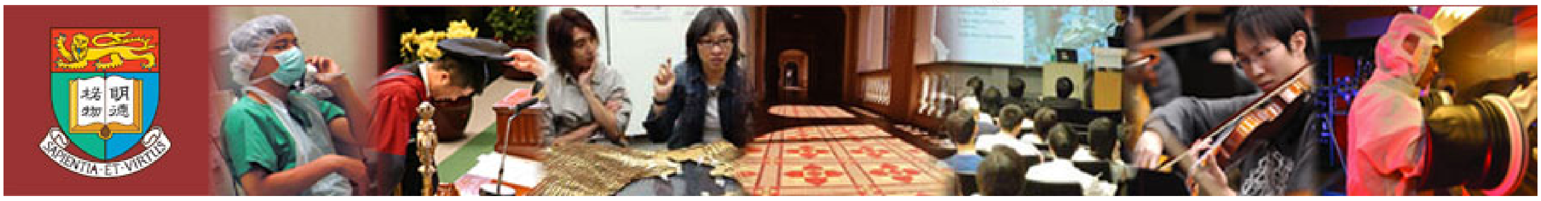

A Privilege Escalation Vulnerability Checking System for Android Applications

Patrick P.F. Chan
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
Email: pfchan@cs.hku.hk

Lucas C.K. Hui
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
Email: hui@cs.hku.hk

S.M. Yiu
Department of Computer Science
The University of Hong Kong
Pokfulam, Hong Kong
Email: smyiu@cs.hku.hk

***Abstract*—Android is a free, open source mobile platform based on the Linux kernel. The openness of the application platform attracts developers, both benign and malicious. Android depends on privilege separation to isolate applications from each other and from the system. However, a recent research reported that a genuine application exploited at runtime or a malicious application can escalate granted permissions. The attack depends on a carelessly designed application which fails to protect the permissions granted to it. In this research, we propose a vulnerability checking system to check if an application can be potentially leveraged by an attacker to launch such privilege escalation attack. We downloaded 1038 applications from the wild and found 217 potentially vulnerable applications that need further inspection.**

I. Introduction

A recent research from Nielsen shows that Android now owns 29% market share of smartphone users in the US and is pulling ahead of RIM Blackberry (27%) and Apple iOS (27%) [1]. Moreover, the Android Market is the fastest growing mobile application platform. According to a recent report released by mobile security firm Lookout, the Android Market is growing at three times the rate of Apple's App store [2]. However, unlike the Apple's App store, there is no screening process of the apps being published on the Android Market. Occasionally, Google needs to take down some malicious apps from the Android Market after they were found to contain malware.

Android is basically a privilege-separated operating system [3]. Every application runs with a distinct system identity in its own Davik virtual machine. This mechanism isolates applications from each other and from the system. By default, an application has no right to perform any operations that would adversely impact other applications, the operating system, or the user. To acquire extra capabilities, an application needs to statically declare the permissions it requires, and the Android system prompts the user for consent at the time the application is installed.

However, a recent research points out that an application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee) [4]. Such attack is called privilege escalation attack. On success, it allows a malicious application to indirectly acquire more permissions through a benign application which fails to guard the permissions granted to it. To prevent this attack, applications must enforce additional checks on permissions to ensure that any applications accessing its components must be at least of the same level of privilege as it is. Since most of the application developers are not security experts, delegating the task to perform these checks to them is an error prone approach.

In this research, we propose a vulnerability checking system to detect benign applications which fail to enforce the aforementioned additional checks on permissions. We make use of the *AndroidManifest.xml* file which defines the permissions an application uses and the permissions other applications need in order to access the components of that application. Every Android application must include the *AndroidManifest.xml* in its Android package (APK) file as required by the Android system.

We implemented a prototype of the proposed system and scanned 1038 applications we downloaded from the wild. Among them, we detected 217 applications which do not enforce appropriate checking on permissions. All of them contain one or more components which are accessible by other applications and are security sensitive. For the remaining 821 applications, 497 of them are properly protected and 324 of them do not request security sensitive permissions. We conclude that a large portion of the applications are potentially vulnerable. Generally speaking, developers are not aware of the potential privilege escalation attack their applications are open to.

