PrivateDroid: Private Browsing Mode for Android

Su Mon Kywe*
Singapore Management University
Singapore
Email: monkywe.su.2011@smu.edu.sg

Christopher Landis
Carnegie Mellon University
Pittsburgh, USA
Email: clandis@alumni.cmu.edu

Yutong Pei
Carnegie Mellon University
Pittsburgh, USA
Email: ypei@andrew.cmu.edu

Justin Satterfield
Carnegie Mellon University
Pittsburgh, USA
Email: jsatterf@andrew.cmu.edu

Yuan Tian
Carnegie Mellon University
Pittsburgh, USA
Email: yt@cmu.edu

Patrick Tague
Carnegie Mellon University
Pittsburgh, USA
Email: tague@cmu.edu

***Abstract*—Private browsing mode is a privacy feature adopted by many modern computer browsers. With the increased use of mobile devices and escalating privacy concerns for mobile users, browser applications on mobile devices have also started incorporating private browsing mode. Even so, the use of private browsing mode is limited to the browser applications and cannot be applied directly on other third-party mobile applications. In this paper, we propose PrivateDroid, which provides a private browsing mode for third-party applications on the Android platform. First, we discuss three possible approaches of implementing mobile private browsing mode: code instrumentation, an extra sandbox, and a Linux container approach. Then, we implement PrivateDroid, which creates a new sandbox for every application in private mode and destroys the sandbox once the application is closed. After that, we evaluate usability, efficiency and security of the system with 25 popular Android applications. Our design considerations, implementation details, evaluation results, and challenges lay a foundation of private browsing mode on mobile platforms.**

***Index Terms*—Mobile Privacy, Private Browsing Mode**

I. Introduction

Private browsing mode of modern browsers plays a great role in protecting user privacy. People use it to search for something private, such as medical conditions, new job opportunities, to browse secretly for gifts, or to log into personal accounts on public computers so as to not leave a history of their online actions. Private browsing mode ensures that personal information, such as browsing histories, cookies and cache information, are cleared once the browsing session ends. With the increased use of mobile devices, some mobile browsers have also incorporated the private browsing mode. However, it is not trivial to guarantee similar level of privacy due to the architectural differences between Personal Computers (PCs) and mobile devices. To make matters worse, Android allows any third-party applications with appropriate permission to access the web and leave history or cache information behind. Thus, private browsing mode in mobile browsers is not sufficient to ensure privacy of mobile users.

In this paper, we introduce PrivateDroid, which extends the private browsing mode concept to third-party Android applications. PrivateDroid is a modified version of Android framework, in which users can specify a mobile application to launch in private mode. PrivateDroid leverages the sandboxing mechanism of the Android framework. When an application is launched in private mode, PrivateDroid creates a new sandbox for that application. Once it is closed, PrivateDroid removes the sandbox together with its private data storage directory. Thus, all locally stored state information of users is destroyed when users exit the private mode application. Moreover, PrivateDroid spoofs the International Mobile Equipment Identity (IMEI) of mobile devices. Hence, whenever an application is launched in the private mode, it would 'think' that it is the first time running on that device.

The contributions of our paper are as follows.

1) We are the first to investigate the problem of the private browsing mode concept in the context of mobile applications. We also propose three different designs of implementing private mode for mobile applications. We discuss their strengths and weaknesses in terms of security, efficiency, usability, and ease of implementation.
2) From the proposed designs, we choose the extra sandbox approach, which is relatively lightweight and provides a balance between security and usability. We implement it as PrivateDroid and incorporate various usability features for private mode, such as indicators for active sessions of private mode.
3) We evaluate PrivateDroid with 25 popular Android applications. The evaluation results and our challenges reveal insights into different ways of improving private mode for mobile applications.

Our paper is organized as follows. Section II provides background information on Android architecture, Android's sandbox, and our threat model. In Section III, we propose and compare three different approaches for implementing private mode for mobile applications. Section IV discusses the implementation details of PrivateDroid and evaluation results are given in Section V. Section VI summarizes the related work and we conclude our paper in Section VII.

*This work was done, when the author was in Carnegie Mellon University

II. Background

A. Android Platform

1) Android OS Architecture: The Android architecture consists of three layers, namely the application layer, middleware layer, and kernel layer. Android's application layer includes default system applications, such as the Contacts List, and other third-party applications installed by users. The middleware layer provides services to the application layer via the application framework or Application Programming Interfaces (APIs). The middleware layer also includes the Dalvik virtual machine in which applications run, along with core Java libraries. As the foundation of this architecture, the kernel layer is responsible for managing drivers, network sockets, processes, and file systems.

2) Android Sandbox: The main purpose of Android's sandbox is to isolate applications. When an application is installed, Android assigns a unique user ID and private data directory in which the application can store its data. After that, each app, with its unique user ID, runs in its own Dalvik virtual machine. In this way, the sandbox ensures that each application manages its own processes and files and cannot interfere with other applications' processes and files. If an application wants to communicate with the Android system or other third-party applications, it is required to request appropriate permissions. During application uninstallation, the sandbox destroys the application's user ID and its private data directory.

B. Threat Model

As the private browsing mode is a relatively new concept of browser security, even some popular PC browsers, such as Chrome and Firefox, do not have commonly agreeing threat models. However, according to G. Aggarwal et al. [1], there are two types of attackers that should be prevented by private browsing mode: local attackers and web attackers.

1) Local Attackers: Local attackers are people who have physical access to the device or malicious software that is installed on the device after the browsing session. An example scenario is where a user plans to secretly buy a surprised gift for a family member. The family member becomes a local attacker in this case. Local attackers are passive; they will not install tracking applications or key loggers to detect application usage history. Local attackers can be generally prevented by removing all the cookies, session caches, and browsing histories added by users in private browsing mode.

