is article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2814999, IEE
Transactions on Dependable and Secure Computing

AdCapsule: Practical Confinement of Advertisements in Android Applications

Xiaonan Zhu, Jinku Li, Yajin Zhou, and Jianfeng Ma

Abstract—Nowadays, app developers tend to integrate advertisement libraries (or ad libraries) into their apps to get revenue from ad networks. However, researches have shown that both ad libraries and ad contents could raise serious security and privacy concerns. In this paper, we propose `AdCapsule`, a user-level solution to practically confine advertisements, including ad libraries and ad contents. Our solution does not need to change the Android framework, nor requires the root privilege, thus can be readily deployed. Specifically, we propose the `permission sandbox`, which isolates the permissions used by ad libraries from the host app, and the `file sandbox`, which separates the file operations of advertisements. The ad library and ad content cannot read or write any file outside this sandbox. We have implemented a prototype of `AdCapsule`. Our evaluation results indicate that `AdCapsule` can successfully enforce security policies to block attempts of accessing private information or manipulating files of the host app, and the performance overhead introduced by `AdCapsule` is low.

Index Terms—Security and privacy protection, ad library, permission sandbox, file sandbox

1 Introduction

It is no doubt that smartphones have become an essential part of our life. One potential reason of such popularity is that there are millions of applications (or apps) available to users. For instance, 2,600,000 apps are available in Google Play Store [1]. Among these apps, more than 90% are free ones [2]. Users download and use these apps without paying app developers.

To compensate for their work, app developers have incentives to integrate one or more advertisement (or ad) libraries in their apps (or host apps), and get paid. To facilitate the integration, ad service providers offer SDKs or libraries to app developers. App developers integrate these libraries with minor efforts, sometimes adding only a small amount lines of code. After that, ad libraries communicate with the ad network, retrieve and render ad contents.

Unfortunately, previous researches [3–12] have shown that both ad libraries and ad contents could raise serious security and privacy concerns to app developers and app users. First, as ad libraries run in the context of the host app, there is no clear security boundary between ad libraries and the host app. Ad libraries inherit all the permissions of the host app. As a result, ad libraries can probe and abuse permissions of the host app and behave aggressively [3, 7]. In addition, ad libraries and ad contents have incentives to read users' private information, to track and target users for advertisements [13, 14]. Second, besides permissions, ad libraries and ad contents can operate on any private file of the host app, and directly get or indirectly infer users' personal information. Previous studies revealed that malicious ads could read sensitive information [6, 8], e.g., geographical location and chat logs, by accessing files of the host app. What's worse, ad libraries may have vulnerabilities which could be exploited by attackers to attack the host apps and users [4, 15, 16]. For instance, the PontiFlex ad library was reported vulnerable that could be exploited for arbitrary command execution [4]. We will give examples of aggressive and vulnerable ad libraries that stimulate our work in Section 3.

One potential way to address this problem is that app developers could carefully vet ad libraries before integrating them. However, this is not a practical solution, due to the following reasons. First, these libraries (and the ad contents) are usually obfuscated and not easy to be statically analyzed. Most app developers do not have time and technical skills to analyze these libraries. Second, ad libraries could bypass the static vetting process by using dynamic code execution technique [7] and executing aggressive payloads at runtime.

Accordingly, researchers have proposed several solutions [17–19] to solve the security and privacy concerns of ad libraries. AdDroid [18] and AdSplit [19] are two representative ones. AdDroid is a privilege-separated ad framework for the Android platform. It introduces new APIs for ad libraries and corresponding permissions in Android platform. By doing so, the privileged operations of ad libraries are implemented in a system service and totally separated from the host app. To adopt the system, ad libraries, apps using ad libraries and the underlying Android framework need to be changed. AdSplit, on the other side, isolates ad libraries into another process with different user identification. Ad libraries and the host app are two different apps, hence it could leverage the Android application UID mechanism to isolate them from each other. AFrame [20] further isolates not only process/permission, but also the

- *This work was supported in part by the Natural Science Basic Research Plan in Shaanxi Province of China under Grant 2015JM6351 and the Key Program of National Natural Science Foundation of China under Grant U1405255.*
- *Xiaonan Zhu, Jinku Li, and Jianfeng Ma are with the School of Cyber Engineering, Xidian University, Xi'an 710071, China. E-mail: xnzhu@stu.xidian.edu.cn; jkli@xidian.edu.cn; jfma@mail.xidian.edu.cn.*
- *Yajin Zhou is with the Cyber Security Research Center and School of Computer Science, Zhejiang University, Hangzhou 310027, China. E-mail: yajin@vm-kernel.org.*

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

inputs and display. These systems may solve the problem. However, one serious limitation, i.e., the modification to the Android framework, obstructs practical deployment of these systems. Users have to flash custom ROMs to adopt these systems, which is nearly impossible for normal users. Even in the case that phone vendors want to integrate these systems, the well-known Android fragmentation [21] problem makes the deployment to different phone models and Android versions a time-consuming process.

In contrast, PEDAL [22] leverages bytecode rewriting technique to rewrite ad libraries and apply security policies for the privacy-concerned APIs. It does not need any change to the ad library or the underlying Android framework, thus is easy to be deployed. However, this system is not a complete solution, and could be bypassed due to its design. First, static code rewriting could be bypassed by ad libraries using dynamic code execution technique, or through the exported JavaScript interface. This has been demonstrated by existing ad libraries [3, 7]. Second, sensitive information could be inferred by malicious ads through reading private files of the host app, or publicly accessible files on SDCard, without invoking any privacy-concerned API [8]. Thus, we need a practical and complete solution to confine advertisements, including ad libraries and ad contents.

In this work, we propose `AdCapsule`, a user-level solution to confine ads in Android apps. Our solution does not need to change the Android framework, and it does not require the root privilege, thus can be readily deployed. Specifically, it leverages two sandboxes, i.e., the **`permission sandbox`** and the **`file sandbox`** for the purpose. The permission sandbox isolates permissions used by ad libraries from the ones used by the host app. In this case, ad libraries are running inside an isolated sandbox with their own permissions and contexts, thus cannot abuse permissions of the host app to perform aggressive operations. The file sandbox separates file-related operations of ad libraries and ad contents. All the file read and write operations are confined inside a sandbox. They cannot touch any file outside this sandbox. By doing so, our system ensures that ad libraries and ad contents cannot directly retrieve or indirectly infer any private information by accessing files of the host app.

We have implemented a prototype of `AdCapsule`. Our implementation is based on three key techniques, including binder hooking, in-VM API hooking and GOT hooking [23] to reliably regulate permission-related and file-related operations. Specifically, to intercept the privacy-concerned APIs, `AdCapsule` proposes a technique called binder hooking which leverages Java dynamic proxy to reliably hook Android framework APIs and apply security policies. Compared with systems using bytecode rewriting [22, 24–27] to intercept framework APIs, `AdCapsule` provides a robust and tamper-resistant way to hook these APIs, even in the case of code obfuscation, Java reflection and dynamic code execution. To regulate file-related operations, `AdCapsule` first leverages in-VM API hooking to redirect corresponding APIs to our own implementation. This works for the APIs in ad libraries, however, does not work for the ad contents accessing local files through WebView component and JavaScript interface. Further, `AdCapsule` hooks the GOT table of the WebView component to regulate the file access of rich-media ads. After intercepting privacy-concerned and file-operation APIs, `AdCapsule` enforces security policies to allow or deny such access, or provide bogus values in some cases.

To evaluate the effectiveness and compatibility of `AdCapsule`, we apply `AdCapsule` on 500[1] representative apps that embed the top 10 ad libraries according to AppBrain [28]. Our evaluation results indicate that `AdCapsule` can successfully detect and block ad libraries from accessing users' private data. Moreover, file operations from ad libraries and ad contents are isolated, and attempts to manipulate local files have been detected and blocked. `AdCapsule` is compatible with existing ad libraries and popular apps, and the performance overhead is low.

In summary, this paper makes the following contributions:

- We propose a practical solution called `AdCapsule` to confine advertisements, including both ad libraries and ad contents, without the need to change the Android framework or existing ad libraries.
- We propose two user-level sandboxes, i.e., the permission sandbox, to isolate the privacy-related operations of ad libraries, and the file sandbox to confine the file operations of ad libraries and ad contents.
- We have implemented and evaluated a prototype of `AdCapsule`. The evaluation results demonstrate the effectiveness and compatibility of `AdCapsule`. And it incurs low performance overhead.

The rest of the paper is structured as follows: we first introduce the necessary background information in Section 2. We then present the motivation examples and provide a case study of existing ad libraries in Section 3. We illustrate the design, implementation, and evaluation of our system in Section 4, Section 5, and Section 6, respectively. We discuss the limitation and potential improvements to our system in Section 7 , and the related work in Section 8. Finally, we conclude the paper in Section 9.

