and security fixes are more smoothly applied on all devices; and 3) the open-source nature of Android makes it an easier target of attacks than iOS [1]. Others [3], [4] suggest that the two platforms achieve comparable security but in different ways. These different voices clearly raise the need for establishing a *baseline* for security comparison among different mobile platforms. Unlike most prior efforts in comparing the abstract and general practices towards security [1], [2], [3], [4], we make the first attempt to establish such a baseline by analyzing the security-sensitive API usage on cross-platform applications.

A *cross-platform application* is an application that runs on multiple mobile platforms, e.g., the Facebook application has both an Android and an iOS version with almost identical functionality. We first try to identify these cross-platform applications by crawling information on both Google Play and iTunes App Store. Our web crawler collects information of more than 300,000 Android applications and 400,000 iOS applications. Several data mining techniques are adopted to match the applications released for the two platforms. We find that 12.2% of the applications on Google Play have a replica on iTunes Store. Among them, we select the most popular 1,300 pairs to further analyze their security-sensitive API usage.

A security-sensitive API (SS-API) is a public API provided for third-party applications that may have access to private user data or control over certain device components (e.g., Bluetooth and camera). In order to analyze the similarities and differences of the SS-API usage, the first challenge is to develop an SS-API mapping between Android an iOS. Based on the permission concept on Android and the existing Android API-to-permission mapping provided by Felt et al. [5], we group the SS-APIs on iOS into 20 different API types and map them to the corresponding Android SS-APIs. Our analysis produces a list of SS-API types that are both supported by Android and iOS. With such API mappings available, we statically analyze the cross-platform applications (Android Dalvik binaries and iOS Objective-C executables).

By analyzing the 1,300 pairs of cross-platform applications, which are sampled from the most popular applications, we show that 73% of them on iOS access additional SS-APIs, compared to their replicas on Android. The additional SS-APIs invoked are mostly for accessing sensitive resources such as device ID, camera, user contacts, and calendar, which may cause privacy breaches or security risks without being noticed. We further investigate the underlying reasons by separately analyzing third-party libraries and applications' own code. Our results show that the commonly used third-party libraries on iOS, especially the advertising and analytic libraries access more SS-APIs compared to the corresponding libraries on Android. Similar results are observed from the applications' own code. Further analysis shows that a likely explanation of such differences is that sensitive resources can be accessed more stealthily on the current iOS platform, compared to Android where all the privileges required by an application have to be shown to the end user during installation. We also discover, and confirm with the Android application developers, that SS-APIs may be intentionally avoided if the same functionality can be implemented by non-security sensitive APIs. These results suggest that Apple's vetting process may not be as effective as that most users think, particularly in protecting users' private data from third-party applications. This problem might also have been realized by Apple Inc., as the newly released iOS 6 has added privilege notifications for accessing user contacts, calendar, photos, and reminders.

II. Cross-platform Applications

A. Preliminary Data Collection

In order to find out what are the applications that exist on both Android and iOS, we need to compare their detailed information such as application name, developing company, application description, etc. The application product pages from Google Play and iTunes App Store do provide such information, although public APIs of obtaining this information do not exist. Thus, we build web crawlers for both Google Play and iTunes Store, and collect detailed application information for 312,171 Android applications and 478,819 iOS applications from April to May in 2012, which are further analyzed to identify cross-platform applications.

B. Identifying Cross-platform Applications

We consider two applications (one on Android and the other on iOS) to be two versions of the same *cross-platform* application if they have the same set of functionality. For example, both Android and iOS has a Facebook application that provides the same functionality.

To be able to handle the large number of candidate cross-platform applications, we first develop an automatic tool to find the most likely candidates by comparing their names, developer information, and the application descriptions. These candidates are categorized into five non-overlapping sets according to the degree of similarity in the three attributes, and we randomly select some candidate applications from each set and manually analyze the functionality of them for verification.

An interesting output of this analysis is that it enables us to estimate the total number of cross-platform applications on Android and iOS. Using the true positive rates obtained from our manual verification, we find that 12.2% (about one in eight) applications on Android have a replica application on iOS.

C. Stratified Sampling

To minimize the propagation of errors from the identification of cross-platform application into subsequent analysis, we focus our static analysis of cross-platform application on the candidate set that contains application pairs that have exactly the same name and developer information as well as a high degree of similarity in the descriptions. This set provides a total number of 20,171 cross-platform applications. The distribution of these applications among different categories is given in Figure 1, which is compared with the distribution of all applications on Android and iOS in the entire data set collected. As shown in Figure 1, cross-platform applications are more likely to appear in "Business" and "Games" categories, and are less likely to appear in "Books" or "Utilities".

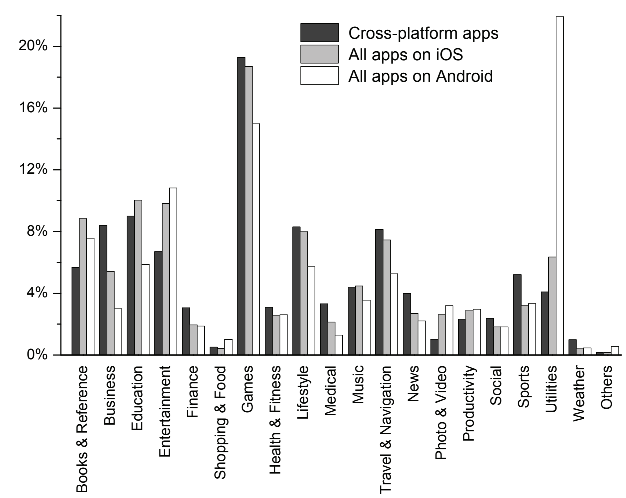

Figure 1. The distribution of the cross-platform apps vs. the distribution of all third-party apps on Android and iOS.