Nonetheless, clearing such information is much more difficult in mobile applications than in PC browsers. Consider a scenario where a person borrows a mobile phone from her friend, launches the Instagram application in private mode, takes a photo and uploads it to her Instagram network. Instagram application normally stores the photo in its local storage before uploading it to the social network. To prevent local attackers, when the user closes the private session of Instagram application, the photo should also be removed from the device's storage along with her cookie information.

Note that even the popular PC browsers do not completely prevent local attackers to support usability. For instance, they do not remove the files downloaded during the private session. In this paper, we also decide to follow a relatively loose security model. PrivateDroid is designed to clear private data from both internal storage, such as application private data directory, and external storage, such as SD card. However, it will not remove data added via Inter-Process Communication (IPC). For example, if a mobile application in private mode modifies a contact of user's address book, PrivateDroid will not revert such information.

2) Web Attackers: According to G. Aggarwal et al. [1], there are three goals of web attackers: (1) to link a user visiting in private mode to the same user visiting in public mode, (2) to link a user visiting in private mode to the same user visiting in private mode and (3) to determine if the user is currently using the application in private mode. Some PC browsers prevent web attackers by separating the availability of public and private cookies. On the other hand, some other popular PC browsers, such as Safari, do not even consider web attackers in their threat model.

In the context of mobile devices, applications have access to a lot of user identifiers, such as the IMEIs of the devices and personal email accounts, allowing web attackers to easily link a user across different sessions. Thus, to completely prevent web attackers, all available user identifiers from the mobile devices should be spoofed during the private mode session. Our current implementation of PrivateDroid spoofs the device IMEI so that they look legitimate to web attackers. Note that Internet Protocol (IP) address tracking and browser fingerprinting are not prevented by private mode.

III. Design Considerations

In this section, we present three possible approaches of implementing private mode for mobile applications. The first approach is code instrumentation, in which the source code of third-party applications are modified to support private mode. The second approach uses a Linux Container, in which Operating System (OS)-level virtualization is used to separate public and private modes. The third approach is the extra sandbox approach, in which applications in different modes are isolated in terms of their processes and file systems. We will discuss the security, efficiency, usability, and ease of implementation of all approaches.

A. Approach I: Code Instrumentation

The first approach is to apply code instrumentation to third-party mobile applications. Code instrumentation is the insertion or modification of specific code to the original source files. It can be used to analyze and modify the application source code that accesses mobile user identifiers or the code that creates permanent states on mobile devices. For instance, every `TelephonyManager.getDeviceId()` method in the application source code can be replaced with a fake IMEI value. Similarly, the source code that leaves permanent data on the device can be modified or disabled. Thus, this approach can be used to prevent both local attackers and web attackers of private mode users.

1) Security: Source code instrumentation is normally used for data logging, debugging, or software performance analysis and not for security. Moreover, static analysis of source codes is known to have imperfect coverage and may fail to modify the applications that use Java reflection and code obfuscation. Listing 1 shows simple obfuscated code with Java reflection for the `TelephonyManager.getDeviceId()` method. Unfortunately, it is not uncommon for application developers to use automated tools with more complex obfuscation methods. Thus, we rate the security of the code instrumentation approach as *bad*.

Listing 1. Invoking `TelephonyManager.getDeviceId()` via Java Reflection and Code Obfuscation

```
char[] s = {'a','b','c','d','e','f','g','h',
    'i','j', 'k','l','m','n','o','p','q','r',
    's','t','u','v','w','x','y','z'};
char[] S = {'A','B','C','D','E','F','G','H',
    'I','J','K','L','M','N','O','P','Q','R',
    'S','T','U','V','W','X','Y','Z'};
String className = "" + S[19] + s[4] + s[11]
    + s[4] + s[15] + s[7] + s[14] + s[13] +
    s[24] + S[12] + s[0] + s[13] + s[0] +
    s[6] + s[4] + s[17];
String methodName = "" + s[6] + s[4] + s[19]
    + S[3] + s[4] + s[21] + s[8] + s[2] +
    s[4] + S[8] + s[3];
Class iClass = Class.forName(className);
Method iMethod =
    iClass.getDeclaredMethods(methodName,
    null);
String deviceIMEI = iMethod.invoke();
```

2) Efficiency: Source code instrumentation involves four time-consuming steps: (1) reverse engineering the binary code of applications, (2) searching for the source code to be modified, (3) modifying the source code, and (4) repackaging those applications. All of these steps would be performed on mobile devices during application installation; therefore, this approach will not only increase the source code size but also considerably lengthen the installation duration. However, this is just a one-time process and little difference can be observed by users after the installation process has completed. We rate the efficiency of this approach as *moderate*.

3) Usability: Repackaging third-party applications will break some functionality of the applications. For instance, Oauth [2], which allows users to use their Facebook or Google accounts to authenticate themselves in other applications, requires valid application signatures. However, repackaging invalidates the signatures and makes these applications unusable. Nonetheless, such cases are rare and we rate the usability level of code instrumentation as *moderate*.

4) Ease of Implementation: Implementing source code instrumentation involves modifying the source code of Android framework to repackage third-party applications. However, compared to modifying the kernel layer, this approach is easier to implement. As such, we rate the ease of implementation as *moderate* for the source code instrumentation approach.

B. Approach II: Linux Container

The second approach is to maintain two parallel Android OSs on one single device: one for public mode and another for private mode. When an application is launched in private mode, it will be installed on the private container and run in an isolated environment. After the private session is ended, the application will be uninstalled from the private container, removing all the states that are created during the process. The private container should also be able to spoof users' identities to enforce private mode.