The rest of the paper is organized as follows. In Section 2, we give a brief overview of the cornerstones of the Android security. In Section 3, we present the privilege escalation attack on Android. The solution we propose is covered in Section 4 and the evaluation results are presented is section 5. We discuss limitations and future work in Section 6. Finally, we summarize related work in Section 7 and draw our conclusion in Section 8.

II. Cornerstones of Android Security

Android is a free, open source mobile platform based on the Linux kernel. Android applications are written in Java and composed of four types of application components: activities,

978-1-61284-307-0/11/$26.00 ©2011 IEEE

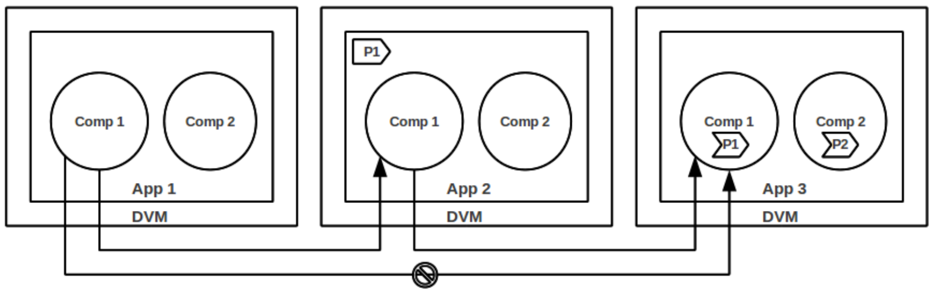

Fig. 1: Privilege Escalation Attack

services, content providers, and broadcast receivers. Components can communicate to each other and to components of other applications through an inter component communication (ICC) mechanism through intent messaging. An intent is a passive data structure holding an abstract description of an operation to be performed. To build performance-critical portions of the application in native code, developers can make use of the Android NDK companion tool which provides headers and libraries when programming in C or C++. However, inclusion of C or C++ libraries kicks away the security guarantees provided by the Java programming language. Several different vulnerabilities in native code of the JDK (Java Development Kit) have been identified [5]. We will discuss about the four cornerstones of Android security in the rest of this section.

A. Sandboxing

Android is a privilege-separated operating system. Each application runs within its own distinct system identity and its own Dalvik virtual machine (DVM). System files are owned by either the "system" or "root" user. As a result, an application can only access files it owns or files of other applications that are marked as readable / writable /executable for others explicitly. This provides a sandbox for each application which isolates it from other applications and from the system.

B. Application Signing

Each application must be signed with a certificate whose private key is held by its developer. The certificate is used solely for distinguishing application authors. It does not need to be signed by a certificate authority. Typically, it is a self-signed certificate. It is used by the Android system to decide whether to grant or deny application access to signature-level permissions and whether to grant or deny an application's request to be given the same Linux identity as another application. The certificate is included in its APK file such that the signature made by the developer can be validated at install time.

C. Permissions

Additional finer-grained security features are provided through a "permission" mechanism that enforces restrictions on the specific operations that a particular process can perform, and per-URI permissions for granting ad-hoc access to specific pieces of data. By default, an application has no permission to perform any operations that would adversely impact other applications, the operating system, or the user. To share resources and data with other applications, an application must declare the permissions it needs for additional capabilities not provided by the basic sandbox. The permissions an application requires is declared in the *AndroidManifest.xml* file which is compulsory for all applications. At install time, the Android system prompts the user for consent. It relies on the user to judge whether he or she permits the application to use all the permissions it requires or does not install the application.

D. Accessibility of Components

Application components can be specified as public or private. A public component can be accessed by other applications. However, it can still perform permission checking to restrict access to only applications that own certain permissions. On the other hand, a private component is only accessible by components within the same application.

III. Privilege Escalation Attack on Android

In this section, we give the details of the privilege escalation attack on Android. The attack was first proposed by Lucas Davi et al. in [4]. They stated the problem like following:

An application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee).

Figure 1 illustrates an example of privilege escalation attack on Android. In the figure, there are three applications running in their own DVMs. Application 1 owns no permissions. Since components in application 2 is not guarded by any permissions, they are accessible by components of any other applications. As a result, both components of application 1 can access components 1 in application 2. Application 2 own permission *P1*, Therefore, both components of application 2 can access component 1 of application 3 which is protected by permission *P1*.