Among these 20,171 cross-platform applications, we select 1,300 pairs (2,600 applications) to perform detailed static analysis on the application executables. To improve the representativeness of this sample set, we perform a stratified sampling according to the category distribution of these cross-platform applications. We then pick the most *popular free* applications within each category. During the sampling, we also exclude applications that only work on tablets (e.g., iPad and Google Tablet) so that our analysis could focus on applications that are mainly developed for smartphones. Finally, we manually checked all the chosen pairs of applications to ensure that they are real cross-platform applications. The results of the static analysis on these selected applications will be presented in Section V.

III. Comparing Application Privileges

To compare the security architecture of Android and iOS, one of the most important comparison perspectives is to find out the similarity and difference on restricting the privileges for the third-party applications running on these platforms. However, it is not clear how such privileges can be compared as they might be of different granularity on the two platforms, and a mapping of them between the two platforms is not present in the literature. To make things more complicated, although Google provides a comprehensive list of application permissions for Android [6], there is no official documentation specifying what privileges are allowed for third-party applications on iOS – this is one of the iOS mysteries to be revealed in our work.

We choose to focus our work on Android 4.0 and iOS 5.0 which were both officially released in October 2011. Given the 122 application permissions supported on Android 4.0 [6], we first find out what is the exact privilege obtained in each permission by examining the functionality of all APIs related to this permission according to the mapping of Android permission to API[1] provided by [5]. We then carefully investigate both online advisories and offline iOS documentations on Xcode[2] to find out whether each privilege available on Android is supported, and how it is supported on the iOS platform. The overview of the analysis result is given in Table I.

Table I
A Classification of Android Application Privileges

Group of Privileges	#*	SS-API types
Does not actually exist in Android Already deprecated in Android, or no Android API corresponds to it.	**7**	SET_PREFERRED_APPLICATIONS BRICK
Reserved by Android system Only for OEMs, not granted to third-party apps. i.e., these privileges can only be used by apps signed with system keys.	**42**	DELETE_CACHE_FILES WRITE_SECURE_SETTINGS
Not supported on iOS Either iOS does not have such device e.g., removable storage; or iOS does not allow third-party apps to have such privilege.	**51**	CHANGE_NETWORK_STATE MODIFY_AUDIO_SETTINGS
Both supported by iOS and Android Third-party apps have these privileges on iOS as default.	**20**	BLUETOOTH READ_CONTACTS RECORD_AUDIO

* This column lists the number of SS-API types [6] in each privilege group.

Although the term "permission" used on Android platform is concise, it also implies that there is access control in the architecture, which iOS barely has[3]. Thus, in the rest of the paper, we use *SS-API type* to refer to a group of SS-APIs that require the same privilege to access certain private data or sensitive service. The name and scope for most of the SS-API types follow the official Android permission list [6] with three exceptions which will be explained in Section III-C.

As shown in Table I, among all the Android SS-API types, three of them (PERSISTENT_ACTIVITY, RESTART_PACKAGES and SET_PREFERRED_APPLICATIONS) have deprecated, and four of them (such as BRICK) do not really exist in Android, as there are no API calls, content providers or intents in Android related to these SS-API types [5]. The rest of the SS-API types are then divided into three groups according to our findings[4].

A. Privileges reserved for Android system applications

The openness concept of Android and its online documentations may have given a misleading understanding to users and developers that a third-party Android application can obtain any privilege. However, this is not true – many SS-APIs are only provided for original equipment manufacturers (OEMs), and are not granted to third-party applications. Examples of these API types include DELETE_CACHE_FILES, INSTALL_LOCATION_PROVIDER, FACTORY_TEST, etc.

Since there are no official documentations specifying which privileges are reserved for OEMs on Android, we identify this list of SS-API types by analyzing the protection level tags in the frameworks/base/core/res/AndroidManifest.xml file, as API types reserved for system applications are labeled as android:protectionLevel="signatureOrSystem" or android:protectionLevel="signature" in this firmware configuration file. In order to validate this list, a testing application is developed which tries to access all SS-APIs on Android, then those SS-API types that are denied to this application are recorded. Finally, 42 SS-API types are found to be reserved for system applications on Android, which are not granted to third-party applications unless users explicitly give them the root privilege.

B. Privileges not supported on iOS

Among the rest of SS-API types which can be used by Android third-party applications, we are interested in finding out how many of them are also supported by iOS.

[1]The mapping provided by [5] focuses on Android 2.2. We extend the mapping by adding the 10 additional permissions supported on Android 4.0 with a similar method introduced in [5].

[2]Xcode is a suite of tools from Apple for developing software for Mac OS X and iOS. It provides iOS API documentations for registered developers. See http://developer.apple.com/xcode/.

[3]Security entitlements are introduced for iOS applications from iOS 5, which are semantically similar to permissions. However, according to the latest official document [7], the accessible entitlements for third-party iOS developers only control iCloud storage and push notification. Though fine-grained entitlements are available on OS X to control access of private data such as address book and pictures, a third-party iOS application does not need such entitlements to access these data.

[4]The four groups of privileges listed in Table II are exclusive with each other. There could be more refined categorization in each group. E.g., privileges that are reserved by Android system can be further divided according to whether these privileges are supported on iOS. However, we do not further divide each group in Table II, as the focus of this paper is the privileges that are allowed to third-party applications and supported on both Android and iOS, which is the last row in the table.

Surprisingly, our analysis result shows that more than 2/3 of these SS-API types are not supported on iOS. The reasons are either because iOS does not have corresponding functionality/device, or iOS just does not allow third-party applications to have such privileges. Examples of SS-API types which are not supported on iOS are given in Table II.

