COMPUTERS & SECURITY 43 (2014) 205–218

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Permission based Android security: Issues and countermeasures

Zheran Fang [a,c], Weili Han [a,c,*], Yingjiu Li [b]

[a] Software School, Fudan University, Shanghai, 201203, China
[b] School of Information Systems, Singapore Management University, Singapore
[c] Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China

ARTICLE INFO

Article history:
Received 10 September 2013
Received in revised form
21 January 2014
Accepted 19 February 2014

Keywords:
Android security
Permission based security
Access control
Granularity of access control
Policy administration
Over-claim of permission
Permission escalation attack

ABSTRACT

Android security has been a hot spot recently in both academic research and public concerns due to numerous instances of security attacks and privacy leakage on Android platform. Android security has been built upon a permission based mechanism which restricts accesses of third-party Android applications to critical resources on an Android device. Such permission based mechanism is widely criticized for its coarse-grained control of application permissions and difficult management of permissions by developers, marketers, and end-users. In this paper, we investigate the arising issues in Android security, including coarse granularity of permissions, incompetent permission administration, insufficient permission documentation, over-claim of permissions, permission escalation attack, and TOCTOU (Time of Check to Time of Use) attack. We illustrate the relationships among these issues, and investigate the existing countermeasures to address these issues. In particular, we provide a systematic review on the development of these countermeasures, and compare them according to their technical features. Finally, we propose several methods to further mitigate the risk in Android security.

Introduction

Android security has been under a spotlight in information security as Android smartphones become the most popular mobile devices in the current market. Since the first Android-powered phone was delivered in October 2008 (Gozalvez, 2008), Android smartphones have grown to the largest global market share (75%) among all smartphones shipped in the first quarter of 2013 (IDC, 2013). In May 2013, Google announced that 900 million Android devices had been activated (Welch, 2013). According to F-Secure, a cyber security-related company, the number of new mobile threat families and variants continued to rise by 49% from the previous quarter; 91.3% of these threats targeted at Android devices in the first quarter of 2013 (F-Secure, 2013).

Android smartphones are protected by a permission based framework which restricts third-party applications' accesses to sensitive resources such as SMS database and external storage in Android smartphones. The accesses to sensitive resources may lead to money loss. For example, Android malware may send premium rate messages, make premium rate calls, and generate large amount of network data without users' acknowledgment. Moreover, the

* *Corresponding author*. Software School, Fudan University, Shanghai, 201203, China. Tel.: +86 (0)21 51355388.
E-mail addresses: 13212010002@fudan.edu.cn (Z. Fang), wlhan@fudan.edu.cn (W. Han), yjli@smu.edu.sg (Y. Li).
http://dx.doi.org/10.1016/j.cose.2014.02.007

accesses to sensitive resources may lead to leakage of users' private information stored in smartphones such as contacts, emails, and even credit card numbers. Third-party application developers can leverage various smartphone sensors such as GPS, cameras, and microphones, then create applications that do more than what they claimed so as to collect users' private information stealthily (Fragkaki et al., 2012). In the current Android permission framework, accesses to critical resources on smartphones are controlled according to permissions given to applications at install-time. That is, each application must request for certain permissions for it to access system resources on a smartphone at install-time and the user of the smartphone should make a decision on whether or not to grant the permissions requested.[1]

Such permission based framework is criticized as coarse-grained. Many applications tend to request much more permissions than necessary. In most cases, a user has to either grant all permissions an application requests or abort the installation process, instead of granting the permissions one by one. In addition, the permission based framework is vulnerable due to insufficient control of cooperation among applications and poor documentation on how to use various permissions.

Android security has attracted much attention from both academia and industry. To the best of our knowledge, papers related to Android security appeared as early as 2008 (Enck et al., 2008; Schmidt et al., 2008), which is the same year when the first Android-powered smartphone was delivered. Along with the explosive growth of Android-powered devices in the following years, a considerable number of research papers on Android security have been published.

Given the large number of published researches on Android security, especially on Android permission framework, we provide a systematic overview of the current state of Android security. In particular, we investigate the recent advancement on Android security, identify the issues in Android permission framework, and analyze the countermeasures to address the security issues.