2 Background

In this section, we briefly introduce the key concepts of Android system, ad libraries and ad contents to provide necessary background information of the proposed system.

2.1 Android Permission Model

Android is an open source system using Linux kernel. Each app is running within a separated runtime environment[2], and has its own unique running environment. When a new app starts, the pre-initialized zygote process forks a new process, and loads all native libraries (e.g., libc) and the Android framework which contains all the framework APIs into memory. Note that the runtime is not the security boundary. The security boundary is the app sandbox based

1. Among these 500 apps, 395 ones can successfully integrate our system, and 105 ones fail mainly due to the integrity check (see Section 6.1).

2. This runtime environment is called Dalvik runtime in old Android versions and ART runtime in Android versions above 5.0.

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2814999, IEEE Transactions on Dependable and Secure Computing

on Linux UID mechanism. Due to this, an app could intercept native libraries in the app's own process space without any security constraints. That is what we do in the in-VM API hooking and GOT hooking.

Android takes advantage of the underlying Linux operating system to isolate different apps and their resources by making each app run in its own Virtual Machine with an unique UID. Unless explicitly specified, one app cannot read other app's files by default. To communicate with system services and perform dangerous operations, an app should require corresponding permissions. And these permissions should be granted by users when the app is being installed. For instance, if the app wants to send SMS messages in the background, then the app should have the `SEND_SMS` permission.

Ad libraries and the host app are running inside a same virtual machine and sharing the same privileges. Thus, ad libraries could probe and abuse permissions of the host app [3], which causes privacy concerns to users, and further ruins the reputation of the host app. However, in order to get revenue from the ad network, app developers have no choice but trusting and integrating ad libraries. We need a practical solution to confine these libraries.

2.2 Ad Library and Ad Content

To integrate ad libraries into host apps, app developers need to register and get an account from ad service providers (ad providers). This account is the unique identification of the app (and the app developer). Then app developer downloads the SDK provided by ad provider and integrates the SDK into the host app. The SDK communicates with ad provider's network, retrieves and renders ad contents in the host app. Note that, ad contents fetched from the network are usually provided by advertisers, not the ad provider.

Ad libraries require some extra permissions to work properly. The host app has to request these permissions on behalf of ad libraries. For instance, ad libraries need to connect to the remote server to retrieve ad contents at runtime. Hence, the host app has to request the `INTERNET` permission for ad libraries. Also for the purpose of location-based ads, location-related permissions are required by ad libraries, including the `ACCESS_COARSE_LOCATION` and the `ACCESS_FINE_LOCATION`. In addition, ad libraries could probe permissions that the host app has and behave accordingly. One way to do this is that ad libraries invoke the framework API (`checkCallingOrSelfPermission`) to check whether the host app has particular permissions, e.g., `SEND_SMS`, and then behave aggressively if the app has such permissions.

To provide rich-media ads, ad libraries usually leverage the WebView [29] component to display images, videos, and export native functionalities, i.e., framework APIs, to ad contents through the Java-JavaScript bridge. WebView acts like a lightweight web browser which can be embedded in an app. Since Android 4.4, the WebView component is mainly based on Google's Chromium open source project. It has most of Chromium's features such as the JavaScript engine, which can be used by ad contents to do operations dynamically at run time. Similar to other web browsers, the WebView component has the privilege separation mechanism based on the same origin policy. Hence, the JavaScript of ad contents cannot read any content from other origins. To bridge the JavaScript code and the native functionalities of the app, Android provides the `JavascriptInterface` mechanism [30]. But this mechanism raises security concerns due to two reasons. First, if not carefully implemented, the `JavascriptInterface` mechanism could introduce vulnerabilities. For instance, for apps targeting API level 17 and below, the JavaScript code of malicious ad contents could leverage any exposed `JavascriptInterface` to execute arbitrary payload [31] using Java reflection, with permissions of the host app. In addition, many ad contents are loaded over a clear text channel. Attackers could launch Man-in-the-Middle attack to hijack ad contents and inject malicious ones [4]. Second, ad contents fetched and executed in the host app's context are provided by advertisers. Malicious ad contents could abuse this mechanism to invoke dangerous functions through explicitly exposed `JavascriptInterface` by ad libraries. This risk is usually missed by previous works which only confine ad libraries, and it motivates our system to confine ad contents as well.

3 Motivating Examples

In this section, we first give examples of real threats caused by ad libraries and ad contents, and then we report the state of the art of privacy-related behaviors in ad libraries. Both the real threats and the current situation of ad libraries motivate our work to confine advertisements in apps.

3.1 Aggressive Ad Libraries

As stated before, since ad libraries and the host app share same permissions, ad libraries could abuse the host app's permissions and behave aggressively.

Plankton Plankton [3] is an ad library that has been included into apps in Google Play Store. It leverages the dynamic code loading capability to download and execute bytecode at runtime. Due to this, it is hard for the bytecode rewriting systems to confine such behaviors. Specifically, when the library is loaded into execution, it probes permissions requested by the host app, and then uploads the obtained permission list to a remote server. After that, the library downloads a Jar file containing the payload (the dex bytecode) and loads the payload into execution. By doing so, the downloaded payload could evade static analysis and make it hard to be detected. According to the report [3], the downloaded payload has the capability to create shortcut, access browser bookmarks, collect local log information, etc.

Taomike Taomike is a popular mobile ad solution platform in China and the company provides ad library to help developers render rich-media ads. However, this ad library was found to steal SMS messages and upload the messages to a remote server without users' consent [32]. Specifically, this library registers a receiver to monitor the event of receiving SMS messages, and then uploads SMS messages to a remote server controlled by the company. SMS messages often contain highly sensitive information, e.g., the authorization code for mobile banking purpose. Hence, this behavior is considered beyond legitimate functionalities of an ad library.

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

his article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2814999, IEEE Transactions on Dependable and Secure Computing

3.2 Vulnerable Ad Libraries

In the following, we will describe one type of vulnerable ad libraries. This is due to the bridge between JavaScript and native framework APIs.

PontiFlex Ad Library PontiFlex [4] is an ad library embedded in many apps, and was found to be vulnerable due to the `JavascriptInterface`. Specifically, this ad library initializes a list of JavaScript interface which connects to the underlying native framework APIs. As a result, the fetched ad content could use JavaScript to invoke framework APIs, which otherwise are not accessible from the JavaScript interface. However, the ad library is vulnerable because the ad contents are fetched through HTTP protocol without any integrity check. As a result, attackers could hijack the fetched ad contents and inject malicious JavaScript code into the ad contents. Then due to the exported framework APIs through the `JavascriptInterface`, attackers could remotely execute arbitrary commands in the context of the host app, with all permissions the host app has. PontiFlex is not the only one that is vulnerable, previous research also revealed other vulnerable ad libraries [33, 34].

3.3 Malicious Ad Contents

Ad libraries usually leverage the WebView component to render fetched ad contents. Ad contents are confined using the same origin policy of the WebView component.

However, ad contents could infer sensitive information through other ways. Recently, Son *et al.* showed that malicious ad contents could infer browser history by accessing cached images and HTML files on external storage [8]. Specifically, to reduce network traffic, the Dolphin browser caches images in external storage, and ad contents could use JavaScript to determine whether a file of a website exists on the storage. By doing so, malicious ad contents could determine whether a particular website has been accessed by users using the Dolphin browser. This is just one of the proposed attacks shown in the paper. Other attacks include ways to infer social graph, gender preferences for dating partners, user trajectories, etc.

3.4 Study of Existing Ad Libraries

Though we have shown security concerns in previous sections reported by other researchers, we want to further understand the state of the art of ad libraries and their privacy-related behaviors. To this end, we collected 27 world-wide ad libraries according to their popularity on AppBrain [35]. Due to the special nature of the Android ecosystem in China, e.g., highly customized Android systems without Google services, we also collected 29 ad libraries which are popular in China. Moreover, to check whether ad libraries perform more privacy-concerned operations than other libraries, we collected 119 popular development tool libraries on AppBrain for comparison.

To automatically check the behaviors of each library, we wrote a python script for this purpose. Specifically, we first download the latest SDK provided by each ad provider. The downloaded SDK usually is in the format of a Jar file containing Java class files. Then our program translates the class files into Java source files. At last, for each privacy-related behavior we are interested, we develop a corresponding signature. Our program uses the signature to analyze the ad library. For instance, to check whether the ad library accesses user's geographical location to provide location-based ads, our program leverages the signature of well-defined framework APIs, including `getLastKnownLocation`, `requestLocationUpdates`, `getCellLocation`, etc. We acknowledge that our program is conservative and may miss privacy-concerned behaviors in ad libraries. For instance, if the ad library leverages code obfuscation, dynamic code execution or Java reflection to invoke framework APIs, these behaviors may not be detected by our program. Nevertheless, our study still provides an insight of the low boundary of the privacy-concerned behaviors of ad libraries we studied.