Table II
EXAMPLES OF UNSUPPORTED SS-API TYPES ON IOS

Reason (1) iOS does not have corresponding functionality/device:		
SS-API type	Description	iOS Explanation
MOUNT_FORMAT_FILESYSTEMS	Allows formatting file systems for removable storage.	There is no removable storage for iPhone, iPad, or iPod Touch.
NFC	Allows applications to perform I/O operations over NFC.	Current iOS devices including iPhone 5 still do not have NFC chip.

Reason (2) iOS does not allow it to third-party applications:	
SS-API type	Description
KILL_BACKGROUND_PROCESSES	Allows an application to kill background processes.
PROCESS_OUTGOING_CALLS	Allows an application to monitor, modify, or abort outgoing calls.
RECEIVE_SMS	Allows an application to monitor, record or process incoming SMS messages.

It is interesting to notice that iOS does not allow some SS-API types to applications due to non-security reasons. Although it is not officially documented, APIs for changing global settings that would affect the user experience (UX) are usually disallowed by Apple, and that is one of the reasons why there are still many people who jailbreak their iPhones. Examples of such SS-API types include MODIFY_AUDIO_SETTINGS, SET_TIME_ZONE, SET_WALLPAPER, WRITE_SETTINGS, etc. Although this would limit the capability of third-party applications, it is still reasonable from the UX perspective. For example, it could be a disaster if you are waiting for an important call, but a third-party application mutes the sound globally without your awareness.

C. *Privileges supported by both Android and iOS*

The last group of privileges in Table I contains the SS-API types supported on both Android and iOS. A comprehensive list of these SS-API types is given in Table III. Note that although there are only 20 SS-API types both supported on Android and iOS, these SS-APIs cover the access rights to the most common resources/services, including user calendar, contacts, Bluetooth, Wi-Fi state, camera, vibrator, etc. As shown in Table III, due to the API difference on Android and iOS, the name and scope of three SS-API types have been changed compared to corresponding Android permissions [6].

The first refined SS-API type is ACCESS_LOCATION. On Android, there are two permissions correspond to the privilege of accessing the location information, which are ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION. There are 20+ API calls related to these two permissions on Android, but all of them only require *either* of the two permissions. Similar as Android, iOS devices employ a number of different techniques for obtaining information about the current geographical location, including GPS, cell tower triangulation and most inaccurate Wi-Fi connections. However, which mechanism is actually used by iOS to detect the location information is transparent to the application and the system will automatically use the most accurate solution that is available. Thus, for an iOS application which invokes the location-related API calls (e.g., CLLocationManager.startUpdatingLocation), it actually requires both ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION privileges. Therefore, we create the ACCESS_LOCATION SS-API type as a common privilege between Android and iOS, in order to perform a fair comparison.

Table III
SS-API TYPES SUPPORTED ON BOTH ANDROID AND IOS

SS-API Type	Abbr.	Description & Explanation
ACCESS_LOCATION	LOC	Allows to access the location info. This type corresponds to both ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION in [6].
ACCESS_NETWORK_INFO	ANI	Allows to access information about networks. This SS-API type corresponds to both ACCESS_NETWORK_STATE and ACCESS_WIFI_STATE in [6].
BATTERY_STATS	BAT	Allows to collect battery statistics.
BLUETOOTH	BLU	Allows to connect to bluetooth devices.
BLUETOOTH_ADMIN	BTA	To discover and pair bluetooth devices.
CALL_PHONE	PHO	Allows to initiate a phone call.
CAMERA	CAM	Allows to access the camera device.
CHANGE_WIFI_MULTICAST_STATE	CWS	Allows applications to enter Wi-Fi Multicast mode.
FLASHLIGHT	FLA	Allows access to the flashlight.
INTERNET	INT	Allows to open network sockets.
READ_CALENDAR	CAL	Allows to read the user's calendar data.
READ_CONTACTS	CON	Allows to read the user's contacts data.
READ_DEVICE_ID	RDI	Allows to read the device ID.
RECORD_AUDIO	RAU	Allows an application to record audio.
SEND_SMS	SMS	Allows to send SMS messages.
USE_SIP	SIP	Allows an application to use SIP service.
VIBRATE	VIB	Allows the access to the vibrator.
WAKE_LOCK	WAK	To disable auto-lock or screen-dimming.
WRITE_CALENDAR	CAL	Allows to write the user's calendar data.
WRITE_CONTACTS	CON	Allows to write the user's contacts data.

Similarly, Android provides APIs for checking the status (e.g., availability or connectivity) of different network types (e.g., WiFi or 3G). However, iOS APIs do not distinguish the different network types when checking the reachability of a given host or IP address. Thus, ACCESS_NETWORK_STATE and ACCESS_WIFI_STATE are combined into a single SS-API type – ACCESS_NETWORK_INFO to mitigate the bias when comparing the SS-API usage on these two platforms.

The last refined SS-API type is READ_DEVICE_ID. On Android, the scope of READ_PHONE_STATE permission corresponds to at least 18 Android API calls, which can be used to read the device ID, phone number, SIM serial number and some other information. However, on iOS, only device ID is allowed to read since iOS 4.0. Other information is forbidden to be accessed by third-party applications due to security reasons. Thus, we create the READ_DEVICE_ID type which only includes the SS-APIs on both platforms that access the device ID. By obtaining the list of SS-API types both supported on Android and iOS, we are now able to analyze the usage differences of these SS-APIs in cross-platform applications.

IV. STATIC ANALYSIS TOOLS

To compare the SS-API usage for third-party applications on Android and iOS, we build static analysis tools for both Android applications (Dalvik bytecode) and iOS applications (Objective-C executables). We explain the work flow of the static analysis on both platforms in this section.