The rest of this paper is organized as follows: Section 2 introduces the background of Android security. Section 3 classifies the issues in Android permission framework. Section 4 investigates existing solutions to address Android security issues. Section 5 discusses the future work. Finally, Section 6 concludes the paper.

Background of Android security

Android is proposed as a software stack for mobile devices. It consists of an operating system, an application framework, and core applications. Each Android application executes in a separate Dalvik virtual machine instance running as a unique user identity assigned at install-time. Thus applications are essentially isolated. This design provides promising security for file accesses and limits potential damage due to programming flaws such as buffer overflow (Enck et al., 2011).

Android restricts accesses to critical resources using permissions. A permission is simply a unique text string which can be defined by Android or third party developers. According to the documentation for Android developers, there are currently 130 permissions (Android, 2013b), which are defined in Android operating system, ranging from access to camera (CAMERA), full access to the Internet (INTERNET), dialing a phone number (CALL_PHONE), and even disabling the phone function permanently (BRICK). According to the study of Wei et al. (2012), the number of Android defined permissions keeps increasing since the first widely-used release (API level 3). The expansion of the permission set aims at not only providing finer-grained permissions but also controlling accesses to new hardware features (Wei et al., 2012). In addition to Android defined permissions, application developers can also declare customized permissions so as to protect their own critical resources.

Permissions may be required when an application is interacting with system resources, including calling system API functions, and reading from and writing to file systems. Granted permissions are assigned to an application's sandbox and inherited by all of the application's components, while required permissions are assigned to application components (Bugiel et al., 2011a). In the manifest file of an application, which is included in the application package, the application declares the permissions which it requires to achieve its functionality, as well as defines the permissions for protecting its own components and resources. A permission can be associated with one of the following four protection levels (Android, 2013a):

- *Normal*: A low-risk permission which allows applications to access API calls (*e.g.*, SET_WALLPAPER) causing no harm to users.
- *Dangerous*: A high-risk permission which allows applications to access potential harmful API calls (*e.g.*, READ_CONTACTS) such as leaking private user data or control over smartphone device.
- *Signature*: A permission which is granted if its requesting application is signed with the same certificate as the application which defines the permission is signed.
- *Signature-or-system*: A permission which is granted only if its requesting application is in the same Android system image or is signed with the same certificate as the application which defines the permission is signed.

At install-time, a user is shown with a list of permissions which an application requests. The user must either grant or deny all of these permissions together. After the user approves the permission request and installs the application, the application owns its permissions throughout its lifetime and it does not need to request them again at run-time. Android controls Inter-Component Communication (ICC) through a reference monitor. The reference monitor provides a *Mandatory Access Control (MAC)* enforcement on how applications access components by evaluating whether the applications are granted with necessary permissions.

[1] In the recent version of Android 4.3, users can revoke an application's permissions after the application is installed.

Analysis of Android security issues

Overview

We summarize the issues of Android security and illustrate their relationships in Fig. 1. We divide the issues into two categories: direct issues and indirect issues. Direct issues may lead to financial losses or leakage of user private information directly. On the other hand, the indirect issues can be used as stepping stones in launching attacks to Android smartphones.

As shown in Fig. 1, direct issues include over-claim of permissions, permission escalation attack and TOCTOU (Time of Check to Time of Use) attack, while the rest, including coarse granularity of permissions, incompetent permission administrators, and insufficient permission documentation, are indirect issues. The coarse granularity of permissions may lead to over-claim of permissions and make it more difficult for users to detect over-claim of permissions. Note that, the TOCTOU attack exists in Android due to naming collusion (Shin et al., 2010). In particular, even if a user un-installs a third party application after approving its defined permissions, the authorization given to the permissions is not revoked. After this, if a malicious application requires a permission which has the same name as one of the authorized permissions, then the malicious application can directly exercise such permission without approval.