We can see that component 1 of application 1 is accessing component 1 of application 2. However, since it does not have permission *P1*, it is not allowed to access component 1 of application 3. On the other hand, application 2 owns permission *P1*. Hence, component 1 of application 2 is allowed to access component 1 of application 3. Therefore, although component 1 of application 1 is not allowed to access component 1 of application 3, it can access it via component 1 of application 2. Therefore, the privilege of application 2 is escalated to application 1 in this case.

In order to prevent this attack, component 1 of application 2 should enforce that components accessing it must possess permission *P2*. This can be done at code level or by guarding component 1 by permission *P2*. However, this relies on application developers to perform the enforcement at the right places. This is an error prone approach as application developers are in general not security experts.

IV. Proposed Checking System

We propose a checking system which takes an Android application as input and check if the application has proper security checking such that applications with less permissions are restricted to access its component(s). We only perform security checking on activity and service components as they are the most commonly used components of an application that can be leveraged to perform privilege escalation attack. In the rest of this section, we give the design details of our proposed checking system. We first give an overview of the *AndroidManifest.xml* file. After that, we discuss how the checking process is done based on that file.

A. *AndroidManifest.xml*

All Android applications are required to include a manifest file called *AndroidManifest.xml* in their APK. The manifest presents essential information about the application to the Android system, information the system must have before it can run any of the application's code. Listing 1 shows an abstract of the general structure of the manifest file. We only focus on security related aspects of the manifest. The *uses-permission* tag requests a permission that the application must be granted in order for it to operate correctly. Permissions are granted by the user to the application at install time. The *permission* tag declares a security permission that can be used to limit access to specific components or features. The *android:permission* attribute of the application tag declares the permission that components of other applications must have in order to interact with the application. Moreover, each component can require extra permission for accessing it. They are declared in the *android:permission* attribute of the tag that declares the component. A component can also be made private by setting the *android:exported* attribute to *false*. Such a component is not accessible by components of other applications. If the *android:exported* attribute is absent, the default value of it depends on whether the component contains intent filters (except for Content Providers, which have the default value being *true*). If there is no intent filters, the default value is *false*. Otherwise, the default value is *true*. An intent filter specifies the types of intents that an activity, service, or broadcast receiver can respond to. Since, the filtering process only focuses on activity and service components, we do not give detailed description of the *provider* tag.