1) Security: In this approach, all three layers of the Android architecture, including application, middleware, and kernel layers, are separated into two distinct and parallel containers. Only then, all the IPCs, file systems, and run-time memories will be isolated by container. This approach is often used in enterprises to isolate cooperate applications from third-party applications [3]. Therefore, we rate the security level of Linux container approach as *good*.

2) Efficiency: A very robust security policy is normally accompanied by high complexity. The same goes for this Linux container approach. If we choose to run both containers simultaneously and continuously, it will negatively affect the device resources, including storage, memory, and battery consumption. On the other hand, if the private mode container is launched only when the user switches the private mode on, the user will need to wait for the private version of the OS to boot. Launching time can also be exacerbated by installation time of private applications in the private container. Thus, we rate the efficiency as *bad* for Linux container approach.

3) Usability: Assuming that Linux container is correctly implemented, it will not interfere with the functionalities of third-party applications. Therefore, we rate the usability level of Linux container as *good*.

4) Ease of Implementation: This approach requires modifying Android's kernel layer source code to separate its file system as well as isolating the run-time memories of the two systems. Hence, we evaluate it as *bad* for ease of implementation.

C. Approach III: Extra Sandbox

Our third approach is to create a new and isolated sandbox for every application in private browsing mode. This approach utilizes the default sandboxing mechanism of Android that isolates applications. By creating extra sandboxes for applications in private mode, we can limit the private applications to run only in their own processes with assigned user IDs and to have access only to their own internal data directories. Thus, all the processes and data from private mode applications are isolated from the applications in public mode.

1) Security: The sandbox mechanism only isolates certain aspects of applications, such as private data storage and code execution. Hence, creating a new sandbox does not prevent private mode applications from communicating with other public applications or establishing persistent states by writing to the content providers, such as the contact list of mobile phones. Therefore, we rate the security level as *moderate*.

2) Efficiency: Every time an application is launched in private mode, it is reinstalled in a new sandbox with a new user ID and data directory. Similarly, when an application is closed from private mode, it is uninstalled and its user ID and data directory are destroyed. Although the time delay in launching is hardly noticeable for applications with a relatively small source code size (less than about 5 MB), the installation time delay is observable for applications with a larger source code size. Therefore, we rate the usability of an extra sandbox approach as *moderate*.

3) Usability: Creating a new sandbox for each private mode application requires changing the private mode application package's name so that both public and private versions of applications can co-exist on the same platform. This has potential of causing name confusions for the application running in private mode. We rate usability of this approach as *moderate* and we further discuss this issue in Section V.

4) Ease of Implementation: Implementing the extra sandboxes involves modifying the source code of Android framework. Nonetheless, it is easier than Linux container approach of modifying the kernel layer. Therefore, we rate the ease of implementation as *good*.

D. Choosing an Appropriate Design

We summarize the merits and demerits of each of these approaches in Table I. By looking at the comparison, people can chose an appropriate design approach based on different user requirements. For instance, requirements that focus primarily on security may prefer the Linux container approach. However, in this paper, we chose the extra sandbox approach, as it provides a balance between security and usability. It has a moderate level of security, efficiency, and usability and it is relatively easy to implement. Thus, PrivateDroid applies the extra sandbox approach with additional usability features, such as private mode indicators.

IV. Implementation

PrivateDroid is implemented on Android OS version 4.3.2.1, Jelly Bean (API Level 18). In PrivateDroid, we added 367, modified 4, and removed 14 source lines of code across 18 files, added 1 additional file containing 20 source lines of code, and added 12 image icon files: six of both icons to accommodate the divers display resolutions. The added icons are the public and private mode selectors and indicators. PrivateDroid is tested on an unlocked LG E960 Google Nexus 4 Global System for Mobile Communications (GSM) Phone with 8 GB of internal storage.

A. Creating the Extra Sandbox

The extra sandbox for a private mode application is created by reinstalling the application with a different package name. Reinstalling creates a new user ID and a new data directory for the private mode application. Therefore, whenever a user launches an application in private mode, it will run in a completely new state and process without affecting any functionalities or data storage of the original application in public mode. The following technical steps are performed to reinstall the application and create a new sandbox.

1) A system application, `Launcher`, is modified to detect whether an application is launched in private or public mode. When launching an application in private mode, `Launcher` sends an intent with an updated package name to another system application, `PackageInstaller`, which is responsible for every application installation. To obtain a new package name for private mode applications, we simply append ".private" at the end of the original package name.
2) In normal situations, `PackageInstaller` prompts the users to confirm installation and approve permissions before initializing the installation process. However, in a private mode scenario, such prompts are not required as users have already approved them during the first installation. Therefore, we modify `PackageInstaller` to bypass all of the prompts, when the application for installation is a private mode application.
3) `PackageInstaller` then interacts with the `PackageManager` and the `PackageManagerService` to complete the installation process. We also modify `PackageManager` and the `PackageManagerService` accordingly to bypass verification process and ensure that package names, process names and data directories of private mode applications are changed accordingly.
4) Once the reinstallation is successful, `Launcher` launches the private mode application by starting its main activity via an intent.

PrivateDroid does not allow users to launch system applications, such as the Settings application, in private mode. This is to avoid having duplicate settings in a mobile device. Instead, PrivateDroid triggers the message in Figure 1, when users attempt to launch system applications in private mode.

Moreover, PrivateDroid does not permit users to run the same application in public and private mode simultaneously. This is due to our implementation decision not to change the component names, such as the names of activities, services and content providers, of the applications in private mode.