Consider the issue of coarse-grained INTERNET permissions. One example is that a malicious developer may claim that the INTERNET permission is only used to display advertisements in a standalone game. While a user is tricked into trusting the legitimacy of the INTERNET request, the malicious developer can leverage the coarse-grained feature of INTERNET permission and access tolled websites secretly.

Incompetent permission administrators and insufficient permission documentation may result in accidental over-claim of permissions. Unconsciously over-claimed permissions assigned to a benign application can be exploited by a malicious application. For example, a malicious application can perform the *confused deputy attack* to exploit the permissions of a benign application (Dietz et al., 2011). More detail about the confused deputy attack is given in Section 3.6.

Fig. 1 shows that indirect issues lead to direct issues via either implicit relation or explicit relation. For a malicious developer to leverage the vulnerability of incompetent permission administrators, he or she must perform an attack exploiting the vulnerability left by an incompetent permission administrator in an application. For example, an incompetent administrator may carelessly approve the installation of an application having access to critical resources. Given this vulnerability, the permission escalation attack can be launched by malicious applications. In addition, for a malicious developer to launch a TOCTOU attack against a user, the attacker must trick the user into installing a malicious application first. On the other hand, coarse granularity of permissions, incompetent permission administrators and insufficient permission documentation may lead to over-claim of permissions without any intermediate process.

Coarse granularity of permissions

Although Android defines 130 permissions, most of them are of coarse granularity. Especially, the INTERNET permission (Barrera et al., 2010), the READ_PHONE_STATE permission (Pearce et al., 2012), and the WRITE_SETTINGS permission (Jeon et al., 2012) are coarse-grained as they give an application arbitrary accesses to certain resources. Taking the INTERNET permission as an example, the INTERNET permission allows an application to send HTTP(S) requests to all domains, and connect to arbitrary destinations and ports (Felt et al., 2011c). As a result, the INTERNET permission provides insufficient expressiveness to enforce control over the Internet accesses of the application (Barrera et al., 2010). According to Barrera et al.'s study in 2010, 62.36% of application samples downloaded from Google Play Store use this permission (Barrera et al., 2010). However, Felt et al. discovered that about 36% of the applications they reviewed use the INTERNET permission for making HTTP(S) requests to specific domains only. These Android applications rely on remote servers for getting content, much like web applications. Moreover, about 7% applications use the INTERNET permission to support Google AdSense which displays advertisements from a single domain in a WebView (Felt et al., 2011c). This indicates that many applications would tolerate a restrictive INTERNET permission which allows applications to access nothing in Internet but a specific list of domains.

While the INTERNET permission is necessary for many applications such as networked games, the usage of this permission cannot be restricted or controlled by users. As a result, a malicious application may camouflage itself as a legitimate application which indeed requires Internet accesses, while misusing its Internet accesses without users' acknowledgment.

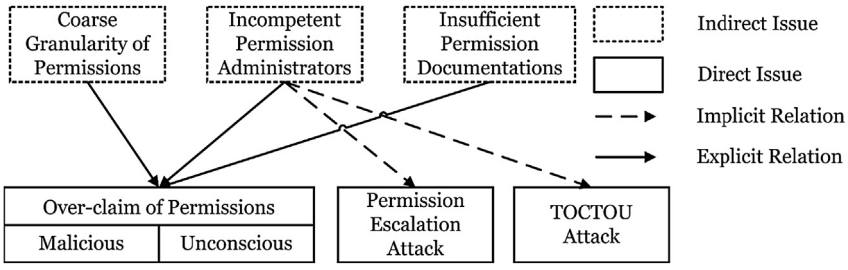

Fig. 1 – Relationship among issues in Android permission mechanism.

Incompetent permission administrators

Several roles, including developers and end-users, are involved in the process of granting application permissions. Developers write manifest files to request permissions for applications, while end-users approve the requests. After approval, applications may run with the granted permissions. In addition, application marketers may verify applications. They may abort the process of permission authorization if applications are put off from markets.