Listing 1: General structure of AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
    <uses-permission />
    <permission />
    <application android:permission="...">
        <activity android:permission="..."
                  android:exported="...">
            <intent-filter>
                ...
            </intent-filter>
        </activity>
        <service android:permission="..."
                 android:exported="...">
            <intent-filter>
                ...
            </intent-filter>
        </service>
        <receiver android:permission="..."
                  android:exported="...">
            <intent-filter>
                ...
            </intent-filter>
        </receiver>
        <provider android:permission="..."
                  android:readPermission="..."
                  android:writePermission="..."
                  android:exported="...">
            <grant-uri-permission />
        </provider>
    </application>
</manifest>
```

B. *The Checking Policy*

The checking system parses the *AndroidManifest.xml* to find out if:

1) The application uses any security sensitive permissions and;
2) There exists an activity or service component that does not require any permission and is publicly visible and security sensitive

If both of the them are true, the system rises an alarm that the application is potentially vulnerable.

We define security sensitive permissions as all permissions except those on the whitelist shown in table I. They are permissions that are not harmful to the security and privacy of the system. The list is subject to change to fine tune the detection of the system. The longer the list is, the more tolerant is the system.

We focus on activity and service components as they are the components that can be leveraged to launch a privilege escalation attack among the four types of components. The activity component provides a user interface and is what the user will see on the screen. On the other hand, the service component performs long-running operation in the background and does not provide a user interface. Both kinds of components can be protected by permissions.

As mentioned in the first part of this section, components with the *android:exported* attribute being set to false or components with no *android:exported* attribute and intent filters will not be accessible by components in other applications. Therefore, we do not need to care about such components and only focus on components that are publicly accessible.

We treat all publicly accessible and unprotected service components as security sensitive as we do not have any idea about what kind of operations they will perform from the manifest file. However, for activity components, there are standard actions that Android defines for launching activities. We can therefore infer the kinds of operations an activity component performs upon receiving an intent message and only focus on activity components that perform security sensitive operations to make the screening process more precise. We define security sensitive activity actions as all actions except those on a whitelist of actions that are not security sensitive. We do not include the whitelist here due to space limitation.

C. The Checking Process

The system starts with scanning the *uses-permission* tags to see if there is any permission required by the application that is not on the whitelist. If there is not any, the system terminates. Next, the system parses the *android:permission* attribute of the *application* tag to find out if the application is guarded by any permissions. If yes, the system concludes that the application is safe. If no, Otherwise, it goes on to perform the following checking for each activity or service component.

1) Activity components: Figure 2 shows the decision tree for the checking of activity components. First, the system checks if there is any permissions declared in the *android:permission* attribute of the tag that declares that component. If yes, the component is safe as it is protected by permission(s). If no, the system checks if the *android:exported* attribute is presence. If yes and the value of it is *true*, the system checks the intent filters of that component. If all the intent filters are security insensitive, the component passes the checking. Otherwise, the component is potentially vulnerable for privilege escalation and the system reports the name of the component for further inspection. If the *android:exported* attribute is absent and the visibility of the component hinges on the presence of intent filters. If there is no intent filters, the component is not visible to other applications by default, and hence, it is safe. Otherwise, the component is visible to other applications. In that case, the system also checks the intent filters of that component. If all the intent filters are security insensitive, the component passes the checking. Otherwise, the component is potentially vulnerable for privilege escalation and the system reports the name of the component for further inspection.

2) Service components: Figure 3 shows the decision tree for the checking of activity components. The checking of service components is similar to that of activity components except that it does not consider whether the intent filters are security sensitive or not since there is no standard intent action defined by Android for service components.

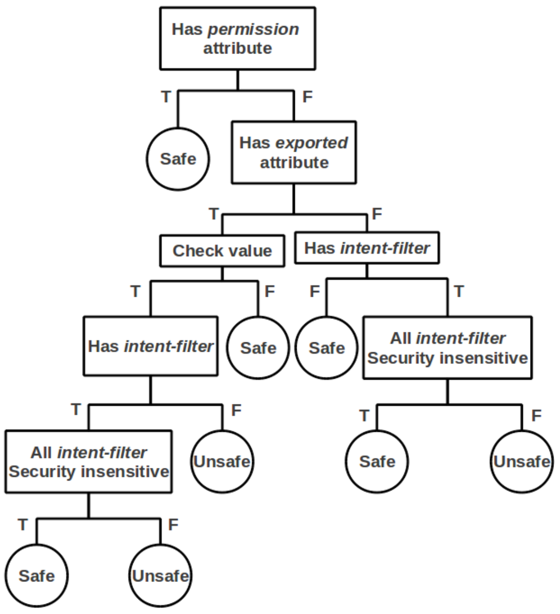

Fig. 2: The discision tree of the checking for activity components.

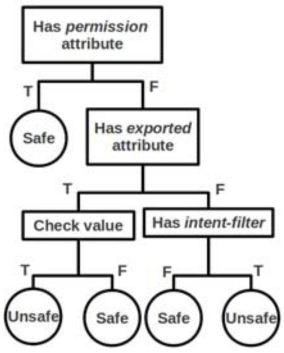

Fig. 3: The discision tree of the checking for service components.

V. Evaluation

We implemented a prototype of the checking system to evaluate our approach. The prototype consists of 200 lines of Java code. We make use of *AXMLPrinter2.jar* to parse the *AndroidManifest.xml* in the candidate APK file which is in binary XML format [6]. The entire checking system runs under Ubuntu 10.10.