TABLE I
Comparison among Different Approaches for Private Mode

Approach	Security	Efficiency	Usability	Ease of Implementation
I. Code Instrumentation	Bad	Moderate	Moderate	Moderate
II. Linux Container	Good	Bad	Good	Bad
III. Extra Sandbox	Moderate	Moderate	Moderate	Good

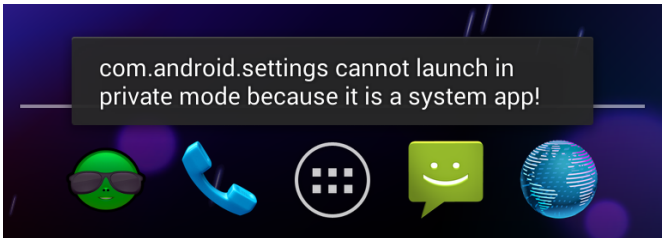

Fig. 1. Warning Message for Launching Systems Applications in Private Mode

Changing their names without changing the application source code may cause the applications to crash. On the other hand, not changing their names introduces complications when both public and private mode applications run simultaneously. This is mainly caused by the use of intents, as intents communicate via the class names of the receiving application. Therefore, in the current implementation or PrivateDroid, we do not allow users to run the same applications in both public and private modes. Instead, we show a warning message, as demonstrated in Figure 2.

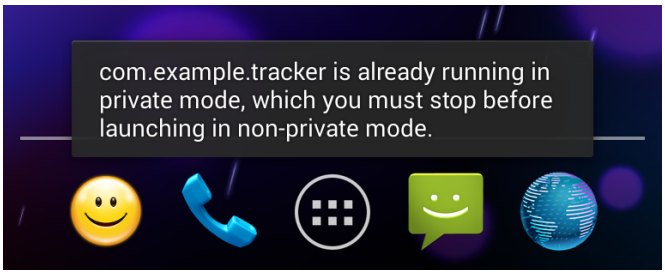

Fig. 2. Warning Message for Simultaneously Running Public and Private Mode Applications

B. Destroying the Extra Sandbox

PrivateDroid implements this capability in the recent applications panel, shown in Figure 3, because it tends to be the location where the users generally close applications. From the user standpoint, closing an application running in private mode is accomplished exactly the same way as closing an application in public mode. To implement this capability, we modified `RecentsPanelView` of the `SystemUI` package. Once the user "flings" or "swipes" a private mode application from the recent applications panel, we call the `PackageManager`'s `deletePackage` method to uninstall the application. Uninstallation removes both the data directory and the source Application Package (APK) file generated during the private mode installation process. Thus, a completely new sandbox will be re-created, the next time the user launches the application in private mode.

C. Improving the Extra Sandbox

Destroying the extra sandbox only removes the internal storage of private mode applications. However, there are still chances that private mode applications may have left data in

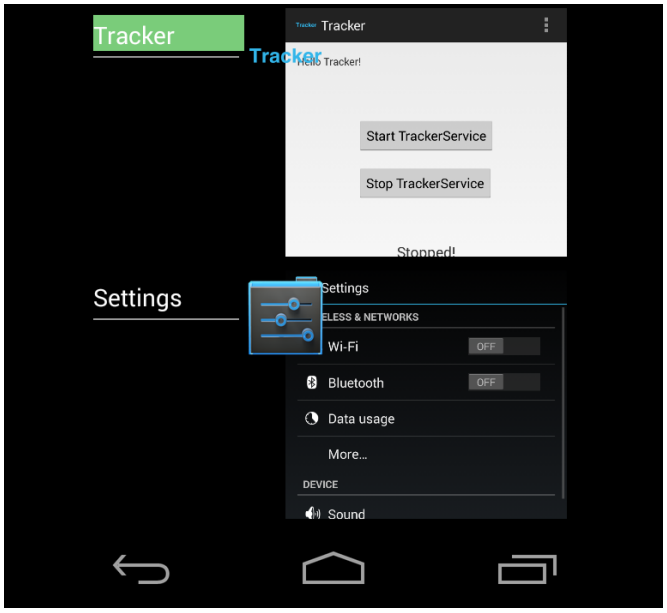

Fig. 3. Private Mode Application in Recent Applications Panel

external storage, which is mainly the SD card of the device. Therefore, we modify the Android OS `Environment` to indicate that the external storage is not present when a private mode application tries to check the status of external storage via the `getExternalStorageState()` method. Moreover, the `getDataDirectory()` and `getExternalStorageDirectory()` methods are modified to return the internal storage directory instead when a private mode application tries to retrieve the external storage paths of the devices. Thus, all the data of private mode applications are stored in their internal storage directories and are removed once the applications are closed.

Here we assume that applications normally use the `getDataDirectory()` and `getExternalStorageDirectory()` methods to access the external storage. The assumption is reasonable, as the exact path of external storage varies from device to device and applications are recommended to use these methods. Moreover, in this case, we are mainly protecting against local attackers and not web attackers. Thus, we accept the slight possibility that applications may still access the external storage without calling these methods, such as by getting the `root` directory and navigating the subdirectories dynamically.

D. Spoofing the IMEI

Implementing the extra sandbox takes care of local attackers by removing any data that the application may have stored on the phone. However, web attackers can still uniquely and persistently identify the users by accessing the unique identifiers of the mobile devices and sharing state information with a remote server. Hence, PrivateDroid also incorporates spoofing a fake IMEI for applications launched in private mode.

To obtain the IMEI, applications call `getDeviceId()` method of `TelephonyManager`. Thus, the spoofing mechanism is implemented within this method and spoofing occurs, whenever applications in private mode call this method. To make the spoofed value appear legitimate, PrivateDroid emulates the structure defined by the Global System for Mobile Communications Association (GSMA) technical specification of "TS.06 IMEI Allocation and Approval Process" [4]. It defines the IMEI as a 15-digit value with 3 fields as shown in the Figure 4. The first 8 digits represent the Type Allocation Code (TAC), a unique identifier for the specific type of device. The next 6 digits contain the serial number assigned by the manufacturer of the device. The final digit is a check-sum digit generated by Luhn's algorithm, defined by technical specification GSM TS 02.16 [5].