TABLE 1
Results of sensitive operations of ad libraries and development tool libraries

	Chinese Ad Libs	World-wide Ad Libs	Development Tool Libs
Read/Write Contacts	3%(1/29)	0%(0/27)	7%(8/119)
Read SMS	3%(1/29)	0%(0/27)	0%(0/119)
Send SMS	48%(14/29)	19%(5/27)	5%(6/119)
Read Location	72%(21/29)	59%(16/27)	5%(6/119)
Read IMEI	97%(28/29)	44%(12/27)	6%(7/119)
Read Phone Number	10%(3/29)	3%(1/27)	0%(0/119)
Read Accounts	0%(0/29)	11%(3/27)	3%(4/119)
Read App List	55%(16/29)	33%(9/27)	0%(0/119)
DexClassLoader	17%(5/29)	7%(2/27)	2%(2/119)
WebView	93%(27/29)	85%(23/27)	9%(11/119)

Table 1 shows the results. First, we found that reading IMEI number is a common behavior in ad libraries. In addition, some ad libraries are reading phone numbers. We believe this is because ad service providers use this type of information to create unique identification of each user (or each device). Second, compared to development tools, the percentage of the behavior of location access is surprisingly high in ad libraries. This may because ad libraries need location information to provide more precise ad targeting, e.g., location-based advertisements. Though accessing location is a common practice in ad libraries, we argue that this behavior should be confined, since both ad libraries and advertiser could leverage location information to precisely and continuously track users.

Moreover, some behaviors cannot be justified in ad libraries. There are no legitimate reasons for ad libraries to send SMS messages in the background, read users' SMS messages and retrieve application list. For instance, around 48% (14 of 29) of Chinese ad libraries have the capability to send SMS messages in the background, and the percentage of world-wide ad libraries is 19% (5 of 27). Reading installed app list does not require any permission, but raises privacy concerns because ad libraries could leverage the list to infer user's private information [6]. We also found that, `DexClassLoader` is used by 5 Chinese ad libraries and 2 world-wide ad libraries to dynamically load bytecode into execution. These libraries cannot be easily confined by static bytecode rewriting tools, since the loaded bytecode could be remotely downloaded at runtime. These aggressive behaviors show the demanding need to practically confine

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

is article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2814999, IEEE Transactions on Dependable and Secure Computing

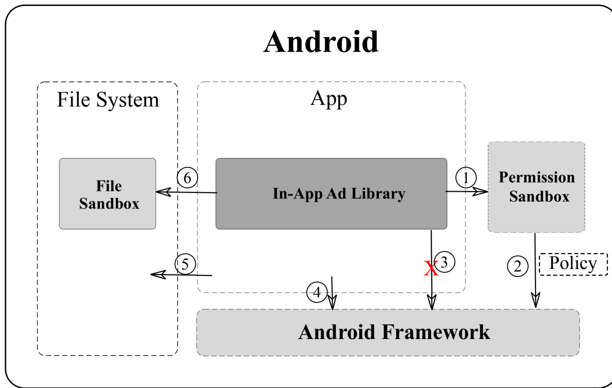


Fig. 1. The overall design of AdCapsule

ad libraries.

4 System Design

4.1 Overview

Threat Model In this work, in-app ad libraries and ad contents are untrusted. Ad libraries are untrusted either because they are aggressive or malicious, or they are not malicious but vulnerable. For better ad targeting, ad libraries have the motivation to aggressively track users, using the retrieved identifier, location data or the list of installed apps. While for ad contents, they are untrusted because they are possibly malicious. Note that ad contents fetched from the ad network are usually provided by advertisers, not the ad providers. Advertisers could leverage malicious ad contents to infer user's sensitive information by accessing user's location or local files using JavaScript [8]. App developers are trusted and they want to get revenue from ad providers by integrating ad libraries into their apps. At the same time, they have the incentive to maintain the reputation of their apps and regulate these libraries. Similar to other works, we trust the underlying Android framework and the operating system.

Overall Design The goal of our work is to confine advertisements, including in-app ad libraries and ad contents. To this end, we propose a user-level solution to sandbox ads, which does not need the change of underlying Android framework. Specifically, our system consists of two different sandboxes. The first one is the `permission sandbox`, which regulates permissions that could be abused by ad libraries. This sandbox intends to regulate the privacy-concerned operations of ads. The other one is the `file sandbox`, which provides a separate area of files for ads. Advertisements cannot read or write any files outside the sandbox. This sandbox prevents ads from directly reading private files of the host app [6], or inferring user information based on the determination of existence of particular files [8].

Figure 1 shows the overall design of our system. The direct interaction between ad libraries and sensitive Android framework APIs is not allowed (③). Our system intercepts such interactions and applies security policies specified by app developers (① and ②), while the host app itself can invoke the underlying framework APIs freely (④). For file operations, our file sandbox allocates a special area for advertisements. Only this special area could be read or written by ad libraries (⑥). Any file access outside this special area will be blocked by our system. However, the host app is not restrained by `AdCapsule` and can operate on files normally (⑤). In the following, we will describe these two sandboxes respectively.

4.2 Permission Sandbox

The main purpose of the permission sandbox is to regulate privacy-concerned behaviors of ad libraries (and ad contents) and enforce security policies at runtime. There are several design challenges in our permission sandbox. First, we need to bridge semantic gaps between privacy-concerned behaviors of ad libraries and the underlying Android framework APIs. For instance, if we want to regulate the behavior of accessing location, we need to find corresponding framework APIs that could be used to obtain locations. Second, our system intercepts interactions between ad libraries and the underlying Android framework to enforce security policies. But reliable interception of Android APIs is difficult since ad libraries could use Java obfuscation and dynamic code loading to invoke framework APIs and evade regulation. Third, we need to obtain the context of framework API calls, and distinguish whether the invocation is from app code or from ad libraries. We only need to regulate framework API invocations from ad libraries.

Bridging Semantic Gaps Fortunately, Android framework APIs have a well-defined mapping with privacy-concerned behaviors. For instance, in order to send SMS messages in the background, the app should invoke the framework API `SmsManager.sendTextMessage`. In our system, for each sensitive behavior that needs to be confined, we find the corresponding Android framework APIs. Then we hook the invocation of such APIs in ad libraries and add reference monitor code to check and enforce security policies, e.g., block the access.

Binder Hooking After finding corresponding APIs of privacy-concerned behaviors, our system needs to intercept such APIs. Previous systems either modify the Android framework [36, 37] or leverage bytecode rewriting [22, 24–27, 38] for this purpose. However, statically rewriting the bytecode cannot deal with dynamic code loading at runtime. In our system, we propose a new mechanism called `binder hooking`, which reliably hooks the Android framework managers and proxies their APIs. It does not require extra efforts to support dynamically loaded classes, and naturally tolerates obfuscation of the app's bytecode because the app eventually needs to call those interposed APIs to be effective.

Our binder hooking is implemented through Java dynamic proxy mechanism [39]. By leveraging this mechanism, our system proxies different manager objects, e.g., `LocationManger`, and applies security policies in the proxy manager objects. Specifically, for each privacy-concerned API, we find the corresponding manager object and put a proxy there. Then we add reference monitor code in the proxy and control the behavior of ad libraries. Similarly, to confine ad contents in the WebView, we hook and proxy the

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

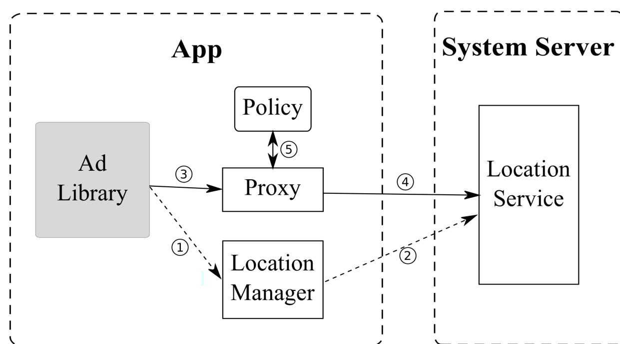


Fig. 2. Binder hooking of LocationManager

`WebSettings` in the WebView, and configure the WebView's setting through our proxy object. WebView in ad libraries will be restricted by its WebSettings. After that, privacy-concerned operations of ad contents in the WebView are confined. For instance, ad contents cannot access user's location.

To illustrate the whole process, we use the hooking process of the `LocationManager` as an example. When an ad library wants to obtain one user's location, it queries the `ServiceManager` and get the `LocationManager` object through the framework API `getSystemService`. Then the ad library communicates with the returned `LocationManager` and the `LocationManager` launches RPC call to the system server to get the current location (① and ② in Figure 2). To interpose this process, our system uses a proxy object to replace the original `LocationManager` object. All the requests from the ad library goes through to the proxy first (③). Then the proxy object checks the predefined policies (⑤). If the operation is allowed, then the request goes through to the system server (④). Otherwise, the request is blocked or a fake location is returned according to security policies.