We downloaded 1038 Android applications from the wild to evaluate our system. They are of different categories including SMS applications, web browsers, office applications, games, navigation applications, etc. Some of them are the most popular applications with the highest number of downloads. Our system reports that 821 of the applications are free from the privilege escalation attack. On the other hand, there are 217 applications that did not pass our check.

Although the above result is satisfactory, we still want to improve it. We picked 50 of them to further investigate the

TABLE I: Security non-sensitive permissions

Permission	Description
VIBRATE	Allows access to the vibrator
INTERNET	Allows applications to open network sockets
READ_HISTORY_BOOKMARKS	Allows an application to read the user's browsing history and bookmarks
WRITE_HISTORY_BOOKMARKS	Allows an application to write the user's browsing history and bookmarks
WRITE_EXTERNAL_STORAGE	Allows an application to write to external storage
RESTART_PACKAGES	Deprecated permission
WAKE_LOCK	Allows keeping processor from sleeping or screen from dimming
SET_WALLPAPER	Allows applications to set the wallpaper
SET_WALLPAPER_HINTS	Allows applications to set the wallpaper hints
BIND_WALLPAPER	Allow an application to bind to WallpaperService
CAMERA	Required to be able to access the camera device
CHECK_LICENSE	To check and make sure the user bought the app
RECEIVE_BOOT_COMPLETED	Allows an application to receive intent after the system finishes booting
DISABLE_KEYGUARD	Allows applications to disable the keyguard

reason why they did not pass the test. We found that 30 of them have intent filter(s) with custom intent action and our system could not anticipate what actions they would perform upon receiving such intents. 3 of them has an exported activity component with no intent filter. In that case, it can be started by any intent and the actions it will perform is unknown. 10 of them contain unprotected components whose intent filter(s) is for security sensitive intent action. 21 of them has publicly accessible service component(s) that is not protected by permission. In future work, we hope we can further narrow down the set of applications that do not pass the test by half. Our preliminary ideas will be covered in the next section.

VI. Discussion

In this section, we discuss the limitations of our approach. After that, we propose future work to improve it. Finally, we provide security guidelines for application developers that can help them protect their applications against privilege escalation attack.

A. Limitations

Since our checking system makes use of the manifest file only, there may be some missing information at the code level that can facilitate our checking. For example, an activity or service component may not perform any security sensitive operations even though it has the required privilege to do so. Moreover, it may have permission checkings at the code level before performing sensitive operations. Furthermore, an activity component may perform actions that are not defined in any of its intent filters upon receiving an explicit intent. We did not notice any applications with this behavior though. We leave code level checking for future work.

B. Future Work

The control flow grap of the application can be analyzed to check if there is a permission checking (by invoking the *checkPermission(Permission)* API call) before any security sensitive operations. A control flow graph is a graph representation of all paths that might be traversed during the execution of a program [7]. A node represents a code block which will be executed from the first line to the last line straightly. An edge represents a jump which can be an if-statement. For components which performs permission checkings before all security sensitive operations or for components which do not perform any security sensitive operations, we regard them as safe components.

C. Security Guidelines

To protect an application against privilege escalation attack, developers should avoid making the components of the application accessible by components in other applications. This can be done by either by not declaring any intent filters or by setting the *android:exported* attribute of the component to *false*. In case the component has to be made public, it should be protected by a permission such that other applications have to be privileged in order to access it. Besides, developers should be aware of the explicit intent messages sent to their applications. They should not rely on intent filters to protect a component since other applications can always send an explicit intent message to their components. Therefore, a publicly accessible component should not perform security sensitive operations upon receiving an explicit intent message.

VII. Related Work

The privilege escalation attack on Android was first proposed by Davi et al. [4] in which they demonstrated an example of the attack. They showed that a genuine application exploited at runtime or a malicious application can escalate granted permissions. However, they did not suggest any defense for the attack in the paper.

Before that, there were a few works on security extensions to Android security architecture. Saint [8] is a modification of Android to enable application providers to express the application security polices that regulate the interactions among them. It allows an application to control which applications can be granted the permissions it declares. Moreover, when an application needs to access an component of another application, both parties can assert controls of the communication between them through defining run-time interaction policies. In particular, the caller application selects which application's interfaces it uses and the callee application controls how its interface is used by other applications. Saint policy provides certain protection against privilege escalation attacks as the application can control which applications can access it. However, Saint assumes that access to components is implicitly allowed if no Saint policy exists. This put the burden of enforcing security to application developers which is error prone as most of them are not security experts.