TAC	Serial No	Check Digit
NNXXXXXX	ZZZZZZ	A

Fig. 4. Structure of the 15-Digit IMEI

PrivateDroid creates a legitimate IMEI value by starting with the real TAC of the device, generating a pseudorandom serial number, and calculating a check-sum for the entire sequence of digits. This value, returned to the calling private mode application, would pass formatting or validity checks that applications could perform. However, note that if the user closes a private mode application and re-launches it in private mode, PrivateDroid will spoof a different IMEI. Consequently, an application that makes multiple requests for the IMEI will be able to detect the spoofing.

E. Selecting and Indicating Private Mode

Since user-friendliness is one of the primary requirements of PrivateDroid, we incorporate multiple private mode indicators in our framework. First of all, PrivateDroid includes a switch between private mode and public mode as shown in the Figure 5. The yellow icon indicates normal browsing mode, while the green icon indicates private browsing mode. The two modes can be toggled with one tap from the user. Once the private mode switch is on, all the applications will be launched in private mode. We implement this indicator by modifying the `Hotseat` within the `Launcher` application.

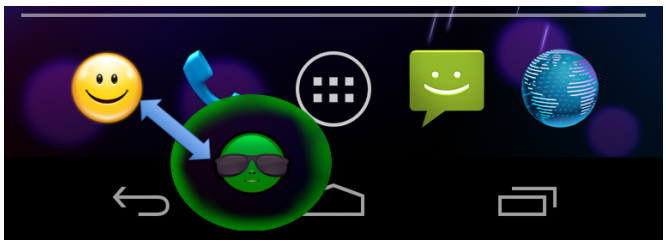

Fig. 5. Private Mode Selector/Indicator

The second indicator is displayed in the recent application panel, shown in Figure 3, in which all the open applications are listed. The labels of applications in private browsing mode are highlighted in green, while public mode applications have the default transparent background. This is also the place, where users can close the private mode applications and destroy their state information by "flinging" them out of the stack.

The third and fourth indicators are shown in Figures 6 and 7. When a user launches an application in private mode, a notification is sent to the user and the notification light flashes as a reminder of the private mode state. PrivateDroid incorporate these indicators by modifying the `NotificationManagerService` of the Android framework. These indicators are chosen, in such a way that they do not interfere with the user interfaces of third-party applications. At the same time, they allow users to easily distinguish between applications that are run in private browsing mode and those run in public mode.

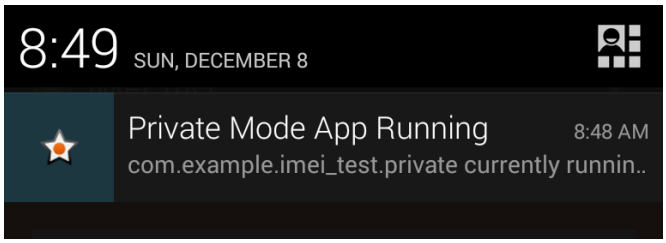

Fig. 6. Private Mode Notification Message

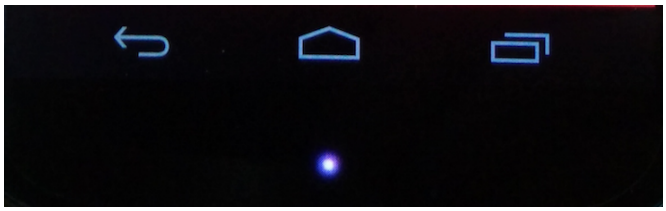

Fig. 7. Private Mode Notifications

V. Evaluations, Limitations and Suggestions

We evaluate PrivateDroid with 25 popular applications that each have at least 5 million downloads. We exclude the applications that are unlikely to be used in private browsing mode, as they do not appear to store users' state or information. Examples of these applications include antivirus, weather, wallpaper, and flashlight applications. Table II shows a list of tested applications with their package names as well as how we categorize them.

With these applications, we test the usability (i.e., whether their functionalities are affected by our modified private mode framework) and the efficiency (i.e., how long they take to launch in private mode) of PrivateDroid. We also qualitatively evaluate the security of PrivateDroid (i.e., to what extent PrivateDroid can prevent local and web attackers).

A. Security

Although PrivateDroid removes state information from external and internal storage and spoofs IMEIs with seemingly-

TABLE II
TESTED APPLICATIONS BY NAMES

Application Name	Package Name
4 Social Networking Applications	
Facebook	com.facebook.katana
Instagram	com.instagram.android
Linkedin	com.linkedin.android
Twitter	com.twitter.android
12 Utility Applications	
Adobe Reader	com.adobe.reader
Amazon Kindle	com.amazon.kindle
Barcode Scanner	com.google.zxing.client.android
Chrome Browser - Google	com.android.chrome
CNN App for Android Phones	com.cnn.mobile.android.phone
Ebay	com.ebay.mobile
IMDb Movies & TV	com.imdb.mobile
Pandora	com.pandora.android
Viber	com.viber.voip
WhatsApp Messenger	com.whatsapp
Yelp	com.yelp.android
Youtube	com.google.android.youtube
9 Game Applications	
Angry Birds	com.rovio.angrybirds
Bejeweled Blitz	com.ea.BejeweledBlitz_na
Clash of Clans	com.supercell.clashofclans
Cut the Rope FULL FREE	com.zeptolab.ctr.ads
Drag Racing	com.creativemobile.DragRacing
Fruit Ninja	com.halfbrick.fruitninjafree
Subway Surfers	com.kiloo.subwaysurf
Temple Run	com.imangi.templerun
Temple Run 2	com.imangi.templerun2

legitimate fakes, there are still other security features that have yet to be implemented.

