Quantifying the Effects of Permission Removal from Android Applications

By

KRISTEN M. KENNEDY
B.E,M.E (Catholic University of America)

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Committee in charge

2013

Quantifying the Effects of Permission Removal from Android Applications

Abstract

With the growing popularity of Android smart phones, it is increasingly important to ensure the security of sensitive user information. A recent study found that approximately 26% of Android applications in Google Play can access personal data, such as contacts and email, and 42 percent, GPS location data [6]. While Android is known for giving the user control, it falls short when it comes to enabling and disabling the permissions on applications. Currently, the user is given the option to either give the application every permission it desires or not install it. While researchers have proposed approaches for allowing users to modify the permissions granted to applications, it is unclear how removing permissions would affect the behavior of current applications. At present, developers expect all requested permissions to be granted.

This study takes the first step towards quantifying the impact of enabling users to statically remove permissions on Android applications post-installation. An automated testing system, Pyandrazzi, was developed for evaluating the effect of removing individual permissions from applications. Using Pyandrazzi, the effects of removing common permission were evaluated using sets of randomly selected applications that request them. It was determined that approximately 5.8% of the 700 applications tested crash after a permission is removed and that the removal of certain permissions is handled more gracefully than others. The results of this study will help users make more informed decisions when removing permissions and help developers make their applications more robust to permission revocation.

Acknowledgments

xxxxxxx

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Android is by far the most popular smartphone operating system at the time of this writing. With around 75% of the global smartphone market [10], its application ecosystem has grown along with it. According to a recent Google press release, Google Play now has around 700,000 applications [12]. Unlike the Apple iOS ecosystem whose applications are centrally verified before becoming available, Android applications are more lightly vetted. This puts more responsibility on the user to make the important decision of which applications they trust. The permissions architecture employed by Android, however, makes these trust decisions more difficult.

1.0.1 Mobile Device Permission Models

In the current implementation, application developers are responsible for defining the set of permissions their application asks for independent of what their applications code actually uses [4]. The list is presented to the user at install-time whether installing an application obtained through the Google Play market or elsewhere. This list contains the canonical names given to each permission object, but provides little context as to what the permission is being used for. In the latest API, Google has added more description and added warning levels . At the end of the day, however, users have no choice but to accept the permissions list as presented to them, or not install the application.

While all smartphone applications are susceptible to being over permissioned, Apple's iOS and Blackberry's OS both give the user the ability to revoke an applications permissions. iOS uses a "dynamic" permissions model; the user is prompted once per application to use phone features such as the GPS receiver. The user then has the option of changing their decision through the system's settings menu.

Blackberry OS uses a "static" approach similar to that of Android. Application developers declare the set of permissions they would like their program to have, and this list is presented to the user at installation. However, unlike Android, the user has the option to edit the list and revoke any permission. This approach adds a degree of granularity to an otherwise binary trust decision and is the approach explored in this study.

1.0.2 Removing Permissions from Android Apps

Shown in Figure 1.1, is the application permissions listing for a Flashlight application obtained through the Play market. While one could reason this application would only need the hardware controls necessary to turn on the phone's light, due to the large quantities of ad network libraries and other debris embedded within, it asks for much more. If permissions were editable, the user would be able to trust this applications ability to operate the flashlight only, and not have to worry about the true reason for the other permission requests.

There are various ways one could remove permissions from Android applications. Previous work in this area proposes a "privacy mode", where applications running in this mode run with lowered permissions [13]. This approach requires heavy modification to the Android OS itself, but yields a flexible solution. Applications can also be repackaged, with their manifests modified to contain fewer permissions. This is the