Kirin [9] is an application certification service to mitigate malware at install time. It uses existing security requirements engineering techniques as a reference to identify dangerous application configurations in Android. The rules are a set of combinations of permissions that an application must not be granted at the same time. For example, an application being granted permissions to record audio and access location information may be an voice and location eavesdropping malware. Similar to our approach, their certification process relies on the manifest file in the APK of the application. However, their approach cannot identify applications vulnerable to privilege escalation attack. Instead, their work is orthogonal to our work and is targeting at a different kind of attack vector.

There are other works on security of the Andriod system. Schmidt et al. [10] walked through the smartphone malware evolution. They provided possible techniques for creating Android malware(s). Their approach involves usage of undocumented Android functions enabling them to execute native Linux application even on retail Android devices. They also showed that it is possible to bypass the Android permission system by using native Linux applications. Enck et al. [11] gives a description of the security model of the Android system. Jakobsson et al. [12] proposed a software-based attestation approach to detect any malware that executes or is activated by interrupts. Based on memory-printing of client devices, it makes it impossible for malware to hide in RAM without being detected. Nauman et. al. [13] improved the installation process of Android applications to allow user to selectively grant permissions to applications and impose constraints on the usage of resources. Shabtai et al. [14] makes use of Security-Enhanced Linux (SELinux) to help reduce potential damage on the Android system from a successful attack.

VIII. Conclusion

In this research, we addressed the privilege escalation attack with an application checking system. Through checking the security and permission configurations of the applications, we identify privileged components that are poorly protected by permissions. Although these components are genuine by themselves, their permissions can be leveraged by other applications which may be malicious. Evaluation of our system with 1038 applications downloaded from the wild showed that our system successfully identified 217 applications (21%) that are potentially vulnerable. This essentially forms the first line of defense against the privilege escalation attack by eliminating its attack vector. Future directions are identified to further improve the precision of the system.

Acknowledgment

The work described in this paper was partially supported by the General Research Fund from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. RGC GRF HKU 713009E), the NSFC/RGC Joint Research Scheme (Project No. N_HKU 722/09), and HKU Seed Fundings for Basic Research 200811159155 and 200911159149.

References

[1] Nielsen, "Who is winning the u.s. smartphone battle?" http://blog.nielsen.com/nielsenwire/online_mobile/who-is-winning-the-u-s-smartphone-battle/, March 2011.

[2] Lookout, "App genome report," https://www.mylookout.com/appgenome, February 2011.

[3] Android Open Source project, "Security and permissions," http://developer.android.com/guide/topics/security/security.html, April 2011.

[4] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Proceedings of the 13th international conference on Information security*, ser. ISC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 346–360. [Online]. Available: http://portal.acm.org/citation.cfm?id=1949317.1949356

[5] G. Tan and J. Croft, "An empirical security study of the native code in the jdk," in *Proceedings of the 17th conference on Security symposium*. Berkeley, CA, USA: USENIX Association, 2008, pp. 365–377. [Online]. Available: http://portal.acm.org/citation.cfm?id=1496711.1496736

[6] D. Skiba, "Axmlprinter2," http://code.google.com/p/android4me/, October 2008.

[7] B. A. Cota, D. G. Fritz, and R. G. Sargent, "Control flow graphs as a representation language," in *Proceedings of the 26th conference on Winter simulation*, ser. WSC '94. San Diego, CA, USA: Society for Computer Simulation International, 1994, pp. 555–559. [Online]. Available: http://portal.acm.org/citation.cfm?id=193201.194302

[8] M. Ongtang, S. Mclaughlin, W. Enck, and P. Mcdaniel, "Semantically rich application-centric security in android," in *In ACSAC 09: Annual Computer Security Applications Conference*, 2009.

[9] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 235–245. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653691

[10] A.-D. Schmidt, H.-G. Schmidt, L. Batyuk, J. Clausen, S. Camtepe, S. Albayrak, and C. Yildizli, "Smartphone malware evolution revisited: Android next target?" in *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, oct. 2009, pp. 1 –7.

[11] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *Security Privacy, IEEE*, vol. 7, no. 1, pp. 50 –57, jan.-feb. 2009.

[12] M. Jakobsson and K.-A. Johansson, "Retroactive detection of malware with applications to mobile platforms," in *Proceedings of the 5th USENIX conference on Hot topics in security*, ser. HotSec'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–13. [Online]. Available: http://portal.acm.org/citation.cfm?id=1924931.1924941

[13] 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

[14] A. Shabtai, Y. Fledel, and Y. Elovici, "Securing android-powered mobile devices using selinux," *Security Privacy, IEEE*, vol. 8, no. 3, pp. 36 –44, may-june 2010.