Context Aware Policy Enforcement After interposing the APIs, our system enforces security policies specified by app developers. For instance, if the ad library is not allowed to read the geographical location, our system enforces this policy in the reference monitor.

However, the app code and the ad library is running inside the same process and can invoke the same set of framework APIs. For instance, like ad libraries, the app code could use the same framework APIs to obtain the current location. Our system needs to distinguish the actual caller of the API, i.e., the context of the API invocation, to enforce right policies. In our system, we leverage the call stack to infer the context of certain API calls. Specifically, if the trace of the call stack contains the package name of the ad library, then the function is invoked from the method in the ad library. Figure 3 shows the function call stack from the AdMob library to get WebView's WebSettings. From this stack, we can find that this function call contains the method `run()` inside the class `com.google.android.gms.internal.fa$2`, which is in the AdMob library. After getting the context of the API invocation, our system enforces security policies accordingly.

Note that multi-thread is not an issue for our system.

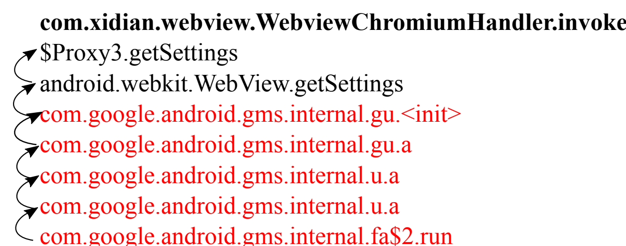


Fig. 3. The call stack of getSettings of WebView from the AdMob library

Our system checks the call stack, and we can get the stack trace from the `run()` function of a new thread. If the new thread originates from the ad library, it will contain the ad library's package name and we can successfully identify the ad library.

4.3 File Sandbox

The purpose of the file sandbox is to regulate all the file operations of ad libraries and ad contents. As a result, ad libraries and ad contents cannot manipulate any private files of the host app on the internal storage, or public files on the external storage. All the files that ad libraries could access are in a special area of the file system. However, we cannot use the binder hooking technique introduced in permission sandbox (Section 4.2) to interpose file operations, since file access mainly happens in the framework layer (and the WebView component) without going through the system server managers. To interpose file operations, we introduce other two techniques, i.e., `in-VM API hooking` and `GOT hooking`.

In-VM API Hooking In-VM API hooking is implemented through direct manipulation of VM's internal data structures. The VM maintains a data structure for each Java class in the app, through which we can find the methods of the class. To interpose framework APIs (Java methods), our system finds the corresponding method pointers in the class, i.e., `Method` on Dalvik and `ArtMethod` on ART, and replace them with our own implementation.

Specifically, in order to operate on files, an ad library needs to use file-IO APIs of the Android framework. All these APIs eventually go to the methods inside the `IoBridge` class. We hook file-IO related methods inside this particular class. For APIs which pass a file path as the parameter, we apply a sandbox namespace prefix to the path. As a result, all files that ad library can touch is inside the sandbox with this special namespace. For instance, if the ad library opens the file with the path */data/data/com.hostapp/files/files_open*, then the actual file opened is */data/data/com.hostapp/ad_sandbox/files/files_open*. Of course, to prevent the ad library from escaping the sandbox using the path traversal technique, we need to sanitize the file path in our sandbox.

GOT Hooking Besides file operations from ad library, ad contents could also access files using HTML or JavaScript. In this case, the file access from HTML and JavaScript eventually goes through the WebView component and accesses the files from the native layer of the WebView. Our system leverages GOT hooking to intercept file access from the native layer of WebView. Specifically, the WebView component

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

his article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2814999, IEEE Transactions on Dependable and Secure Computing


```
1  cls = Class.forName("android.os.ServiceManager");
2  Method getServiceMethod = cls.getDeclaredMethod(
3            "getService", String.class);
4  IBinder rawBinder = (IBinder)getServiceMethod.invoke(
5            null, SERVICE_NAME);
6
7  BinderProxyHookHandler bh = new BinderProxyHookHandler(
8            rawBinder, interfaces);
9  bh.setmInterfacehandler(createBinderProxyHookHandler(
10           rawBinder, stubClass));
11 IBinder hookedBinder = (IBinder)Proxy.newProxyInstance(
12           cls.getClassLoader(),
13           new Class<?>[]{IBinder.class}, bh);
14
15 Field sf = cls.getDeclaredField("sCache");
16 sf.setAccessible(true);
17 Map<String, IBinder> c =
18           (Map<String, IBinder>)sf.get(null);
19 c.put(SERVICE_NAME, hookedBinder)
```

Fig. 4. The code snippet of binder hooking

links to its native libraries such as libc.so for file operations. We hook the GOT table of WebView's native libraries and redirect the file-related APIs to our own implementation. We can redirect all the file operation of ad libraries into the file sandbox.

However, both the app and the ad library could use the WebView component, and it is not an easy task to distinguish the request source in our reference monitor code in the native layer. For the Java framework APIs, our system leverages the stack trace to distinguish whether a request coming from the ad library or the app code itself (see the Context Aware Policy Enforcement in Section 4.2). However, in the native layer, it is difficult to reliably obtain the stack trace. To solve this problem, we leverage the `shouldInterceptRequest` callback in the WebView component. Specifically, this API is called when the WebView component accesses the local and remote resources, including local files. We can easily distinguish the request source in this callback based on the Java stack trace because this callback is in the Java layer. For the file access request from the app code, we open and return the file resource, and for the file access request from the ad library (and the ad content), it passes through to the native layer of the WebView and is regulated by the GOT hooking of our system.

5 Implementation

We have implemented a prototype of `AdCapsule`. The permission sandbox and file sandbox are implemented in Java and C++ programming languages. The security policies are implemented in XML format. In this section, we will illustrate the detailed implementation of this prototype.

5.1 Permission Sandbox

As discussed in Section 4, the permission sandbox leverages binder hooking which is implemented using Java dynamic proxy mechanism. For each manager object our system is hooking, we create a corresponding proxy class to accomplish the hooking before the initiation of the app's code. Figure 4 shows the code snippet of binder hooking. Take the `TelephonyManager` as an example, `AdCapsule` first obtains the original `IBinder` object of the `TelephonyManager` by querying the local `ServiceManager` class (lines 1-5) and the value of the parameter `SERVICE_NAME` is a constant value `Context.TELEPHONY_SERVICE`. The returned `IBinder` object is packed into the proxy handler using the Java dynamic proxy technique [39] (lines 7-13). The method `createBinderProxyHookHandler` in line 9-10 return a handler (`TelephonyHandler`), which will be put into a Java map, the static field `sCache` of the `android.os.ServiceManager` class (lines 15-19). After that, when the ad library obtains the `TelephonyManager`, the system will lookup the Java map inside the `android.os.ServiceManager` class. In this case the `hookedBinder` instead of the original one will be returned, and the context will be checked and the policy will be enforced afterwards.

5.2 File Sandbox

Binder hooking cannot directly apply to the file sandbox since file operations do not go through the manager object of the system service. To this end, we propose in-VM API hooking and GOT hooking to regulate file related operations.

In-VM API Hooking Our in-VM API hooking mechanism is implemented in a native library similar to AndFix [40] and YAHFA [41]. Our system detects the underlying runtime and performs differently. For file operations implemented in the Java layer that need to be hooked, our system finds the `Method` structure in the VM using the native method `GetMethodID` on Dalvik and `FromReflectedMethod` on ART. Then it creates a stub Java method which contains the reference monitor of our system for target method and swaps these two method structure pointers. For example, if we want to hook a Java method on Dalvik, we can swap the data structure of the stub method and the target method directly, which is always encoded inside the `Method` structure. When it comes to ART, it is a bit complicated. We have to define the corresponding data structure of the `ArtMethod` on each version of ART first, and then find and swap the `entry_point_from_interpreter_`, `entry_point_from_jni_` and `entry_point_from_quick_compiled_code_` of the stub `ArtMethod` and the target `ArtMethod`. By doing so, the invocation of the original Java method will be redirected to the stub method. And the stub method then checks the context of the invocation. If the invocation is from the ad library, then our system applies corresponding security policies.

GOT Hooking In-VM API hooking can deal with file-related operations of ad libraries in Java. However, it cannot deal with the file operations from the ad content rendered by the WebView component, since these file-related operations eventually go to the native layer. To this end, we apply GOT hooking to intercept file access of the WebView component in the native layer. Specifically, our system first finds the base address of the WebView component `libwebviewchromium.so` in the app's memory space using its own memory map (`/proc/self/maps`). As the `.so` file `libwebviewchromium.so` on Android is in the ELF format, our system locates the GOT table of this library in the memory. Then we find the base address of the target method in GOT table and changes the corresponding GOT entry of file-related operations such as `open`, `close` to our own

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

```
1  webview.setWebViewClient(new WebViewClient() {
2    @Override
3    public WebResourceResponse shouldInterceptRequest(
4        WebView view, String url) {
5      WebResourceResponse response = null;
6      if (isFilepath(url)&&isFromApp(getcallstacktrace())){
7        response = new WebResourceResponse("", "",
8            getPath(url));
9      }
10     return response;
11   }
12 });
```

Fig. 5. The code snippet to distinguish the context of file operations

stub code. Our stub code adds a prefix to the file path. By doing so, we redirect all the file-related operations to our own implementation in the native layer.