1) Limiting Access to Content Providers: Redirecting external storage access and removing data from internal storage are not enough to ensure that no persistent state or data are stored by private mode applications. Applications can still leave data in content providers or databases. To make matters worse for PrivateDroid, there are centralized content providers, such as Calendar and Contacts List, that do not belong to a private mode application.

One way to limit the possibility of private mode applications establishing persistent data is to apply a taint tracking mechanism [6] while allowing private mode applications to access content providers. With fine-grained taint tracking, if an application modifies a row within a content provider, that change can be reversed once the private session is closed. However, taint tracking is very resource-intensive with regard to processing and battery consumption. In addition, taint tracking, with its many known bypasses, is not secure [7].

Another way is to modify the Android framework to redirect access attempts to content providers. This can be accomplished by modifying `ContentResolver` methods, which provide the basic Create, Retrieve, Update, and Delete (CRUD) functions to access content providers. Unfortunately for PrivateDroid, application intents provide another indirect way of accessing content providers. Intents allow an application to call another application with access to perform desired actions on its behalf. As intents are part of Android's IPCs, we will discuss more about them in the following subsection.

2) Limiting Inter-Process Communications: IPCs in Android include intents and binders. They allow communications between application components and provide an additional path of communication that could lead to compromising private mode. One way to solve this problem is to block all intents from private mode applications. Obviously, such a modification will result in application failures.

Alternatively, one can analyze private mode applications' intents and redirect those that are bound for public mode applications to the private mode version of the intended receiving applications. Analyzing IPCs in Android has been performed in ComDroid by E. Chin et al. [8]; however, ComDroid only performs static analysis and is still has an open question on how to perform dynamic analysis on Android IPCs. Our paper does not cover this since the dynamic analysis itself is a big, open research problem beyond this scope.

3) Spoofing Other User Identifiers: Our implementation of the private mode framework only spoofs devices' IMEIs. This is because the IMEI is one of the main sources of privacy leaks in Android devices, as shown by E. Chin et al. [9], C. Gibler et al. [10], and Z. Yang et al. [11]. However, there are other user and device identifiers, such as the phone number, email accounts, subscriber IDs, IP addresses, etc. These unique identifiers should also be spoofed to build a security-robust private mode framework.

B. Efficiency

In this section, we evaluate the efficiency of the extra sandbox approach by calculating the application launch time. Launch time is the main bottleneck in the extra sandbox approach, as most private mode functions, such as reinstallation, are performed during the launch. After the launch, users will notice little or no difference in CPU or battery usage between private and public mode applications. We calculate the launch time as the time differences between when the user taps the application launcher icon and when the application user interface is triggered. The loading time of applications is excluded in our prototype.

First, we launch each application 5 times in private mode and another 5 times in public mode. We then calculate average scores of launch times and compare the launch times between private and public modes. Figure 8 shows the average launch time difference for all tested applications. The time difference for each application is obtained by subtracting average launch time in public mode from average application launch time in private mode. Overall, private mode prolongs the application launch by an average of 7.108 seconds. However, note that in terms of run-time and battery life, there is little or no difference between private mode and public mode, once the private application is re-installed and launched.

Moreover, we still can improve the efficiency of PrivateDroid by pre-installing the private applications before the launch. This can be accomplished by having a system thread

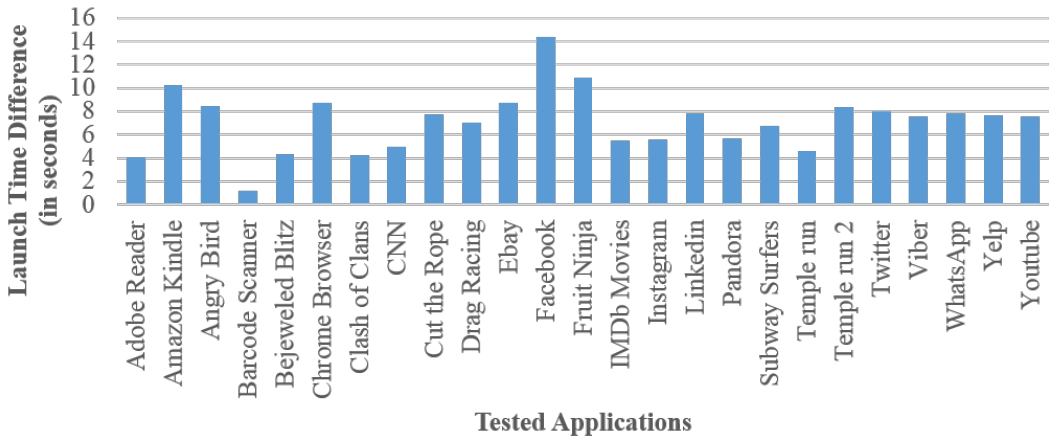

Fig. 8. Average Launch Time Difference between Applications in Private and Public Mode

create an extra sandbox for each application, once the original installation is finished or after closing a private mode application, as described in section IV-B. This solution comes with storage overhead because all of the applications on the device need to be duplicated for the private mode. Thus, in situations where the efficiency is more important than storage capacity, we suggest modifying PrivateDroid to pre-install applications in their private mode extra sandboxes.

C. Usability

Re-installation is successful for all applications (i.e. extra sandboxes can be properly created for all these applications). After re-installation, 18 out of 25 tested applications run well without any errors. Their functionalities are not affected by the extra sandboxes. However, 7 applications throw errors when certain functions are triggered. Table 9 compares the number of applications that function normally in private mode and the number of applications that do not in each application category. The result is not very satisfactory, considering that popular applications, such as Facebook, can crash in PrivateDroid. We include the detailed explanations of the thrown errors and make a few suggestions so that such mistakes can be avoided in a future implementation.