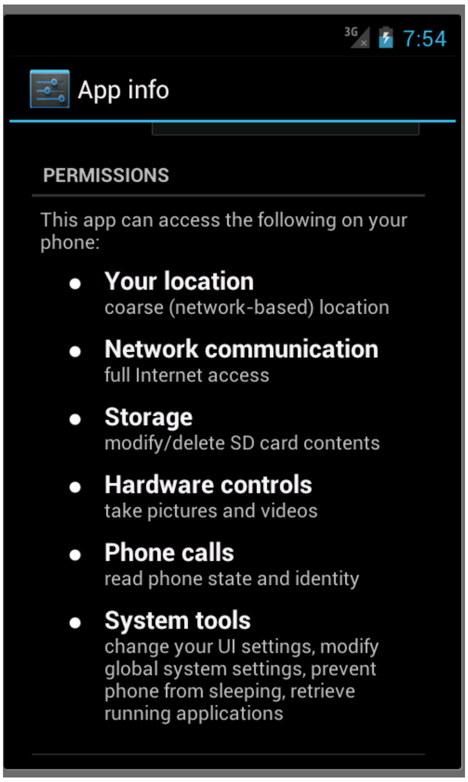

Figure 1.1: Flashlight app permission requirements

most popular approach at present, as applications containing this functionality can be readily obtained from major application markets [7]. Additionally, the application's executable code could be dynamically rewritten to protect sensitive API calls [2].

All of the above approaches, however, are dependent on Google's policy regarding application permissions, as they influence how developers write their applications. Google's documentation discusses the implementation in enough detail for a developer to use it, but does not go into the rationale behind the approach. While they do suggest that the "dynamic" permission model would be too much of a burden on the user, they do not mention editable static permissions at all. Furthermore, application developers are not explicitly instructed to handle cases of revoked permissions in their applications. Since the lack of a required permission is typically implemented in the form of a Java SecurityException [5], some programmers will handle these gracefully out of habit. However, if an application or library it uses is not explicitly written to handle the permission revocation, modification of the source code will be required to ensure proper function.

1.0.3 Effect of Permission Removal

While previous research has demonstrated that a significant number of Android applications request and use too many permissions [4], it is unclear how removing permissions would affect the application's behavior. Under Android's security model, developers typically expect all the requested permissions to be available. Therefore, it would not be surprising if many developers do not handle the lack of permissions gracefully. When an application invokes an API call without the necessary permissions, it typically throws a `SecurityException`. However, if such an exception is caught by library or wrapper code, the application can continue running. Moreover, permissions are not all equal. Some permissions describe a devices' hardware capabilities, such as `CAMERA`. Since they are not universally available, one would expect libraries to be able to handle their absence more gracefully. A relatively common instance of this is the removal of the camera on DoD phones for operational security.

When removing a permission, users will expect to see any correlating features be disabled. An everyday example being the removal of GPS capabilities from Google Maps. Currently, many users turn their GPS off in attempts to extend their battery life. When this is done, at launch, the user is notified that the accuracy of Google Maps would be improved if GPS was enabled. When removing permissions from and application, users are going to expect similar behavior. If an application was not written with permission removal in mind, however, features that are not obviously correlated may malfunction leaving the user confused. This could theoretically include data corruption if the application is not written to handle its data in a robust manner. If this is the case, data would be susceptible to being corrupted if the application crashes regardless of permission removal. While these occurrences are of concern, in the long run, they can easily be addressed by developers.

In this paper, we take the first step to quantitatively measure the effects of removing permissions from Android applications by trying to answer the following questions:

- How likely will an application crash after a permission is removed?
- Which permissions, if removed, are less likely to cause an application to crash?
- Why do applications handle the lack of certain permissions more gracefully?

Our work will benefit both users and developers. Users can make more informed decisions when deciding which permissions to remove (when they use tools for removing permissions). Application developers can make their code more robust against missing permissions. Android library developers can design their libraries to handle lack of permissions more gracefully.

In the remainder of this paper, we will attempt to quantify the effects of removing permissions from Android applications. In Section 2 we will discuss related work. Then, in Section 3, we will discuss our methodology and the tool we developed for automated testing, PyAndrazzi. We present the results of our testing in Section 4 and discuss the impact of our findings on users and developers in Section 5. Finally, in Section 6 we discuss future work and conclude with Section 7.