Unfortunately, both developers and end-users usually lack professional knowledge. In addition, the developers and end-users may have conflict of interest (Han et al., 2013). When a developer writes a manifest file requesting permissions for his or her application, the developer may not know in detail what is at risk for end-users if the application is granted with these permissions. While some enthusiastic developers might take time to learn what each of the 130 permissions does and request them appropriately, other developers might choose to simply over-claim permissions so as to make sure their applications work any way (Barrera et al., 2010).

As for end-users, the survey performed by Felt et al. (Felt et al., 2012) shows that only 3% of Internet survey respondents correctly answered all three permission comprehension questions and 24% of laboratory study participants demonstrated competent but imperfect comprehension.

Insufficient permission documentation

Google Inc. provides a great deal of documentation for Android application developers, but the content on how to use permissions on Android platform is limited (Vidas et al., 2011). As investigated by Felt et al., the lack of permission usage information may lead to developers' errors. In Android 2.2 documentation, permission requirements are provided for 78 methods; however, Felt et al.'s test reveals permission requirements for 1259 methods, which is a sixteen-fold improvement over the documentation. The documentation lists additional permissions in several class descriptions, but it is not clear which methods of the classes require the stated permissions. Moreover, six errors are identified in Android permission's documentation. The insufficient and imprecise permission information confuses Android application developers, who may write applications with guesses, assumptions and repeated tries. Consequently, this leads to defective applications which become threats with respect to security and privacy of Android users (Felt et al., 2011b).

Furthermore, the content of permissions is usually too technical for end-users to understand. Taking `INTERNET` permission as an example, when an end-user reads the permission description *FULL INTERNET ACCESS: Allows an application to create network sockets* (Android, 2011), he or she often feels that this description is too complex and abstruse. The user might not know what risk he or she would face when approving the permission request.

Over-claim of permissions

Over-claim of permissions is probably the most severe threat to Android security. It directly breaks the principle of least privilege (PLP) (Saltzer, 1974). This violation of PLP exposes users to potential privacy leakage and financial losses. For example, if a standalone game application requests for the `SEND_SMS` permission which is unnecessary, the permission can be exploited to send premium rate messages without users' acknowledgment.

Felt et al. identified that 56% of the over-privileged applications have only one extra unnecessary permission and 94% have 4 or fewer extra permissions. The low degree of over-claim of permissions per-application indicates that developers attempt to add correct permissions rather than arbitrarily request for a larger number of unnecessary permissions. Developers may make wrong decisions because of several reasons, concluded by Felt et al. (Felt et al., 2011b), including: at first, developers tend to request for permissions with names that look relevant to the functionalities they design, even if the permissions are not actually required; second, developers may request for permissions which should be requested by deputy applications instead of their own applications; finally, developers may make mistakes due to using copy and paste, deprecated permissions, and testing artifacts.

As shown in Fig. 1, the issue of over-claim of permissions can be categorized into: malicious and unconscious. Three

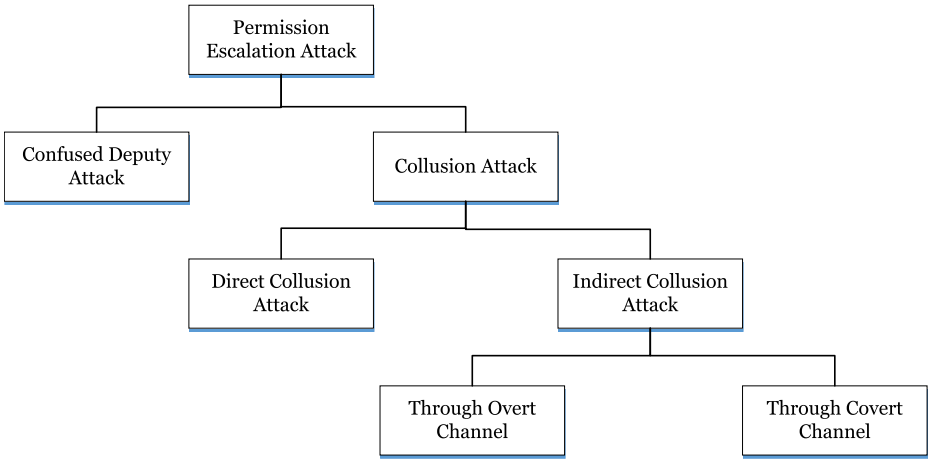

Fig. 2 – Categorization of the permission escalation attack.