However, one challenge is that we need to distinguish the context of app code and ad library in the native layer. In our system, we leverage the `shouldInterceptRequest` API for this purpose. Figure 5 shows the code snippet. For each URL to be loaded, we first check whether this is a file path. If so, we then check the context using the stack trace (in Function isFromApp() - line 6). If the request comes from the app code, we directly open the file and return the resource (lines 7 - 8). Otherwise, a null response is returned (line 10) and the WebView will open the file using native functions, which will be intercepted by our GOT hooking and redirected to the file sandbox.

5.3 Confined Operations and Security Policies

We follow previous works [42, 43] to obtain the permission-API mapping. Table 2 shows the privacy-concerned operations confined by our current prototype. It is relatively easy to extend this list by hooking more APIs if needed. For example, our system can monitor `TelephonyManager`'s `getDeviceId`, `getLine1Number`, `getImei`, `getSubscriberId`, `getSimSerialNumber`, `getCellLocation`, `getNetworkType`, `getAllCellInfo`, `getCellNetworkScanResults`, `getLine1NumberForDisplay` and `getDefaultSim` methods in the current prototype. If we want to monitor more methods in the `TelephonyManager`, we just need to add these API names into the `TelephonyManager`'s API name list and define the corresponding fake return values. If we want to extend our system to more managers, we need to copy the code and adjust the manager name, API name list and corresponding return values. We leave the extension of this list as one of our future works.

For each privacy-concern behavior that is regulated, our system provides different types of security policies. In the current prototype, our system supports three different configurations: `allow`, `deny` and `none`. The `allow` configuration means that the access of the API should be granted. `Deny` means that our system denies the access to such API. But for different APIs, our system acts differently according to the type of the return value of the API. For example, if the security policy of the API to send SMS messages is denied (the return value is `null`), our system blocks it immediately. But when it comes to the API to obtain device serial number (the return value is `String`), a fake value will be returned.

TABLE 2
Confined operations in our system

Hooked Service Manager	Confined Operations
ActivityManager	Contentprovider to access contacts and SMS
CameraManager	Camera related operations
ClipboardManager	read or write clipboard
LocationManager	access location
NotificationManager	display notification
SmsManager	send SMS
TelephonyManager	read phone information
PackageManager	read app list
AssetManager	access app resources
WifiManager	operate on Wifi