A. Android Static Analysis Tool

Each Android application provides a list of privileges that is shown to the user during installation, which is recorded in the AndroidManifest.xml in each application package file. However, this is not the exact list of SS-API types that this application actually accesses – many third-party applications are overprivileged by requesting a superset of privileges [5]. Thus, the ultimate goal of our Android static analysis tool is to output a minimum set of SS-API types that are accessed by the given application. The work flow of our Android tool is shown in Figure 2.

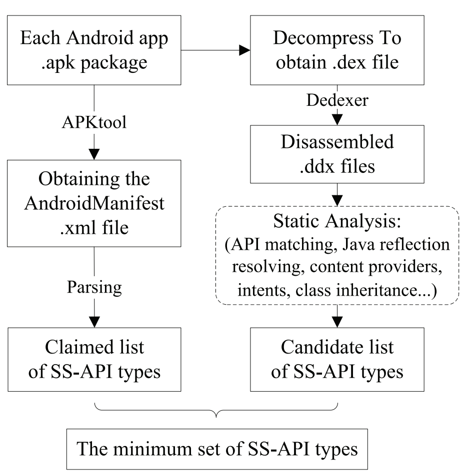

Figure 2. The work flow of our Android static analysis tool.

As shown in Figure 2, for each Android application, we first obtain the corresponding Dalvik executable (DEX), which is then disassembled into a set of .ddx files using the Dedexer tool [8]. With the extended Android API call to permission mapping [5], our tool then performs multiple iterations on parsing and analyzing the disassembled files to produce a candidate list of SS-API types that this application accesses. However, this candidate list is not a minimum set due to the ambiguity in the Android API-to-permission mapping, which is caused by Android's permission validation mechanism. For example, android.app.ActivityManager.killBackgroundProcesses API call requires either RESTART_PACKAGES or KILL_BACKGROUND_PROCESSES – i.e., either permission is sufficient for the application to invoke this API call. In order to further determine the exact privilege needed and output a minimum set of SS-API types, our tool then takes the intersection of the candidate list and the claimed list of SS-API types (parsed from AndroidManifest.xml). The output set of SS-API types is then used to compare with the set of SS-API types used by the replica application on iOS.

There are several technical challenges in analyzing the disassembled applications. On Android, SS-API calls may be invoked with different class names due to inheritance. By analyzing class information in the disassembled files, our tool rebuilds the class hierarchy so that it can recognize the API calls invoked from the applications' own classes, which are inherited from API classes. API calls may also be invoked through Java reflection. Our tool performs backward slicing [9] to resolve the method name and class name actually invoked in each reflection instance – it traverses the code backwards, resolving all instructions that influence the method variable and class variable used in corresponding reflection. We also apply specific heuristics to resolve inter-procedural or inter-classes reflections. Although it is not possible to completely resolve all reflections statically [10], fortunately Android applications rarely use reflections according to our observations. Finally, SS-APIs may be accessed through content providers and intents on Android. Our tool adopts the same mechanisms as Stowaway [5] to recognize the invocation of content providers and intents in the applications.

B. iOS Static Analysis Tool

Compared to Android, static analysis on iOS platform is more challenging, as iOS is a closed-source architecture. Apple tries to control all software executed on iOS devices (iPhone, iPad and iPod Touch), which has several effects. First of all, the only way for a non-jailbroken iOS device to install third-party applications is through iTunes App Store. When an application is downloaded via iTunes Store, it will be encrypted and digitally signed by Apple. The decryption key for the application is added to the device's secure key chain, so that each time this application is launched, it can be decrypted and then start to run on the iOS device.

It is not possible to directly perform static analysis on encrypted application binaries. Thus, before analyzing each application downloaded from iTunes Store, we need to obtain the decrypted application binary, which can only be achieved on a jailbroken iOS device. Jailbreaking gives us the capability to install the GNU Debugger, the Mach-O disassembler oTool and also the OpenSSH server on the device. These development tools enable us to crack any installed application on the device. After obtaining the decrypted iOS application binary, we utilize IDA Pro. [11] to disassemble the binary to obtain assembly instructions.

However, IDA itself is only able to mark a very small portion of Objective-C methods, especially when the symbols are stripped in the binary. The underlying reason is that iOS binaries are allowed to interchangeably use two instruction sets, ARM and THUMB, which have different instruction sizes and alignments. Without knowing the starting point of a method, IDA may treat a code fragment as a data entry by mistake. Thus, our analysis tool extracts metadata[5] from the application binary to guide IDA's disassembling process.

After disassembling all methods in IDA, the next step is to resolve all the API calls in the assembly instructions, where the key step is to handle the objc_msgSend function. In an Objective-C executable, all accesses to a method or attribute of an Objective-C object at runtime utilize this objc_msgSend function, which is used to send messages to an instance of class in memory [12]. To statically determine the corresponding API call for each observed objc_msgSend, we adopt the backward slicing and forward constant propagation proposed by [13] in our iOS static analysis tool. The work flow of our iOS static analysis tool is illustrated in Figure 3.

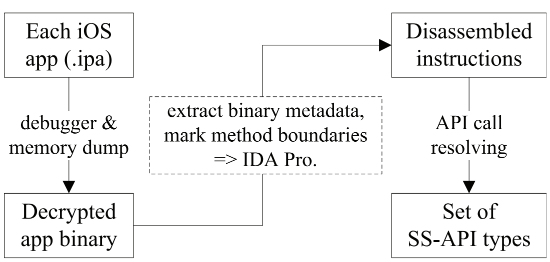

Figure 3. The work flow of our iOS static analysis tool.