Chapter 2

Related Work

Recently, there have been a number of academic and non-academic works in the area of Android permissions. The work that most closely relates is [8]. In their work, they attempt to analyze the effects of returning fake data to the sensitive API calls who's permissions they want to restrict. The behavioral changes are analyzed by using image comparison techniques. The work performed in this study differs from their's in four significant ways. Firstly, the removal of permissions is tested without modifying the Android framework, giving a more realistic experiment. Secondly, all of the testing is performed autonomously on emulators and does not require human generated application specific scripts, resulting in a scalable testing method. Thirdly, the samples are used completely random where as their's are significantly skewed towards over permissioned applications which results in an admittedly overestimate of the side effects. Finally, the applications exception behavior is observed instead of using image analysis to detect changes when permissions are removed. The overall difference is this study attempts to quantify the behavior of an application when a permission is removed, where as [8] focuses on analyzing the effects of using fake instead of removing permissions from applications.

In [11], they also performed simlar work. Like this study, they used UI introspection; however, they utilized humans to record input while the approach utilized in this study generates dynamic input. In addition, they focus on network traffic while the focus of this research is on an application's on device behavior.

Some third party Android distributions, such as CyanogenMod [3] and MockDroid [1], also modify the Android OS to enable permission removal. Like [8], MockDroid provides fake data to API calls where permissions are being removed. CyanogenMod, however, enables the user to actually revoke permissions. While CyanogenMod essentially accomplishes the changes proposed by this study, the developers of CyanogenMod state concerns about applications failing as a result of permission removal. In addition, these third party distributions are cumbersome to many non-technical users due to the fact that they require you to re-flash the firmware of your device thus voiding its warranty. As a workaround, others have developed applications that allow users to implement the security model described earlier. With the exception of the latest to hit the market, Plop, they all require rooting. Among these, LBE Privacy Guard and PDroid provide fake data to handle security exceptions while Plop and Permission Denied do not [7]. Using fake data theoretically decreases the number of exceptions incurred, however, it is still unable to handle all of them, i.e. writing to external storage. Ultimately, revoking permissions or providing fake data, regardless of which method one uses, can lead to applications crashing and unexpected behavior. In the end, the better solution is for Google to alter the security model of the Android OS and for developers to handle the exceptions.

Chapter 3

Design and Implementation

Pyandrazzi, a system for evaluating how removing permissions impacts the behavior of Android applications, was designed for the purpose of this study. To evaluate each application, Pyandrazzi automatically runs the application, supplies it with various UI events, detects when the application crashes, and logs the cause of the crash. Pyandrazzi consists of the following components (see Figure 3.1):

Permission Removal Pyandrazzi decodes the application's APK file using APK-Tool. Then, it removes the permission being evaluated. Finally, it rebuilds the APK and signs it using Android's built-in debug key.

Installation and Execution Pyandrazzi installs and runs applications in emulators. It utilizes the MonkeyRunner framework to install and uninstall applications, provide UI inputs, and take screen shots of applications.

Automatic UI Exploration Pyandrazzi needs to execute as much code as possible to maximize the number of adverse effects resulting from permission removal. For each application, Pyandrazzi determines the list of activities and executes each one starting with the `Main` activity. During each activity, Pyandrazzi takes a screenshot

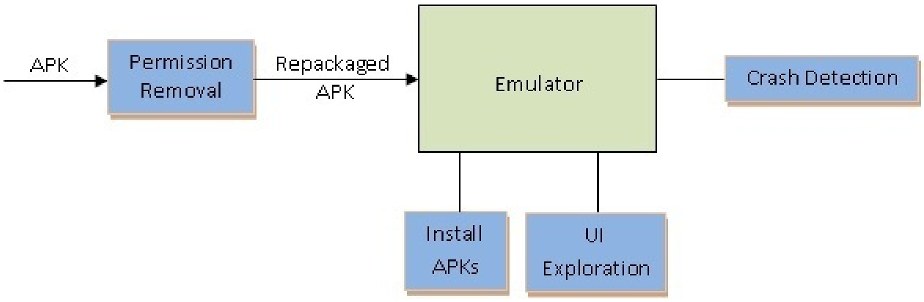

Figure 3.1: PyAndrazzi Component Diagram

and performs a series of screen touches. This functionality is implemented using a UI introspection approach based on the AndroidViewClient library [9]. Using this method, Pyandrazzi is able to query the screen for click-able elements and performs "random" touches with a high probability of changing the application state.