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <root>
3  <infor id="">
4    <PackageName>com.package.adlibrary</PackageName>
5    <ManagerName>
6        android.telephony.TelephonyManager
7    </ManagerName>
8    <MethodName>getDeviceId</MethodName>
9    <Permission>deny</Permission>
10 </infor>
11 <infor id="">
12   <PackageName>com.package.adlibrary</PackageName>
13   <ManagerName>
14       android.app.NotificationManager
15   </ManagerName>
16   <MethodName>enqueueNotificationWithTag</MethodName>
17   <Permission>allow</Permission>
18 </infor>
19 </root>
```

Fig. 6. An example of security policy

This is useful because ad libraries expect a String value is returned and they need to use the returned value to function properly. In the case, returning a fake value to ad libraries is a better choice since the ad library can function well, while at the same time user's privacy is protected. If the policy sets `none` or has not been defined for one behavior, a prompt window will be displayed at runtime to alert users that the ad library is doing something which needs to be confirmed. This means our system supports both predefined policy by app developers and runtime confirmation of app users.

In the prototype, all the policies are specified using an XML file when integrating our system. Figure 6 shows an example policy used in our prototype. From this policy, the ad library is not allowed to obtain the real IMEI number of the device (a fake IMEI number will be returned), while it is allowed to display notification.

5.4 Impact of Code Obfuscation

As the deployment of our system is during the app's development cycle by app developers, they have to provide security policies based on package/class names of ad libraries when integrating our system. However, code obfuscation tools, e.g., Proguard, is enabled by default in Android studio, and the package/class names of ad libraries could be changed by the tool, which will impact the specified security policies. We developed a python script to help developers to analyze package names of ad libraries before and after code obfuscation based on Proguard's map file. Similar map files also exist in other code obfuscation tools. Table 3 shows an

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

his article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2814999, IEEE Transactions on Dependable and Secure Computing

TABLE 3
Package Names of Chartboost

Before Obfuscation	After Obfuscation
com.chartboost.sdk	com.b.a
com.chartboost.sdk.impl	com.b.a.e
com.chartboost.sdk.InPlay	com.b.a.a
com.chartboost.sdk.Libraries	com.b.a.b
com.chartboost.sdk.Model	com.b.a.c
com.chartboost.sdk.Tracking	com.b.a.d

example of package names of Chartboost library before and after code obfuscation by Proguard.

5.5 Anti-modification

Since our system has the same privilege with the ad library, and they are running inside the same process, malicious ad libraries could bypass our system leveraging the similar hooking mechanisms we used. Further, ad libraries can invoke the services or remove our system by native code. Our system took this into consideration and proposed several mechanisms for self-protection. First, our system is the first component that is started in the app. Like other systems, this is done by using the Application class [44, 45]. Second, our system hooks all the related methods of Java reflection, e.g., Class.forname() and Method.invoke(), so that the ad library cannot use Java reflection to change the proxies we have placed. Third, ad libraries are not allowed to load native libraries into execution. This is a reasonable assumption since the ad library does not have legitimate reasons to load native code. In fact, only 1.06% ad libraries [22] are using native libraries, and we leave it as a future work to confine the native code of ad libraries.

5.6 System Deployment

Our system is mainly developed for app developers. In particular, app developers integrate our library together with ad libraries into their projects, and add a few lines of code to activate `AdCapsule` when the app is started. After that, developers can define the security policies according to ad library's package name list and put the policy file into the raw apk file. Finally, developers can sign the apk file and upload it to the app store. More scenarios of our system can be found in Section 7.

Developers may set the policy of `none` to some operations. In this case, users should take care the prompt windows when using the app. These windows will tell users that the ad library is performing some operations that need to take further actions.

6 Evaluation

In this section, we evaluate the effectiveness, compatibility and the performance overhead of our system.

TABLE 4
Evaluation Results on Real Applications

Operations	Total	Ad Libraries
File	240	191
TelephonyManager	74	15
PackageManager	325	311
AssetManager	280	3
NotificationManager	69	1
LocationManager	53	9
AccountManager	10	0
WifiServiceMessenger	48	0
Contact	5	0
SMS	3	0
WebView.getSettings	101	62

6.1 Effectiveness

Real Apps and Ad Libraries To evaluate the effectiveness of our system, we downloaded 500 apps from Google play store. These apps are reported to contain ad libraries. Specifically, we downloaded the top 50 apps in each of the top 10 categories according to AppBrain. Then we develop a tool to repackage the app to include `AdCapsule`. To this end, we first decompile the apk into smali files using Apktool [46]. And then we inject our system into these apps by changing these smali files. After that, we convert the smali files to the apk file again. At last, we generate a public-private key pair and sign the generated apk file. Note that, this repackaging process is only for evaluation purpose. The main deployment of our system is during the app's development cycle by app developers.

For 500 apps we downloaded, after integrating our system using the repackaging method described previously, 59 apps failed to generate apps and 46 crashed. To further investigate whether the crash is caused by `AdCapsule`, we repackaged these 105 apps without integrating our system. It turns out that these apps still failed. We then manually analyzed these apps which crashed and found that the failure is mostly due to the integrity check used by these apps, such as checking the signing key. We believe this type of integrity check is used by apps to prevent them from being pirated. And some apps crashed because of the network error, download error and others. Nevertheless, there are still 395 apps left with our system for evaluation.

Table 4 shows a summary of our test results. The first column shows different operations that our system could confine. The second and third column show the number of operations detected by our system in total and in ad libraries respectively. `AdCapsule` can distinguish privacy-concerned operations from the host app and ad libraries successfully, and apply corresponding policies. For instance, among 395 apps, ad libraries in 311 apps read the list of installed apps or probe host app's permissions through the `PackageManager`. Ad libraries in 9 apps retrieve user's location through the `LocationManager`.

Hypothetical Ad Library To further evaluate the effectiveness, we developed a hypothetical ad library which aggressively abuses the host app's permissions for malicious purposes, and steals host app's files. We use this hypotheti-

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

his article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2814999, IEEE Transactions on Dependable and Secure Computing

TABLE 5
Evaluation Results on the Hypothetical Ad Library

Operations	Ad Library	Regulated
Read IMEI	Y	Y
Read SMS	Y	Y
Send SMS	Y	Y
Access Location	Y	Y
Access Clipboard	Y	Y
Access Contacts	Y	Y
Access App List	Y	Y

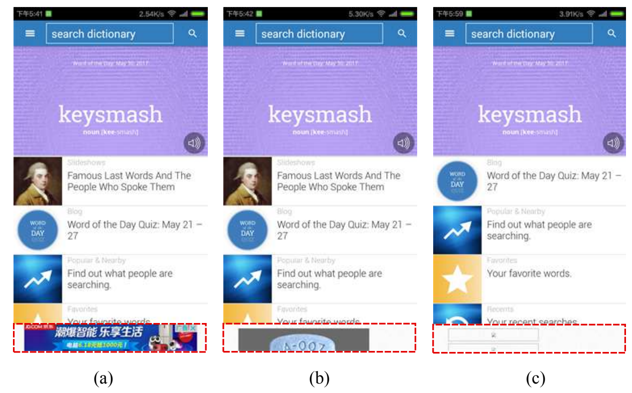
cal ad library to evaluate the effectiveness of our system in the worst case. We integrate this ad library into an app we developed. In addition, to evaluate the context-aware policy enforcement for ad libraries, the app also performs privacy-concerned operations and invokes sensitive APIs. We want to evaluate whether `AdCapsule` could distinguish the context of the API invocation.

Table 5 shows the evaluation results. `AdCapsule` can capture the privacy-concern APIs from the hypothetical ad library and apply corresponding security policies. In addition, our system can successfully distinguish the context of API invocation and only apply security policies for the ad libraries, not the app code itself.

Malicious Ad Contents As presented by previous research, malicious ad contents could infer sensitive information by accessing local files or inferring the existence of local files using JavaScript [8]. According to the paper, GoodRx is a popular app to help patients compare prescription drug prices and find coupons at more than 60, 000 US pharmacies. But it caches pictures of user's drugs, which could be used to infer the user's health information.

To evaluate the effectiveness of `AdCapsule` to confine malicious ad contents, we use the Dictionary app which contains AdMob library, the most popular ad library provided by Google. The AdMob library downloads ad contents in plain text from the remote server, and it is easy for us to intercept the network packet and replace it with our own malicious ad contents. The malicious ad content reads files of the GoodRx app, the same attack as shown in previous research [8].

Specifically, the AdMob library in the Dictionary app fetched a normal ad content and displayed successfully (Figure 7 (a)). Then we intercepted AdMob's network traffic and injected a malicious ad (in the format of HTML), which is similar to the attack proposed in the paper [8]. In particular, the injected malicious ad detects the existence of local images of the GoodRx app using the JavaScript described in the paper. Since the GoodRx app caches pictures of searched drugs, the ad contents can successfully read the local image files (in Figure 7 (b)). Note that for demo purpose, the image file of cached drug is displayed in Figure 7 (b). In real attack, the files is read and uploaded to a remote server in the background. After integrating our system, the injected malicious ad contents cannot load the image files since all the local files operated by the ad contents are inside a file sandbox with its own namespace (Figure 7 (c)). This evaluation demonstrated that, because of our system, malicious ad contents cannot directly operate on the local files and all the attacks proposed in the paper [8] based on



(a) (b) (c)

Fig. 7. Evaluation results of the confinement of ad contents

local file access are no longer effective. Meanwhile, non-ad-contents in host app remain unaffected.

6.2 Compatibility

During our evaluation, for each repackaged app with `AdCapsule`, we run the app for five minutes and use the Monkey [47] to generate different inputs and system events such as keyboard events. We also parsed the app's manifest file and start services and activities if they are exported. We did not observe any crash of these apps. To further evaluate the compatibility, we uploaded repackaged apps to testin [48], a free testing platform of compatibility on different Android devices with different versions. The results showed that all these apps passed the compatibility test without any problem caused by our system.

6.3 Performance Overhead

After evaluating the effectiveness and compatibility, we then evaluate the performance overhead introduced by our system. Our evaluation is on a XiaoMi 2S phone with Android version 4.4 and HongMi Note 3 phone with Android version 6.0. For the privacy-concerned APIs, our system checks the stack trace to know the context and distinguish the ad library from the host app. This is a time-consuming task. In addition, our system queries and enforces security policies. These operations may increase the time used to finish the invocation of framework APIs. We picked nine typical privacy-concerned operations and tested the time used to perform these operations with and without our system. For each operation, we tested 100 times and calculated the average time used to invoke these APIs. The results are shown in Table 6. The time increased for most operations is below 1 ms, which is beyond the range that users can actually perceive [49].

7 Discussion

In this section, we will discuss possible limitations and potential improvements of our system.

First, our system mainly adopts binder hooking, in-VM API hooking and GOT hooking to reliably hook the privacy-related APIs. However, these techniques can be bypassed through native libraries. For instance, ad libraries can load a native library to call the managers directly or remove all