The last step of our static analysis tool is to output the set of SS-API types used in an iOS application. The access to most SS-API types can be directly recognized through corresponding API classes and methods, for example, user contacts are operated through ABPerson and ABAddressBook related APIs. However, some SS-API types like CALL_PHONE and SEND_SMS require further analysis of the parameter value. For example, given an API call [[UIApplication sharedApplication] openURL:[NSURL URLWithString:[NSString stringWithFormat:@"tel:123-456-7890"]]], this will only launch the phone dialer when the string parameter starts with "tel:" prefix. The SS-API type SEND_SMS, however, has two forms of realizing the SMS functionality – the SMS sending view can be triggered by openURL with "sms:" prefix; an application can also call API such as MFMessageComposeViewController.setMessageComposeDelegate to send SMS messages. We carefully handle each of the cases for every resolved API call and corresponding parameter values in order to detect such SS-API invocations.

V. Comparison Analysis Results

We applied our static analysis tools to the 1,300 pairs of selected cross-platform applications (downloaded in June 2012), the basic statistics of these applications are given in Table IV. The direct outputs of our analysis tools are the lists of SS-API types accessed by these applications. By obtaining such lists, we are then able to compare the SS-API usage for each pair of cross-platform applications.

Table IV
Statistics of downloaded cross-platform applications

Parameters	Android apps	iOS apps
Number of apps	1,300	1,300
App size range (.apk & .ipa)	11KB∼47MB	106KB∼366MB
Total size of apps	7.42 GB	14.5 GB
App executable file size range (.dex & Objective-C binary)	3KB∼6.2MB	25KB∼39.5MB
Total size of executable files	1.10 GB	5.03 GB

A. *Comparisons on both-supported SS-API types*

Our first comparison focuses on the 20 SS-API types that are both supported on Android and iOS. We are interested in finding out how differently these SS-API types are used on the two platforms for cross-platform applications. Our results show that the total amount of SS-API types [6] that are used by 1,300 Android applications is 4,582, which indicates that each Android application uses 3.5 SS-API types on average. In comparison, the corresponding 1,300 iOS applications access a total amount of 7,739 SS-API types, which has on average 5.9 types per iOS application. 948 (73%) of the applications on iOS access additional SS-API types compared to its Android version.

Among the 20 different SS-API types, some of them are accessed almost equally by the applications on both platforms. For example, INTERNET is required by 1,247 Android applications, and 1,253 iOS applications. However, some other SS-API types are used much more often by iOS applications compared to Android applications. The top 10 SS-API types that are accessed more often on iOS compared to Android are listed in Table V.

[5]These metadata extracted include the class names, the instance method list, the class method list, the instance variable list, the property list and the protocols that classes conform to. For each method, the method name, method signature string, and the start address of the method body are collected to guide IDA disassembling process.

[6]The set of SS-API types used by an application contains no duplicates, which indicates that the maximum number of SS-API types used by each application is 20. When calculating the total amount of SS-API types for 1300 applications, we simply sum the number of SS-API types used by each application.

Table V
SS-API TYPES WITH GREATEST DISPARITY THAT ARE ACCESSED BY THE APPLICATIONS ON ANDROID AND IOS.

SS-API type	Number of Android apps	Number of iOS apps	**Only on iOS**[1]	Only on Android	On both platforms	Lib / App Ratio[2]	Exclusive Lib / App Ratio[3]
READ_DEVICE_ID	510	925	469	54	456	60% / 64%	36% / 40%
CAMERA	172	601	435	6	166	38% / 73%	27% / 62%
VIBRATE	374	522	290	142	232	62% / 46%	54% / 38%
ACCESS_NETWORK_INFO	885	1065	269	89	796	15% / 96%	4% / 86%
READ_CONTACTS	151	388	256	19	132	52% / 75%	25% / 48%
SEND_SMS	29	264	248	13	16	49% / 68%	32% / 51%
WRITE_CONTACTS	86	297	219	8	78	51% / 80%	20% / 49%
ACCESS_LOCATION	553	728	217	42	511	48% / 67%	33% / 53%
RECORD_AUDIO	37	177	155	15	22	35% / 99%	1% / 65%
READ_CALENDAR	35	174	141	2	33	35% / 67%	33% / 65%

[1] The number of cross-platform apps which access the corresponding SS-API type only in its iOS version, but not in its Android version. The value of this column equals to the difference between "Number of iOS apps" and "On both platforms".
[2] A break-down for the sources of SS-API types: Lib – from third-party libs; App – from the app's own code. The base of the ratio is column "Only on iOS".
[3] The ratio of applications in column "Only on iOS", where the corresponding SS-API type is exclusively caused by third-party libraries or apps' own code.

To obtain a detailed understanding of the results provided in Table V, we look into typical applications in each SS-API type. We find out that famous applications such as Twitter and XECurrency [14] do not access READ_DEVICE_ID APIs on Android. However, on their corresponding iOS version, we observe 5 locations in Twitter's code and 6 locations in XECurrency's code which read the device ID. Another typical instance is the famous free game "Words With Friends" [15] application. Compared to its Android version, the additional SS-API types accessed by its iOS version include (but are not limited to):

- BATTERY_STATS, as API call UIDevice.setBatteryMonitoringEnabled is observed;
- CALL_PHONE, as UIApplication.openURL with "tel:" parameter is observed in IMAdView.placeCallTo and two other locations;
- CAMERA, as UIImagePickerController.setSourceType with argument value 0x1 (which is UIImagePickerControllerSourceTypeCamera) is observed in MobclixRichMediaWebAdView.takePhotoAndReturnToWebview;
- FLASHLIGHT, as AVCaptureDevice.setTorchMode is observed in MobclixRichMediaWebAdView.turnFlashlightOnWithSuccess; etc.