1) Permission Denial: Ebay, Linkedin, and Viber each throw a `SecurityException` when they are launched in private mode. Analyzing the error messages shows that these applications in private mode still try to access content providers and services from the original, public mode applications. Whereas Ebay and Linkedin throw this exception when they attempt to access their own content providers, Viber throws this exception when it tries to start its services. These permission denial problems are caused by the confusion in component names because we do not rename application components, such as content providers and services, during private mode installation. The solution involves renaming all of the components and modifying the source code of private mode applications that access these components. In other words, code instrumentation and application repackaging

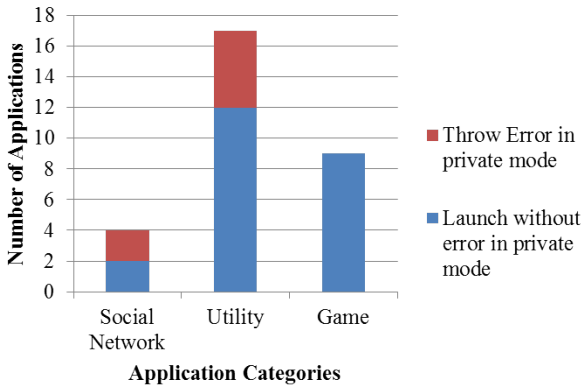

Fig. 9. The number of applications for each category

should be incorporated with the extra sandbox approach to obtain higher usability.

2) Third-Party Libraries: The Facebook, Amazon Kindle, and Pandora applications each throw a `Resources$NotFoundException` upon launching. Analyzing the errors shows that they are thrown from other packages or libraries that exist inside the private mode applications. For instance, the exception from Pandora (`com.pandora.android`) is thrown from the `com.comscore.analytics` analytics library inside the application.

We hypothesize the explanation as follows. The third-party libraries or packages included inside the applications can access the resources from the original applications by importing the `R.java` files. The syntax used for accessing resources is `[<package_name>.]R.<resource_type>.<resource_name>`. Thus, it is likely that these libraries may have attempted to import resources using the original package name while the package name of the application had been changed for private mode (i.e., they have been concatenated with ".private"). This subsequently leads to `ResourceNotFound` exceptions in private mode applica-

tions. Similar to the solution in the previous section, the best solution to solve this error is to change the source code of external libraries so that they only import and use the `R.java` files from the private mode applications and not from the original, public mode applications.

VI. Related Work

Our related work is divided into two parts. The first part is about the problem of tracking by third-party servers, cookie management, and the private browsing mode of PC browsers. The second part is related to spoofing user identities and protecting data in mobile devices.

A. Private Browsing and Third-Party Tracking

G. Aggarwal et al. [1] define the goals of private browsing, its threat model, and survey its implementation in different modern browsers, including Firefox, Chrome, Internet Explorer, and Safari. E. Y. Chen et al. [12] introduce an application isolation mechanism in which a user can enjoy a multi-browser experience in a single browser. A. M. Dunn et al. [13] propose Lacuna, which allows private applications to talk to peripheral devices, such as graphic, sound, and Universal Serial Bus (USB) input devices. Moreover, J. Wang et al. [14] introduce an isolated, lightweight virtual environment in browsers called SafeFox. SafeFox runs in its own process namespace, file system, and IP address in a browser. Similarly, K. Onarlioglu et al. [15] propose Privexe, which allows desktop applications to use OS services for private execution. However, these solutions are mainly intended for computer browsers and cannot be applied directly in mobile phones.

U. Shankar et al. [16] introduce a cookie management system named Doppelganger. With Doppelganger, users can set fine-grained privacy policies for cookies (locally-stored state information) when using a web browser. This is similar to our private mode implementation in that users choose which applications are to run in private mode. Yet, cookies are not the only aspect of protecting users' privacy. T.-F. Yen et al. [17] show that even deleting cookies cannot prevent third-party websites from tracking users' information because of other information available to remote servers. J. R. Mayer et al. [18] provide a comprehensive survey on different kinds of technologies used in tracking by third-party websites, policies against tracking, and users' opinions on defending against tracking. F. Roesner et al. [19] develop a client-side detection tool, which can automatically detect tracking from third-party websites and analyze the tracking behavior. They also categorize five types of mainstream web tracking: third-party analytics, advertising, advertising with pop-ups, advertising networks, and social widgets.

B. Protecting Data in Mobile Devices

There are many mobile platform extensions that limit third-party applications' access to mobile users' identities and other personal information. AppFence [20] introduces two privacy controls: substituting sensitive data with shadow data and blocking the network transmission of sensitive data. The authors then examine how these controls affect applications' user interfaces.

For users who prefer to choose their mode for releasing private data, Taming Information Stealing Smartphone Applications (TISSA) [21] provides four options to users: trusted, empty, anonymized, and bogus. The trusted option operates as in normal phones via a permission system. The empty option simply returns an empty result, indicating that the requested information is not present. The anonymized option still allows the data access but anonymize the users. On the other hand, the bogus option provides a fake result of the requested information. MockDroid [22] allows users to try applications' functionalities without compromising the user data by providing only empty data. The protected data include location, Internet, Short Message Service (SMS)/Multimedia Message Service (MMS), calendar, contacts, device ID, and broadcast intents. Although the above solutions also spoof the IMEI with legitimate-looking fakes, the exact mechanism of spoofing is not provided, unlike this paper.

Some other mechanisms provide fine-grained access control policies to users. They allow users to define policies on when and how frequently the data can be accessed. Apex [23] allows users to set "always allow," "always deny," or "allow under certain constraints" on Android's permission list. Very similar to Apex, Context-Related Policy Enforcement (CRePE) [24] allows users to set run-time constraints on application data access. The constraints are based on the time and location contexts of users. These solutions, however, focus on the access control mechanism of personal data to third-party applications. They do not emphasize removing state information to prevent local attackers. Other solutions, such as CleanOS [25], constantly encrypts the sensitive data on mobile devices and stores the key on the cloud. However, unlike PrivateDroid, these mechanisms do not prevent local attackers, such as family members, or web attackers.