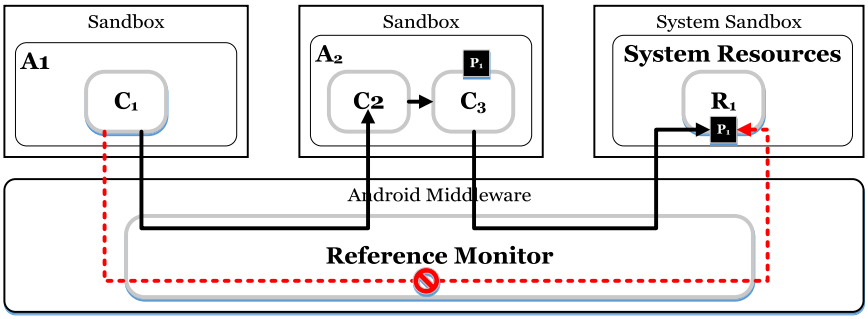

Fig. 3 – Permission escalation attack (Bugiel et al., 2011a).

other issues, including coarse granularity of permissions, incompetent permission administrators, and insufficient permission documentation, are drivers of over-claim of permissions.

Permission escalation attack

In contrary to the general belief that the damage imposed by an Android malware is limited to an application's sandbox, the permission escalation attack allow a malicious application to collaborate with other applications so as to access critical resources without requesting for corresponding permissions explicitly (Bugiel et al., 2011a; Felt et al., 2011a; Marforio et al., 2012).

Fig. 2 shows the categorization of the permission escalation attack. The permission escalation attack can be classified into two categories: *confused deputy attack* and *collusion attack*.

The *confused deputy attack* exploits the vulnerabilities in unprotected interfaces of privileged benign applications (Dietz et al., 2011). As shown in Fig. 3, the application A_1 is not granted with the permission P_1; C_1, which is a component of A_1, cannot directly access system resource R_1 protected by permission P_1. However, C_1, can access R_1 transitively if application *A2* is granted with permission *P1* and one of *A2* 's component, *C2* does not require any permission to be accessed. As a result, *C1* can access R_1 through *C2* and *C2* (Bugiel et al., 2011a).

In addition, the *collusion attack* can be carried out by multiple applications in generating a joint set of permissions which enables them to perform an unauthorized or malicious actions (Schlegel et al., 2011). The *collusion attack* can be further classified by the way applications communicate with each other, into *direct collusion attack*, where applications communicate directly, and *indirect collusion attack* where applications communicate via a third application or component in between (Bugiel et al., 2011a).

The *indirect collusion attack* usually involves another application or component as a mediation which can provide either *overt channels* or *covert channels* (Marforio et al., 2012). *Overt channels*, such as buffers, files and I/O devices, use a data object as the entity to hold certain information. In other words, the entity is an object that is normally viewed as a data container (Kemmerer, 2002). Other examples of *overt channels* include shared preferences, system logs, and UNIX socket communication (Marforio et al., 2012).

In contrast, *covert channels* use entities which are normally not intended to be used for communication (Kemmerer, 2002; Marforio et al., 2011). It is possible to use *covert channels* to bypass the Android's middleware layer (Bugiel et al., 2011a). Examples of *covert channels* include file locks (Schlegel et al., 2011), change of screen state, change of vibration settings (Bugiel et al., 2011a), type of intents, threads enumeration (Marforio et al., 2012), *etc.* Marforio et al. measured the throughput of a number of overt and covert channels. The measurement shows that even *covert channels* with low throughput are still sufficient to exchange private information (Marforio et al., 2012).