More interestingly, we also check the most popular game application Angry Birds, although it does not belong to the 1,300 sampling set as it is not free on iOS. The result shows that compared to its Android version, Angry Birds on iOS additionally reads the user contacts data, as API call ABAddressBookGetPersonWithRecordID and ABAddressBookCopyArrayOfAllPeople are observed in the code section of CCPrivateSession.getArrayOfAddressBookEmailAddressesNamesAndContactIDs and four other locations[7].

[7]The API calls which access user contacts have been observed in all previous versions of Angry Birds including version 2.1.0, which was released in March 2012. However, from Angry Birds version 2.2.0 (released in August 2012), these API calls have been removed from this game, which is probably due to the privacy changes of the newly released iOS 6.

As shown in Table V, our findings in the comparisons on the 20 SS-API types both supported on Android and iOS show that iOS third-party applications turn to access more often to some devices (such as camera and vibration) and are more likely to access sensitive data such as device ID, user contacts and calendar. Thus, our next step of analysis is to find out the underlying reason why such phenomenon exists. As one may notice from the examples given above, some of these APIs are actually invoked by the third-party libraries used in these applications (such as IMAdView and MobclixRichMediaWebAdView classes in the WordsWithFriends application). Thus, our next step is to analyze the SS-API usage of the third-party libraries on both platforms.

B. SS-API Usage of Third-party Libraries

In order to analyze the SS-API usage of third-party libraries, first of all, we need to identify all the third-party libraries within each application. As there are no clear boundaries that an included library or package in a given application is written by the application developer or belongs to a third-party library, we first process the whole application set to calculate the number of different package names (on Android) or class names (on iOS). Then the packages or classes that appear in more than 10 applications (and at least belong to two different companies) are automatically collected. We then manually check this list to identify the third-party libraries, which include advertisement libraries, analytic libraries or just third-party development libraries. Some of the packages or classes are combined because they belong to the same third-party library. Finally, we identified 79 third-party libraries on Android and 72 third-party libraries on iOS that are commonly used. The 8 most commonly used advertising and analytic libraries on Android are listed in Table VI, and the 8 most common libraries on iOS are listed in Table VII.

By tracking the code regions of these libraries, our static analysis tools are able to determine the origin of the SS-API calls in each application. We can then identify the types of

Table VI

MOST COMMON ADVERTISING/ANALYTIC LIBRARIES ON ANDROID

Library Name	App Ratio	SS-API types*
com/google/ads	21.7%	ANI, INT
com/flurry/android	19.1%	LOC, INT
com/google/android/apps/analytics	12.5%	ANI
com/tapjoy	7.9%	INT, RDI
com/millennialmedia/android	7.3%	ANI, INT, RDI
com/admob/android/ads	4.4%	LOC, INT
com/adwhirl	3.8%	LOC, INT
com/mobclix/android/sdk	3.2%	LOC, ANI, INT, RDI

* The abbreviations of these SS-API types are given in Table III.

Table VII

MOST COMMON ADVERTISING/ANALYTIC LIBRARIES ON IOS

Library Name	App Ratio	SS-API types*
Flurry	19.9%	LOC, INT, RDI
GoogleAds	15.9%	ANI, INT, RDI, SMS, VIB, WAK
Google Analytics	9.8%	INT
Millennial Media	9.3%	LOC, ANI, CAM, INT, CON, RDI, VIB
TapJoy	9.1%	ANI, INT, RDI
AdMob	7.2%	LOC, INT, CON, RDI
AdWhirl	6.9%	LOC, ANI, INT, RDI
Mobclix	3.7%	LOC, ANI, BAT, CAM, FLA, INT, CAL, CON, RDI, SMS, VIB

* The abbreviations of these SS-API types are given in Table III.

SS-APIs used in each third-party library on both platforms, which are shown in Table VI and Table VII. The data from these two tables clearly indicate that libraries on iOS turn to access more SS-API types compared to Android third-party libraries. Thus, the SS-API usage difference for those cross-platform applications is indeed partially caused by the difference of third-party libraries.

To quantify the influence of the third-party libraries on the SS-API usage difference between the two versions of cross-platform applications, for each SS-API type, we first identify those applications which access the corresponding SS-API type only on its iOS version, but not on Android version (which is the "Only on iOS" column in Table V). For each of these applications, we then track the origins in the code which access the corresponding SS-APIs – either from the third-party libraries used in the application, or from the application's own code. The results are shown in the last two columns in Table V. The two ratios in the "Lib/App Ratio" column represent the percentage of the applications that: (a) the third-party libraries used in the application access the corresponding SS-API type; (b) the application's own code access the corresponding SS-API type. As can be seen from the table, the sum of these two ratios is more than 100%. This is because in some applications, SS-APIs belong to the same type are used both in the application's own code and in the third-party libraries. Thus, the last column in Table V is given, which shows the percentage of applications, where the corresponding SS-API type is only used by the third-party libraries or application's own code.

From the results shown in Table V, we can see that the third-party libraries do have certain impacts on the difference of the SS-API usage for cross-platform applications. For example, 54% applications which use additional VIBRATE APIs on iOS is purely because of the third-party libraries used in these applications. And from Table VII we can find the exact source – libraries such as GoogleAds, Millennial Media and Mobclix all use VIBRATE APIs. Thus, any application which includes these libraries will in turn use this SS-API type. Similar links can be drawn from Table V and Table VII for other SS-API types such as READ_DEVICE_ID and READ_CONTACTS.