VII. Conclusion

With the expanding use of mobile devices and third-party applications, security and privacy concerns arise concurrently. Private mode, which clears the history, cache, and state information, is one way of improving mobile user privacy. In this paper, we initiate the effort of extending the private browsing mode concept to third-party mobile applications. Our paper first applies the threat model of local attackers and web attackers in the context of third-party mobile applications. We describe the fundamental differences between implementing private browsing mode in web browsers on PCs and private mode in mobile devices, highlighting the challenges of introducing private mode to mobile devices. We contemplate various ways of implementing a private mode for mobile applications and provide their relative advantages and disadvantages. Our paper also includes implementation details of PrivateDroid, which embraces a balanced level of security, efficiency, and usability. Limitations of PrivateDroid are also objectively identified and prospective improvements

are suggested in the paper. Thus, we believe that our paper is the first step toward creating a better private mode mechanism for future mobile devices.

VIII. Acknowledgment

This research is supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office, Media Development Authority (MDA).

References

[1] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh, "An analysis of private browsing modes in modern browsers," in *Proceedings of the 19th USENIX Conference on Security*, ser. USENIX Security '10. Berkeley, CA, USA: USENIX Association, 2010, p. 6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929820.1929828

[2] B. Leiba, "OAuth web authorization protocol." *IEEE Internet Computing*, vol. 16, no. 1, pp. 74–77, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/internet/internet16.html#Leiba12

[3] R. J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2nd ed. Wiley Publishing, Inc., 2008.

[4] (2013, October) IMEI allocation and approval process vers. 7.0. GSM Association. [Online]. Available: http://www.gsma.com/newsroom/wp-content/uploads/2013/12/TS.06-v7-Approved.pdf

[5] (2000, August) Digital cellular telecommunications system (phase 2+) international mobile station equipment identities (IMEI) (GSM 02.16 version 5.2.0 release 1996). Global System for Mobile Communications (GSM). [Online]. Available: http://www.etsi.org/deliver/etsi_gts/02/0216/05.02.00_60/gsmts_0216v050200p.pdf

[6] W. Enck, P. Gilbert, B.-G. 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 Conference on Operating Systems Design and Implementation*, ser. OSDI '10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924971

[7] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices." in *SECRYPT*, P. Samarati, Ed. SciTePress, 2013, pp. 461–468. [Online]. Available: http://dblp.uni-trier.de/db/conf/secrypt/secrypt2013.html#SarwarMBK13

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '11. New York, NY, USA: ACM, 2011, pp. 239–252. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000018

[9] J. Kim, Y. Yoon, K. Yi, and J. Shin, "ScanDal: Static analyzer for detecting privacy leaks in Android applications," in *MoST 2012: Mobile Security Technologies 2012*, H. Chen, L. Koved, and D. S. Wallach, Eds. Los Alamitos, CA, USA: IEEE, May 2012. [Online]. Available: http://ropas.snu.ac.kr/scandal/

[10] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale," in *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, ser. TRUST '12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 291–307. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30921-2_17

[11] Z. Yang and M. Yang, "LeakMiner: Detect information leakage on Android with static taint analysis," in *Proceedings of the 2012 Third World Congress on Software Engineering*, ser. WCSE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 101–104. [Online]. Available: http://dx.doi.org/10.1109/WCSE.2012.26

[12] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson, "App isolation: Get the security of multiple browsers with just one," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 227–238. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046734

[13] A. M. Dunn, M. Z. Lee, S. Jana, S. Kim, M. Silberstein, Y. Xu, V. Shmatikov, and E. Witchel, "Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '12. Berkeley, CA, USA: USENIX Association, 2012, pp. 61–75. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387887

[14] J. Wang, Y. Huang, and A. K. Ghosh, "SafeFox: A safe lightweight virtual browsing environment." in *HICSS*. IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dblp.uni-trier.de/db/conf/hicss/hicss2010.html#WangHG10

[15] K. Onarlioglu, C. Mulliner, W. K. Robertson, and E. Kirda, "PrivExec: Private execution as an operating system service." in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013, pp. 206–220. [Online]. Available: http://dblp.uni-trier.de/db/conf/sp/sp2013.html#OnarliogluMRK13

[16] U. Shankar and C. Karlof, "Doppelganger: Better browser privacy without the bother," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 154–167. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180426

[17] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi, "Host fingerprinting and tracking on the web: Privacy and security implications," in *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, Feb. 2012.

[18] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 413–427. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.47

[19] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against third-party tracking on the web," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, ser. NSDI '12. Berkeley, CA, USA: USENIX Association, 2012, p. 12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228315

[20] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 639–652. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046780

[21] 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*, ser. TRUST '11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 93–107. [Online]. Available: http://dl.acm.org/citation.cfm?id=2022245.2022255

[22] 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*, ser. HotMobile '11. New York, NY, USA: ACM, 2011, pp. 49–54. [Online]. Available: http://doi.acm.org/10.1145/2184489.2184500

[23] 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*, ser. ASIACCS '10. New York, NY, USA: ACM, 2010, pp. 328–332. [Online]. Available: http://doi.acm.org/10.1145/1755688.1755732

[24] M. Conti, V. T. N. Nguyen, and B. Crispo, "CRePE: Context-related policy enforcement for Android," in *Proceedings of the 13th International Conference on Information Security*, ser. ISC '10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 331–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=1949317.1949355

[25] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "CleanOS: Limiting mobile data exposure with idle eviction," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI '12. Berkeley, CA, USA: USENIX Association, 2012, pp. 77–91. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387888