Crash Detection Pyandrazzi needs to detect application crashes and their causes. It uses ADB to access logcat, which provides us with the logs from the emulator, including system status and application crashes. When an application crashes due to a fatal error, such as a `SecurityException`, a separate thread monitoring the logs will notify the testing thread, which will record the cause of the crash and launch the next Activity.

Chapter 4

Findings

To evaluate the impact of removing permissions on applications, seven common permissions (Table 4.1) were selected based on their potential threat to the user's security and privacy. For each permission, 100 applications that declare the permission in their manifests were randomly downloaded. These applications are from the official Google Play Market as well as other markets in the US, China, and Europe.

A small portion (less than 3%) of the applications contained manifests with unusual control characters that APKTool would not process (eg. 0x04), and were discarded. Additionally, 2% of the applications caused the emulator to crash and were not able to be tested. Pyandrazzi installed and tested the remaining ones as described in Section 3.

Pyandrazzi detects application crashes and their causes from the emulator's logs. When an application does not have a permission required by an API call, the Android framework will typically throw a `SecurityException` and the application will then crash unless it catches the exception. An application may crash due to other reasons as noted by the fatal exception column in (Table 4.1). For example, bugs in the application, or when Pyandrazzi executes an activity that is not meant to be executed by the user. An additional possibility is that a library acting on the application's

Data Set	# Apps Run	# Fatal Exceptions	SecurityException # Exceptions	SecurityException # Apps Throwing Exceptions
WRITE_SMS Original	95	528	1	1
WRITE_SMS Removed	91	555	1	1
ACCESS_FINE_LOCATION Original	99	623	0	0
ACCESS_FINE_LOCATION Removed	99	616	18	13
CAMERA Original	97	1206	0	0
CAMERA Removed	97	1158	0	0
RECORD_AUDIO Original	93	700	0	0
RECORD_AUDIO Removed	92	659	0	0
READ_CALENDAR Original	91	581	0	0
READ_CALENDAR Removed	91	643	8	6
READ_CONTACTS Original	96	633	0	0
READ_CONTACTS Removed	96	561	40	20
INTERNET Original	95	191	13	3
INTERNET Removed	95	234	13	3

Table 4.1: Results of Random UI Introspection

behalf receives the `SecurityException`, and returns the proper error condition, such as a null pointer. Applications that do no check for this kind of error state may crash with other types of exceptions such as `NullPointerException`. Since it is not possible to determine the true causes of these exceptions without extensive static analysis of each application, this study focuses on `SecurityExceptions`, which indicate under-permission. Among all the crashes, 5.8% were due to a `SecurityException`. The seven permissions are examined separately below.

`INTERNET` The cause of the `SecurityExceptions` was ascertained by examining the logs and screen shots. The exceptions generated both before and after permission removal were the result of a change in the permissions system in Android 4.2, which requires a new permission, `WRITE_APN_SETTINGS`, for the API calls the applications performed. This is technically under-permission, but was not caused by removing existing permissions. In the remainder of our samples, ad libraries were the sole reason for an application's request for the `INTERNET` permission. All the versions of the *AdMob* libraries that we examined fail gracefully when the INTERNET permission is removed. For example, one version of *AdMob* displays a message to the user as

Figure 4.1: Admob displays a message when the `INTERNET` permission is removed.

seen in Figure 4.1.

In light of this, a copy of a popular third-party Android web browser was attained and its `INTERNET` permission removed using Pyandrazzi. When executed, the application terminated with a `SecurityException` related the applications request for DNS information. With further investigation it was determined that the application calls the `java.net` functions directly causing a `SecurityException` to be generated and the standard crash screen to be displayed (see Figure 4.2).