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2814999, IEEE Transactions on Dependable and Secure Computing

TABLE 6
Performance Overhead Introduced by AdCapsule

Operation	Android4.4 (ms)		Android6.0 (ms)	
	Original	Overhead	Original	Overhead
Get Clickboard	3.04	0.78	1.16	0.13
Show Notification	6.29	1.09	2.38	0.31
Get IMEI	1.72	0.42	1.42	0.01
Open Camera	496.76	0.77	125.80	0.57
Send SMS	10.18	0.64	6.35	0.28
Add Contact	333.78	3.67	205.48	2.56
Read SMS	18.65	0.78	4.89	0.17
Get Location	5.17	0.78	1.61	0.07
Open File	0.79	0.59	0.32	0.14

the hooks we have placed. However, we argue that we can disable the native library loading in ad libraries, since the native code is always used for the performance. We do not think there are legitimate reasons to use native library inside the ad library. Indeed, this assumption is true in practice. According to PEDAL [22], only 1.06% ad libraries are using native libraries. In terms of confining native libraries, we think SFI-based isolation [45, 50–53] could be leveraged. For instance, AppCage [45] leverages both the dynamic binary rewriting and static compiler-based binary instrumentation to confine native libraries in Android apps. Our system could take the similar approach to dynamically rewrite the native code in ad libraries. Both NativeGuard [50] and Boxify [53] leverage the process boundary to confine native code, but with different approaches. Specifically, NativeGuard disables the sensitive operations (or permissions) with predefined policies, while Boxify uses the new security mechanism, i.e., the isolated process, for this purpose. We could build a sandbox for native code of ad libraries using the separated processes as these two systems. However, how to implement an efficient communication between the bytecode and native code needs to be further studied. That is because every JNI call becomes a remote procedure call (RPC) across the process boundary, which introduces extra overhead. NaClDroid [51] adopts the native client compiler to build the confined native libraries. However, it requires the availability of the source code of the native libraries, which is not a reasonable assumption in our scenario. There are other ongoing works conducted by researchers. We believe that the research progress in the field of native code confinement of Android apps could be borrowed by our system in the future.

Second, we use Java stack trace to identify the context, e.g., the API is invoked from the app code or from the ad library. However, this mechanism does not work in WebView when ad contents retrieve the location using JavaScript. That is because most operations are implemented in the native code of the WebView component. As discussed in Section 4.2, our system hooked the `WebSettings` in the WebView and can only disable the access of the location when the WebView is first loaded. That is to say, our system can not return a fake location at runtime when the ad content is accessing the location. This is a limitation of our system, and we leave it as a future work. In addition, leveraging Java call stack introduces performance overhead to our system. During our evaluation, we found that nearly 80% of the performance overhead is caused by the operation to obtain the Java call stack at runtime. It would be better if we could find an alternative solution to get the context of API invocation with less performance overhead.

Third, our system does not need to change the framework. Hence, it is easier to be practically deployed than the solutions with framework modification. There are several scenarios that our system could be deployed. First, with the engagement of app developers, our system could be deployed during the development process of the app. App developers include our system and specify security policies when integrating ad libraries. This is the most natural scenario to use our system. Second, our system could be used in the enterprise app market in the scenario of BYOD. In this case, if the app's code has been obfuscated, previous work [54–56] could be leveraged to detect ad libraries. Then the app could be repackaged to include our system, as we did in the evaluation, without the engagement of app developers. This ensures that ad libraries running on devices in the enterprise environment cannot leak private information, such as location and SMS messages. Nevertheless, how to facilitate the integration of our system is still an open question.

Fourth, we use the package name to identify the ad libraries. Different from previous works, how to identify the ad library is not a concern in our system. That is because our system is for app developers and we are working towards the original ad libraries (i.e., the jar file), instead of the obfuscated classes that have been mixed with app's code. It is much easier to identify the source of ad library using the imported jar files in the app. Thanks to the publicly available data sources of third-party libraries [54], we could identify the libraries more precisely in our work.

8 Related Work

Privacy Concerns of Mobile Advertisements Mobile advertisements have raised security and privacy concerns in past years. AdRisk [7] is the first system that exposes the potential security and privacy issues inside the in-app ad libraries. The authors studied 100,000 apps from the official Android Market in March-May, 2011. From these apps, they identified 100 representative in-app ad libraries and further developed a system called AdRisk to systematically identify potential risks. The results showed that most existing ad libraries aggressively collected private information, including location, call logs, phone numbers, etc. Moreover, some libraries make use of an unsafe mechanism to directly fetch and run code from the Internet, which immediately leads to serious security risks. Another work [57] collected a sample of 114,000 apps and then extracted and classified the embedded ad libraries. They found that the use of permissions of ad libraries has increased, and more libraries are using permissions that pose particular risks to users' privacy. Recently, researchers systematically studied the reach of mobile advertisements, with the help of machine learning and data mining technique [6]. They found that, ad networks can obtain users' private information from the data collect from the device, such as interests, demographics, medical conditions, etc.

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2814999, IEEE Transactions on Dependable and Secure Computing

Besides ad libraries, ad contents also cause serious privacy issues [8]. Son *et al.* presented the risks posed by malicious ad contents. Based on the experimental results on several popular ad libraries, they found that aggressive ad contents can infer sensitive information about users by accessing files on mobile devices and track users. At the same time, researchers systematically studied the redirections in mobile ad contents [58]. When users tap on an ad, he or she may be redirected to another link, which could be a fraudulent website, or a site containing malicious apps. Our work is motivated by risks raised by ad libraries and ad contents, and provides a practical solution to confine advertisements.

Solutions to Mitigate Risks of Mobile Advertisements To mitigate threats of ad libraries, several solutions have been proposed by researchers. AdDroid [18] is one of them. It introduced additional ad-related APIs which are supported by the ad service in the Android framework, so that ad contents can be shown without requesting privacy sensitive permissions. AdSplit [19] modifies the Android framework to separate the host app and the ad libraries by running them in different processes. Zhang *et. al.* proposed AFrame [20], which provides permission, display and input isolation to separate ad libraries from the host app. However, all these systems require to change the Android framework, which obstacles the practical deployment.

Data-Sluice [59] is a framework to control the incoming and outgoing data for each Android app, including ad libraries and ad contents. It can remove undesired advertisement banners or annoying popups, however, this may reduce app developers' revenue. ADSandbox [10] decides whether a site is malicious or not by detecting attacks through JavaScript. And AdSentry [12] uses a shadow JavaScript engine to sandbox untrusted ads. There are also origin-based works [60–62] confining ad contents and the available JavaScript interfaces. They all can eliminate the threat exposed by ad contents, but left the threat of ad libraries.

Recently, researchers proposed PEDAL [22]. It leverages bytecode rewriting technique to confine in-app ad libraries. Specifically, it first identifies ad libraries packaged within an app and then rewrites them. Privacy-related code in ad libraries are redirected to the reference monitor. As a result, ad libraries work under user's control and cannot behave aggressively. PEDAL is limited by the code obfuscation, Java reflection and especially the dynamic code execution at runtime. Our system confines advertisements at runtime, and is immune to dynamic code execution of ad libraries.

CASE [63] is a more general system aiming at module-level security policies in apps. By interposing a set of selected VM internal functions and system call interfaces, CASE monitors inter-module crossings encountered during an app execution. It utilizes Java call stack trace to identify the caller, as we did in our system. However, it does not work towards WebView, the important component of ad libraries to render rich-media advertisements.

Smartphone Apps Security and Privacy Besides ad libraries, apps also cause serious privacy and security concerns. Researchers have proposed various techniques to understand or assess the risks of smartphone apps. Android Malware Genome Project [64, 65] collected and characterized Android malware. It collected more than 1,200 malware samples that cover the majority of existing Android malware families at that time and characterized them from various aspects. TaintDroid [66] leverages dynamic information flow technique to track the data flow of private data inside the app. This system found several cases that smartphone apps are leaking private information. PiOS [67], on the other side, uses static analysis technique to analyze the apps to find the potential paths between the code of retrieving private data to the ones that send data out.

To confine smartphone apps, some systems extend the Android framework to provide fine-grained control of apps at runtime [36, 37, 68–71]. For example, AppFence [36] and TISSA [37] can return mock results of the sensitive resources such as the location. User-driven access control is a promising solution to provide in-context and non-disruptive permission granting [72]. From another perspective, researchers have proposed systems to confine apps in user space with bytecode rewriting [24–27, 38] or native library interposing [44]. Similar system also exists on other platforms [73]. However, these systems are working towards the whole app, not the ad libraries and ad contents which have different goals and challenges.

9 Conclusion

In this work, we propose a user-level solution to confine advertisements, including ad libraries and ad contents. Our system could be readily deployed since it does not need to change Android framework, nor needs the root privilege. It leverages two sandboxes to confine privacy-concerned operations and file operations of advertisements. We have implemented a prototype of our system and the evaluation results demonstrate the effectiveness and compatibility of our system, and the performance overhead is low.

References

[1] "Number of Android applications." https://www.appbrain.com/stats/number-of-android-apps.

[2] "Free vs. paid Android apps." https://www.appbrain.com/stats/free-and-paid-android-applications.

[3] "Security Alert: New Stealthy Android Spyware – Plankton – Found in Official Android Market." https://www.csc.ncsu.edu/faculty/jiang/Plankton/, 2011.

[4] "PontiFlex Ad Library - Remote JavaScript Command Execution." https://labs.mwrinfosecurity.com/blog/pontiflex-ad-library-remote-javascript-command-execution/, 2013.

[5] "Setting the Record Straight on Moplus SDK and the Wormhole Vulnerability." http://blog.trendmicro.com/trendlabs-security-intelligence/setting-the-record-straight-on-moplus-sdk-and-the-wormhole-vulnerability/, 2015.