Comparing the data in Table VI and Table VII, the results show that the most commonly used third-party libraries, especially advertisement and analytic libraries on iOS, access much more SS-APIs compared to the libraries on Android. A likely explanation of this phenomenon could be because *on iOS, the SS-APIs can be accessed more stealthily compared to on Android, where applications need to list out the types of SS-APIs they need to access during installation*. The privileges to use these SS-APIs on iOS are granted to third-party applications as default without users' awareness, which gives certain freedom for advertisement and analytic libraries to access user data and sensitive resources.

To confirm our findings on the third-party iOS libraries, we further check each library listed in Table VII to see whether it is an open-source library. For the open-source libraries (e.g., AdWhirl [16]), we manually look into their source code and confirm all SS-API types that are accessed. For the closed-source library Flurry, we also find evidences that this library collects the device ID in its official documentation [17], which mentioned "Because Apple allows the collection of UDID for the purpose of advertising, we continue to collect this data as the Flurry SDK includes AppCircle, Flurry's mobile advertising solution."

From the data given in the last column in Table V, one can observe that third-party libraries only contribute a portion of the difference of the SS-API usage for cross-platform applications; the other part of the difference is caused by the application's own code. By removing the SS-API types that are exclusively caused by third-party libraries, our static analysis tools manage to output the lists of SS-API types that are caused by the applications' own code on both platforms. The comparison result shows that 3,851 SS-API types are used by 1,300 Android applications in their own code, while iOS applications use 6,393 – there is still a significant difference for the SS-API usage on the two platforms. This difference leads us to investigate further into applications' code logic to find out the underlying reasons.

C. Microanalysis on Application Code Logic

In order to perform a manual analysis on the code logic of the cross-platform applications, it will be ideal to have full access to the application source code. However, applications which are open-source on both platforms are very rare, given the fact that iOS platform has very little open-source applications. Nevertheless, we manage to find 8 applications that are open-source on both platforms[8]. We retrieve the source code of these applications and analyze the underlying reasons of their SS-API usage differences. The detailed API information collected from closed-source applications is also utilized to assist the analysis. According to our manual inspection, there are at least two factors that have strong correlations with the SS-API usage differences between iOS and Android applications.

1) Coding difference: The most natural reason which may be expected is the implementation difference between the two versions of cross-platform applications. For example, ACCESS_NETWORK_INFO APIs are only used by the iOS version of WordPress, but not by its Android version. In its iOS version, several API calls in WPReachability class are invoked, which are used to test the reachability to the WordPress hosts. However, for the Android version of WordPress, there is no code for testing any reachability. For example, when posting a blog to the server, the code of Android WordPress simply checks the return value of the posting function to see whether the connection is successful or failed. But on iOS, many Objective-C classes in the WordPress code will actively check the reachability beforehand, and notify the users if the network is not reachable. Such implementation difference leads to the SS-API usage difference that WordPress on iOS uses the additional ACCESS_NETWORK_INFO APIs compared to its Android version. Similar evidence can be found in the source code of MobileOrg application.

Such coding difference is also the main reason causing the difference in using the CAMERA SS-APIs. Taking the popular applications such as eBuddyMessenger and SmackIt, in their iOS versions, the user profile photo in the setting can either be chosen from the pictures stored on device, or by directly taking the photo with the device's camera. However, their Android versions do not provide such photo taking option. Note that such implementation difference does not only exist in the applications' own code, but also for the same third-party libraries on two platforms. For example, CAMERA SS-APIs are used by OpenFeint library on iOS, but not by its Android version, which is caused by the same reason mentioned above.

2) Intentional avoidance: On the other hand, we also find evidences that even the functionality of the two versions of cross-platform application are the same, some SS-APIs are intentionally avoided to be used on Android. We use open-source application WordPress to explain this phenomenon.

Compared to its Android version, WordPress on iOS uses the additional READ_DEVICE_ID APIs. In the WordPress iOS code, runStats method of WordPressAppDelegate reads the uuid, os_version, app_version, language, device_model, and then sends them to http://api.wordpress.org/iphoneapp/update-check/1.0/ to check whether this application needs to be updated. On the Android platform, the code of WordPress performs the same functionality – in the wpAndroid class, uploadStats method tries to retrieve the same set of data and sends these data back to WordPress server to check for update. However, there is one major difference for the WordPress code on Android compared to the code on iOS. In its iOS code, the uuid is retrieved by directly calling UIDevice.uniqueIdentifier, which returns the device unique ID. In contrast, for its Android version, the uuid used is a random ID which is unique, but not the real device ID. It is a unique ID that is randomly generated and stored as the first record in WordPress's own SQLite database on the Android device. Thus, the different way of obtaining uuid is the reason that WordPress on iOS uses the additional READ_DEVICE_ID SS-API type.

The special way of obtaining the uuid in the Android version of WordPress makes us speculate that the programmers intentionally try to avoid using the READ_DEVICE_ID APIs on Android. This is further confirmed by consulting one of the WordPress developers, who gives the explanation as: "a random id is better than the device id because it doesn't require that permission which reads quite poorly as 'read phone state and identity' ". Thus, the reason that the developers do not try to avoid using the device ID on iOS is because of the same reason mentioned in Section V-B – on Android, an application needs to show the list of SS-API types it needs to access to the user during installation; while on iOS, no such notification is given to the user. We suspect that this may also be the main reason which causes the difference in accessing SS-API types such as READ_CONTACTS and READ_CALENDAR. But unfortunately, due to the limited access to applications' source code, we are not able to get the ground-truth evidence for these SS-API types, as what has been done for the READ_DEVICE_ID.

D. The Usage of SS-API Types Unsupported on iOS

Previous analyses focus on the 20 SS-API types that are both supported on Android and iOS, without taking into account of the additional 51 SS-API types that are only supported on Android platform. Thus, the last step of our analysis is to find out how frequently these SS-APIs are used by those Android applications, and what are usage characteristics of these SS-APIs.