`CAMERA` and `RECORD_AUDIO` In the case of `CAMERA` and `RECORD_AUDIO`, both functionalities have a service that proxies application requests. If the relevant permissions are removed, they will behave as if no camera or microphone is present. Therefore, no `SecurityExceptions` were observed when these permissions were removed.

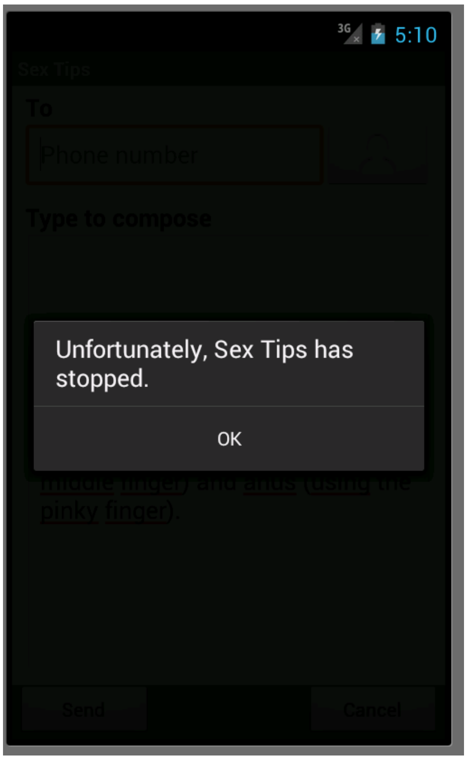

Figure 4.2: App crashes due to SecurityException when the INTERNET permission is removed.

READ_CONTACTS and READ_CALENDAR Manual investigation determined that, when removing READ_CONTACTS, a SecurityException was only being recieved in cases where the API call to read the contacts was actually occurring. While the presence of some other helper library that catches exceptions cannot completely rule out, in every case that was manually examined, the SecurityException was only occurring when the automated testing specifically activated a UI element or activity's startup code that made a request to the Provider URI, content://com.android.contacts/.

The removal of READ_CALENDAR results in similar behavior. Apps attempting to access the Provider URI, content://com.android.calendar/, generate security exceptions in all cases we have examined.

WRITE_SMS The WRITE_SMS permission also uses a Provider URI, content://mms-sms/; however, no SecurityExceptions resulting from permission removal were observed.

The one that was observed in both test cases was a result of the missing permission, `WRITE_APN_SETTINGS`, that was discussed earlier. The lack of `SecurityExceptions` is likely due to the relative depth of writes to this URI within the application, which makes it more difficult to trigger. Any unauthorized Provider URI access produces the standard crash screen (Figure 4.2).

`ACCESS_FINE_LOCATION` The fine-grained location permission, `ACCESS_FINE_LOCATION`, is a special case. It is one of a few permissions in Android that is a nested permission. Many API calls will operate in the presence of either fine or coarse location permissions but prefer fine; if fine is removed it will fall back to coarse grain location. This results in few `SecurityExceptions` (~13%), since they will only occur in cases where the GPS hardware is explicitly being used.

Chapter 5

Discussion

The evaluation shows that the rate of failure varies with the permission. In this test, removing permissions only caused 39 (5.9%) of the 662 applications to crash. In the best cases, removing `CAMERA`, `RECORD_AUDIO` and `WRITE_SMS` respectively, never caused crashes due to permission removal. While the random sample did not result in any `SecurityExceptions` with the removal of `WRITE_SMS`, it is probable that this is due to the API call being triggered by UI elements deep in the UI structure. Furthermore, the results from the random sample showed that, if an application's sole use of the `INTERNET` permission is for advertising, `SecurityExceptions` caused by its removal are likely to be handled by the ad library. If the application makes use of network functionality on its own, as in the manually selected test case, the application may crash with a `SecurityException`. Lastly, it was determined that removing the `READ_CONTACTS` and `ACCESS_FINE_LOCATION` permissions had the greatest impact causing 20 and 13 applications to crash respectively.