[6] S. Demetriou, W. Merrill, W. Yang, A. Zhang, and C. A. Gunter, "Free for All! Assessing User Data Exposure to Advertising Libraries on Android," in *Proceedings of the 23rd Annual Symposium on Network and Distributed System Security*, NDSS, 2016.

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

his article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2814999, IEEE Transactions on Dependable and Secure Computing

[7] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe Exposure Analysis of Mobile In-App Advertisements," in *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ACM WiSec, 2012.

[8] S. Son, D. Kim, and V. Shmatikov, "What Mobile Ads Know about Mobile Users," in *Proceedings of the 23rd Annual Symposium on Network and Distributed System Security*, NDSS, 2016.

[9] N. Daswani and M. Stoppelman, "The anatomy of clickbot. a," in *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pp. 11–11, USENIX Association, 2007.

[10] A. Dewald, T. Holz, and F. C. Freiling, "Adsandbox: Sandboxing javascript to fight malicious websites," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 1859–1864, ACM, 2010.

[11] A. Zarras, A. Kapravelos, G. Stringhini, T. Holz, C. Kruegel, and G. Vigna, "The dark alleys of madison avenue: Understanding malicious advertisements," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, pp. 373–380, ACM, 2014.

[12] X. Dong, M. Tran, Z. Liang, and X. Jiang, "Adsentry: comprehensive and flexible confinement of javascript-based advertisements," in *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 297–306, ACM, 2011.

[13] T. Book and D. S. Wallach, "An empirical study of mobile ad targeting," *CoRR*, vol. abs/1502.06577, 2015.

[14] S. Han, J. Jung, and D. Wetherall, "A study of third-party tracking by mobile apps in the wild."

[15] "How to inject JavaScript into an Android web view for a more dynamic UX." http://www.techrepublic.com/article/pro-tip-inject-javascript-into-an-android-web-view-for-a-more-dynamic-ux/, 2014.

[16] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications.," in *NDSS*, vol. 14, pp. 23–26, 2014.

[17] J. Seo, D. Kim, D. Cho, T. Kimy, and I. Shinz, "Flexdroid: Enforcing in-app privilege separation in android," in *Proceedings of the 23rd Annual Symposium on Network and Distributed System Security*, 2016.

[18] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "AdDroid: Privilege Separation for Applications and Advertisers in Android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ACM ASIACCS, 2012.

[19] S. Shekhar, M. Dietz, and D. S. Wallach, "AdSplit: Separating Smartphone Advertising from Applications," in *Presented as part of the 21st USENIX Security Symposium*, USENIX Security, 2012.

[20] X. Zhang, A. Ahlawat, and W. Du, "AFrame: Isolating Advertisements from Mobile Applications in Android," in *Proceedings of the 29th Annual Computer Security Applications Conference*, ACM ACSAC, 2013.

[21] P. Mutchler, Y. Safaei, A. Doupe, and J. Mitchell, "Target Fragmentation in Android Apps," in *IEEE Security and Privacy Workshops*, pp. 204–213, 2016.

[22] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient Privilege De-escalation for Ad Libraries in Mobile Apps," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, ACM MobiSys, 2015.

[23] "Android GOT Hook." https://shunix.com/android-got-hook/, 2016.

[24] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "AppGuard: Enforcing User Requirements on Android Apps," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2013.

[25] B. Davis and H. Chen, "RetroSkeleton: Retrofitting Android Apps," in *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, 2013.

[26] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications," *Mobile Security Technologies*, 2012.

[27] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*.

[28] "AppBrain.com: Monetize, advertise and analyze Android apps." https://www.appbrain.com/.

[29] "WebView." https://developer.android.com/reference/android/webkit/WebView.html, 2016.

[30] "JavascriptInterface." https://developer.android.com/reference/android/webkit/JavascriptInterface.html, 2016.

[31] "WebView addJavascriptInterface Remote Code Execution." https://labs.mwrinfosecurity.com/advisories/webview-addjavascriptinterface-remote-code-execution/, 2013.

[32] "Chinese Taomike Monetization Library Steals SMS Messages." http://researchcenter.paloaltonetworks.com/2015/10/chinese-taomike-monetization-library-steals-sms-messages/, 2015.

[33] "AppLovin Ad Library SDK: Remote Command Execution via Update Mechanism." https://labs.mwrinfosecurity.com/blog/applovin-ad-library-sdk-remote-command-execution-via-update-mechanism/, 2013.

[34] "Millenial Media Ad Library." https://labs.mwrinfosecurity.com/blog/millenial-media-ad-library/, 2013.

[35] "Android Ad networks." https://www.appbrain.com/stats/libraries/ad/, 2016.

[36] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These Aren't the Droids You're Looking For: Retroffiting Android to Protect Data from Imperious Applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ACM CCS, 2011.

[37] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming Information-Stealing Smartphone Applications (on Android)," in *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST, 2011.

[38] N. Reddy, J. Jeon, J. Vaughan, T. Millstein, and J. Foster, "Application-centric Security Policies on Unmodified

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Android," *UCLA Computer Science Department, Tech. Rep*, 2011.

[39] "Dynamic Proxy Classes." https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html, 2016.

[40] "AndFix is a library that offer hot-fix for Android App." https://github.com/alibaba/AndFix, 2017.

[41] "Yet Another Hook Framework for ART." https://github.com/rk700/YAHFA, 2017.

[42] "Android API to Permission Mapping Extractor." https://github.com/fgwei/android-a2p, 2017.

[43] 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*, pp. 217–228, ACM, 2012.

[44] R. Xu, H. Saidi, and R. Anderson, "Aurasium: Practical Policy Enforcement for Android Applications," in *Proceedings of the 21th USENIX Security Symposium*, USENIX Security, 2012.

[45] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang, "Hybrid User-level Sandboxing of Third-party Android Apps ," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ACM ASIACCS, 2015.

[46] "JavascriptInterface." https://ibotpeaches.github.io/Apktool/, 2016.

[47] "UI/Application Exerciser Monkey." https://developer.android.com/studio/test/monkey.html, 2016.

[48] "Enterprise Mobile App Testing Platform - Testin." http://www.testin.io/, 2016.

[49] B. Shneiderman, *Designing the user interface: strategies for effective human-computer interaction*. Pearson Education India, 2010.

[50] M. Sun and G. Tan, "Nativeguard: Protecting android applications from third-party native libraries," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pp. 165–176, ACM, 2014.

[51] E. Athanasopoulos, V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "Nacldroid: Native code isolation for android applications," in *European Symposium on Research in Computer Security*, pp. 422–439, Springer, 2016.

[52] B. Ford and R. Cox, "Vx32: Lightweight User-level Sandboxing on the x86," in *Proceedings of the USENIX 2008 Annual Technical Conference*, USENIX ATC, 2008.

[53] M. Backes, S. Bugiel, C. Hammer, O. Schranz, and P. von Styp-Rekowsky, "Boxify: Full-fledged App Sandboxing for Stock Android," in *Proceedings of the 24th USENIX Security Symposium*, USENIX Security, 2015.

[54] Z. Ma, H. Wang, Y. Guo, and X. Chen, "Libradar: Fast and accurate detection of third-party libraries in android apps," in *Proceedings of the 38th International Conference on Software Engineering Companion*, pp. 653–656, ACM, 2016.

[55] H. Wang, Y. Guo, Z. Ma, and X. Chen, "Wukong: a scalable and accurate two-phase approach to android app clone detection," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 71–82, ACM, 2015.

[56] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 356–367, ACM, 2016.

[57] T. Book, A. Pridgen, and D. S. Wallach, "Longitudinal Analysis of Android Ad Library Permissions," *arXiv preprint arXiv:1303.0857*, 2013.

[58] V. Rastogi, R. Shao, Y. Chen, X. Pan, S. Zou, and R. Riley, "Are These Ads Safe: Detecting Hidden Attacks through the Mobile App-Web Interfaces," in *Proceedings of the 23rd Annual Symposium on Network and Distributed System Security*, NDSS, 2016.

[59] A. Saracino, F. Martinelli, G. Alboreto, and G. Dini, "Data-sluice: Fine-grained traffic control for android application," in *Computers and Communication (ISCC), 2016 IEEE Symposium on*, pp. 702–709, IEEE, 2016.

[60] M. Georgiev, S. Jana, and V. Shmatikov, "Breaking and fixing origin-based access control in hybrid web/-mobile application frameworks," in *NDSS symposium*, vol. 2014, p. 1, NIH Public Access, 2014.

[61] G. S. Tuncay, S. Demetriou, and C. A. Gunter, "Draco: A system for uniform and fine-grained access control for web code on android," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 104–115, ACM, 2016.

[62] D. Davidson, Y. Chen, F. George, L. Lu, and S. Jha, "Secure integration of web content and applications on commodity mobile operating systems," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 652–665, ACM, 2017.

[63] S. Zhu, L. Lu, and K. Singh, "CASE: Comprehensive Application Security Enforcement on COTS Mobile Devices," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ACM MobiSys, 2016.

[64] "Android Malware Genome Project." http://www.malgenomeproject.org/, 2012.

[65] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Security*, IEEE S&P, 2012.

[66] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2010.

[67] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications," in *Proceedings of the 18th Annual Symposium on Network and Distributed System Security*, NDSS, 2011.

[68] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: Trading Privacy for Application Functionality on Smartphones," in *Proceedings of the 12th workshop on mobile computing systems and applications*, pp. 49–54, ACM, 2011.

[69] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "myTunes: Semantically Linked and User-centric Fine-grained Privacy Control on Android," *Technical Report TUD-CS-2012-0226, Center for Advanced Security Research Darm-*

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

his article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2018.2814999, IEEE Transactions on Dependable and Secure Computing

stadt (CASED), 2012.

[70] S. Bugiel, S. Heuser, and A.-R. Sadeghi, "Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies," in *Proceedings of the 22nd Usenix Security Symposium*, USENIX Security, 2013.

[71] 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*.

[72] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven Access Control: Rethinking Permission Granting in Modern Operating Systems," in *Proceedings of the 33rd IEEE Symposium on Security and Security*, IEEE S&P, 2012.

[73] B. Livshits and J. Jung, "Automatic Mediation of Privacy-Sensitive Resource Access in Smartphone Applications," in *Proceedings of the 22nd Usenix Security Symposium*, USENIX Security, 2013.

Xiaonan Zhu received the B.S. degree in Computer Science from Xidian University, Xi'an, China, in 2015. He is currently pursuing his M.S. degree in the School of Cyber Engineering at Xidian University, Xi'an, China. His research focuses on mobile security.

Jinku Li received the B.S., M.S., and Ph.D. degrees in computer science from Xi'an Jiaotong University, Xi'an, China, in 1998, 2001, and 2005, respectively. From March 2009 to February 2011, he was a postdoctoral fellow in the Department of Computer Science at North Carolina State University, Raleigh, NC. He is currently an associate professor in the School of Cyber Engineering at Xidian University, Xi'an, China. His research focuses on system and mobile security.

Yajin Zhou earned his Ph.D. in Computer Science from North Carolina State University. His research mainly focuses on smartphone and system security, i.e., identifying real-world threats and building practical solutions. He is currently a ZJU 100 Young Professor in the Cyber Security Research Center and School of Computer Science at Zhejiang University, China.

Jianfeng Ma received the Ph.D. degree in computer software and communications engineering from Xidian University, Xi'an, China, in 1995. From 1999 to 2001, he was with Nanyang Technological University of Singapore as a research fellow. He is currently a professor in the School of Cyber Engineering at Xidian University, Xi'an, China. His current research interests focus on information and network security.

1545-5971 (c) 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.