Taking into account of the 51 SS-API types, our results show that the 1,300 Android applications use 1,230 SS-API types in total which are unsupported on iOS. As shown in Table VIII, the most frequently accessed SS-API type that is unsupported on iOS is WRITE_EXTERNAL_STORAGE, which is used by more than half of the Android applications. This can be explained from the nature of Android devices. Different from iOS devices which have 8GBytes to 64GBytes of

[8]These 8 open-source applications are WordPress, Mixare, MobileOrg, andRoc/iRoc, Mp3tunes, ZXing(Barcodes), DiceShaker and MobileSynth.

internal storage, Android devices usually have less internal storage. Thus, all Android devices support external storage such as microSD card. As a result, Android applications which want to store their application data usually need to utilize WRITE_EXTERNAL_STORAGE APIs to write to the microSD card, in order to save the internal storage space.

Table VIII
THE USAGE OF SS-API TYPES UNSUPPORTED ON IOS (TOP 8)

SS-API types unsupported on iOS	# of Android Apps
WRITE_EXTERNAL_STORAGE	762
GET_ACCOUNTS	133
RECEIVE_BOOT_COMPLETED	55
GET_TASKS	45
CHANGE_WIFI_STATE	44
READ_LOGS	15
RECEIVE_SMS	13
READ_HISTORY_BOOKMARKS	11

Except WRITE_EXTERNAL_STORAGE, the remaining 50 SS-API types that are not supported on iOS are used infrequently (only 468 in total, which is 0.36 per application on average). Such a result shows that the 20 SS-API types both supported on iOS and Android are the most commonly used SS-APIs for third-party applications. Note that although the SS-API types only supported on Android are not commonly used in popular Android applications, they may bring serious security breaches when utilized by malicious applications. For example, the READ_LOGS privilege, which allows a third-party application to "read the low-level system log files" can be utilized to read many other sensitive data such as SMS, contacts and location information as demonstrated in [18].

On the other hand, iOS also has its own specific SS-APIs that do not exist on Android. For example, two services, iCloud storage and push notification, which are controlled by entitlements are specific to the iOS platform. However, since the focus of this paper is on the SS-API types supported on both platforms, investigating the SS-APIs that are only supported on iOS is left as future work.

VI. CONCLUSION

In this paper, we made the first attempt towards systematically comparing mobile application security on diverse mobile platforms. In particular, the two most popular mobile platforms, Android and iOS, are chosen to investigate how the platform difference influences third-party applications in terms of privacy protection. As a prerequisite, we investigated the security-sensitive API (SS-API) types supported on iOS and their relations to Android application privileges, which were previously unclear. We then built our static analysis tools to perform massive static analysis for cross-platform applications on their SS-API usage.

Our analysis showed that applications on iOS tend to use more SS-APIs compared to their counterparts on Android, and are more likely to access sensitive resources that may cause privacy breaches or security risks without being noticed. Further investigation revealed a strong correlation between such difference and the lack of application privilege-list on the current iOS platform. Such results may imply that Apple's vetting process is not as effective as Android's explicit privilege list mechanism in restricting the privilege usage by third-party application developers.

REFERENCES

[1] "Trend Micro: Android much less secure than iPhone," electronista News, January 2011. http://www.electronista.com/articles/11/01/11/trend.micro.warns.android.inherently.vulnerable/.
[2] "Why Android App Security Is Better Than for the iPhone," PCWorld News. August 2010. http://www.pcworld.com/businesscenter/article/202758/why_android_app_security_is_better_than_for_the_iphone.html.
[3] "Android, iPhone security different but matched," cNET News, July 2010. http://news.cnet.com/8301-27080_3-20009362-245.html.
[4] "Smartphone Security Smackdown: iPhone vs. Android," InformationWeek, July 2011. http://www.informationweek.com/news/security/mobile/231000953.
[5] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 627–638.
[6] Android API level 14, Manifest.permission, http://developer.android.com/reference/android/Manifest.permission.html.
[7] "Apple, Entitlement Key Reference," http://developer.apple.com/library/mac/#documentation/Miscellaneous/Reference/EntitlementKeyReference/Chapters/AboutEntitlements.html.
[8] G. Paller, Dedexer, http://dedexer.sourceforge.net/.
[9] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*, 1981.
[10] J. Sawin and A. Rountev, "Improving static resolution of dynamic class loading in java using dynamically gathered environment information," *Automated Software Engineering*, vol. 16, pp. 357–381, June 2009.
[11] "IDApro, a multi-processor disassembler and debugger," Hex-Rays, http://www.hex-rays.com/products/ida/index.shtml.
[12] Nemo, "The Objective-C Runtime: Understanding and Abusing," phrack, Volume 4, Issue 66, http://www.phrack.org/issues.html?issue=66&id=4.
[13] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2011.
[14] XE Currency on iOS: http://itunes.apple.com/app/xe-currency/id315241195, XE Currency on Android: https://play.google.com/store/apps/details?id=com.xe.currency.
[15] Words With Friends Free, iOS version: http://itunes.apple.com/app/words-with-friends-free/id321916506, Android version: https://play.google.com/store/apps/details?id=com.zynga.words.
[16] AdWhirl Developer's Resources, https://www.adwhirl.com/home/dev.
[17] Flurry Product Updates, http://blog.flurry.com/updates/bid/33715/New-Flurry-SDK-Available-for-iPhone-OS-4-0-iOS.
[18] A. Lineberry, D. L. Richardson, and T. Wyatt, These Are Not the Permissions You Are Looking For, Def Con 18 Hacking Conference, 2010.