Wrapper Code The manual inspection shows that many wrapper code pieces handle the lack of permissions gracefully. Wrapper code includes both system services such as Audio Manager and Camera, as well as developer-defined libraries like those used for advertising. System services provide abstraction between `getHostByName()`

in the developer's code and the hardware. For example, an application trying to use the device's camera can fail gracefully in the event the device has no camera. This is also true of permissions revocation. When the relevant permission is removed, the service acts as if the hardware it manages does not exist. Third-party libraries can accomplish the same feat by catching the exception on behalf of the developer. Google AdMob, for example, will display a message to the user running an application without the necessary permissions to obtain the ads (See 4.1). Notably, it will allow the application to continue to function without advertising.

While these wrapper code pieces make it easier for the user to remove the permission without causing crashes, they make it harder for the developer to detect the lack of declared permissions programmaticaly. To address this, developers could make their own API calls to force the exception to be thrown if they wanted to take their own action on permissions revocation.

Protected URI In situations where an application accesses protected URIs directly (not through wrapper code), it was verified that the application throws `SecurityExceptions` when it accesses the URIs. In this case, the developer can easily catch and handle the exception appropriately. However, this does require explicit modification of existing code, and without it users cannot easily remove access without their applications crashing.

Protected API The use of protected API calls can result in similar behavior to that of protected URIs. It is important to note, however, that the exception-throwing behavior of each method may change between Android API revisions. In previous iterations of the testing, it was observed that the `getHostByName()` method used to perform DNS lookups failed gracefully under Android 4.0.4 (API 15) and returned an error value; applications behaved as if there was no Internet connectivity. On Android 4.2.2 (API 17), the method generates a `SecurityExceptions` when the `INTERNET` permission is removed causing applications to crash. Therefore, to ensure maximum compatibility, developers should always catch `SecurityException` when using these APIs.

Chapter 6

Limitations and Future Work

One of the limitations of this work, and dynamic analysis in general, is the issue of code coverage. The approach presented relies on exercising the application through its UI, and is unable to guarantee that all sensitive API will be executed calls within an application. Even if one were able to explore all elements of the UI as defined in the APK's resources, chances are this would still not result in complete coverage. Given that the presented approach is based solely on dynamic analysis, it is currently not possible to obtain the list of all the API calls an application is able to make. Consequently, code coverage cannot be estimated. Applications whose execution relies on a WebView present further challenges, as their contents are not visible to the UI introspection techniques used and would require a special coverage calculation scheme. In addition, Pyandrazzi does not attempt to handle native code within applications. While no native code was discovered in the random application samples, this complicates some of the dynamic analysis techniques, and would require its coverage to be measured differently as well.

In future iterations of Pyandrazzi, the intent is to implement methods to increase and accurately determine the extent of code coverage, as well as broaden the types of applications that can be closely examined. Firstly, it would be desireable to add a static analysis phase to the Pyandrazzi to allow the enumeration of API calls made by the application. VM instrumentation or application rewriting could then be used to determine when an API call is executed. Secondly, Pyandrazzi does not handle applications that depend on external third-party libraries such as Adobe Air. Additional analysis of the dataset will be required to determine the prevalence of these libraries so they can be included in the emulator image.

While testing the applications, it was observed that third-party libraries included in the APK tend to balloon the permission requirements far beyond what the applications themselves required. For example, the flashlight application mentioned earlier has most of it's permissions primarily because they are required for the numerous ad libraries it uses to serve ads to the user. In the future, mapping common libraries to the permissions they require, or even these library calls to permissions, as in [4], would be desireable. Lastly, applications that use intent receivers guarded by permissions to protect message passing are not handled; ultimately, more research needs done to determine if handling this case is necessary. It should be noted, however, that activities intended to be invoked by these receivers are already being executed.

Overall, seven different permissions that have a high security or privacy impact were successfully tested but the surface has only been scratched. In the future, the usage of many other permissions, with an initial focus on those that pose a security risk (e.g.billing), should be investigated. Furthermore, the removal of various permission combinations, such as `ACCESS_FINE_LOCATION` and `ACCESS_COARSE_LOCATION`, should be tested.

Chapter 7

Conclusion

Pyandrazzi, a system for the automated testing and measurement of the fatal exception behaviors of Android applications when permissions are removed, was developed for this study. The evaluation shows that not all permissions are equal with there rate of crashes due to permission removal ranging from 0-20%. Overall, 94% of the 662 applications that were successfully tested did not crash due to permission removal as evidenced by a lack of `SecurityExceptions`. If Google decides to implement user editable permissions, they will have the opportunity to further enhance the user experience by wrapping more sensitive API calls in libraries which handle permission errors gracefully.

In regards to developers, if they wish to make their applications more robust when requested permissions are unavailable, they should try to use wrapper code that call sensitive APIs and handle exceptions gracefully rather than calling the sensitive APIs directly. On the other hand, if they wish to restrict functionality unless the requested permissions are available (e.g., `INTERNET` permission for ad libraries that generate revenue), they should invoke the sensitive APIs directly and prevent the application from continuing until the permission is restored.

Bibliography

[1] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. "MockDroid: trading privacy for application functionality on smartphones". In: *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. HotMobile '11. New York, NY, USA: ACM, 2011, pp. 49–54. ISBN: 978-1-4503-0649-2. DOI: 10.1145/2184489.2184500. URL: http://doi.acm.org/10.1145/2184489.2184500.

[2] Benjamin Davis and Hao Chen. "RetroSkeleton: Retrofitting Android Apps". In: *The 11th International Conference on Mobile Systems, Applications and Services (Mobisys)*. Taipei, Taiwan, June 25–28, 2013.

[3] Matt Demers. *CyanogenMod Adds Support For Revoking App Permissions*. Anroid Police. May 25, 2011. URL: http://www.androidpolice.com/2011/05/22/cyanogenmod-adds-support-for-revoking-and-faking-app-permissions/.

[4] Adrienne Porter Felt. "Android permissions demystified". In: *18th ACM conference on Computer and communications security*. 2011.

[5] Google. *Permissions—Android Developers*. Google. Apr. 14, 2013. URL: http://developer.android.com/guide/topics/security/permissions.html.

[6] K. J. Higgins. *More Than 25% Of Android Apps Know Too Much About You*. Dark Reading. Nov. 1, 2012. URL: http://www.darkreading.com/mobile-security/167901113/security/privacy/240012705/more-than-25-of-android-apps-know-too-much-about-you.html.

[7] Chris Hoffman. *How to Restrict Android App Permissions*. Apr. 14, 2013. URL: http://www.howtogeek.com/115888/how-to-restrict-android-app-permissions/.

[8] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David 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*. CCS '11. New York, NY, USA: ACM, 2011, pp. 639–652. ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046780. URL: http://doi.acm.org/10.1145/2046707.2046780.

[9] D. T. Milano. *Android ViewClient*. Apr. 14, 2013. URL: https://github.com/dtmilano/AndroidViewClient.

[10] S. Perez. *IDC: Android Market Share Reached 75% Worldwide In Q3 2012*. Tech Crunch. Nov. 2, 2012. URL: http://techcrunch.com/2012/11/02/idc-android-market-share-reached-75-worldwide-in-q3-2012.

[11] Vaibhav Rastogi, Yan Chen, and William Enck. "AppsPlayground: automatic security analysis of smartphone applications". In: *Proceedings of the third ACM conference on Data and application security and privacy*. CODASPY '13. New York, NY, USA: ACM, 2013, pp. 209–220. ISBN: 978-1-4503-1890-7. DOI: 10.1145/2435349.2435379. URL: http://doi.acm.org/10.1145/2435349.2435379.

[12] B. Womack. *Google Says 700,000 Applications Available for Android*. Bloomberg Businessweek. Oct. 29, 2012. URL: http://www.businessweek.com/news/2012-10-29/google-says-700-000-applications-available-for-android-devices.

[13] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vince W. Freeh. "Taming Information-Stealing Smartphone Applications (on Android)". In: *4th International Conference on Trust and Trustworthy Computing (TRUST 2011)*. Pittsburgh, PA, June 2011.