TOCTOU attack

The vulnerability of TOCTOU (Time of Check to Time of Use) exists in Android mainly due to naming collusion. No naming rule or constraint is applied to a new permission declaration (Shin et al., 2010). Moreover, permissions in Android are represented as strings, and any two permissions with the same name string are treated as equivalent even if they belong to unrelated applications (Fragkaki et al., 2012). Malicious application developers may exploit this flaw. Suppose a malicious developer manages to trick a user into installing malicious application *A* which declares permission *P'*, and another malicious application *B* which requests permission *P'*. The name of permission *P'* is the same as permission *P* which protects accesses to a critical resource. Afterward the user uninstalls application *A* and installs benign application *C* which declares permission *P*. Now the malicious application *B* would be able to use permission *P'* to access the critical resource. According to Shin et al., this TOCTOU flaw exists in Android 1.5, 1.6, 2.0 and 2.1, on both emulators and actual devices. In addition, Fragkaki et al. claimed to have reproduced it in Android 2.3.7 (Fragkaki et al., 2012). Unfortunately, this vulnerability cannot be thwarted using protection levels in the current Android permission system.

Enhanced designs and implementations

Countermeasures to over-claim of permissions

Overview

Generally, the effort of coping with the issue of over-claim of permissions is two-fold: (i) detection and analysis of over-

claim of permissions in Android applications (Felt et al., 2011c, 2011b; Wei et al., 2012), and (ii) proposals of enhanced frameworks which allow users to revoke over-claimed permissions at install time or run-time (Nauman et al., 2010; Beresford et al., 2011; Hornyack et al., 2011). Since advertising is a key revenue for free Android applications, the advertising libraries contained in Android applications usually require INTERNET, ACCESS_NETWORK_STATE, and READ_PHONE_STATE permissions (Pearce et al., 2012). Therefore, there also exists a category of countermeasures specific to the over-claim of permissions introduced by the advertising libraries contained in Android applications (Pearce et al., 2012; Shekhar et al., 2012; Leontiadis et al., 2012).

Finer-grained permission models introduced in Section 4.2 is an indirect countermeasure to combat the issue of over-claim of permissions.

Countermeasures

Detection of over-claim of permissions. As for detecting over-claim of permissions in Android applications, Felt et al. manually reviewed the top free and top paid applications from 18 Google Play categories in 2011 (Felt et al., 2011c). For each of the applications, Felt et al. compared its functionalities with the permissions it requested by exercising its user interface. Four out of 36 applications were over-privileged, while the INTERNET permission accounted for three of the over-privileged applications.

Felt et al. built *Stowaway*, an automatic tool to detect over-claim of permissions in compiled applications (Felt et al., 2011b). *Stowaway* analyzed a set of 940 applications and identified that about one-third of these applications have unnecessary permissions. The most common unnecessary permissions include ACCESS_NETWORK_STATE, READ_PHONE_STATE, ACCESS_WIFI_STATE, WRITE_EXTERNAL_STORAGE, and CALL_PHONE. Such unnecessary permissions can be leveraged by malicious applications.

In 2012, Wei et al. applied *Stowaway* to a set of 237 evolving third party applications covering 1703 versions. The result showed that the overall tendency was towards over-claim of permission (Wei et al., 2012). In particular, 19.6% of updated versions of applications were over-privileged due to added permissions, and 25.2% of applications were initially over-privileged and stayed over-privileged during their evolutions. On the other hand, 11.6% of applications resorted from over-privileged to legitimate.

Au et al. built *PScout*, a static analysis tool which captures permission requirements for every API call by examining the entire Android source code (Au et al., 2012). After comparing the permission lists produced by *PScout* with those specified in the manifest files, Au et al. discovered that 543 out of 1260 applications required at least one over-claimed permission.

Enhanced system implementations. In 2010, Nauman et al. proposed *Android Permission Extension (Apex)* (Nauman et al., 2010), a policy enforcement framework which allows users to selectively grant permissions to an application using a simple and easy-to-use interface provided by *Poly*, an augmented application installer. When a user installs a new application, he or she may grant or deny permissions one by one. With *Apex*, even after an application has been installed, a user is able to grant more permissions or revoke some of the granted permissions.

In 2011, *MockDroid*, proposed by Beresford et al., allows users to revoke access permissions to particular resources at run-time (Beresford et al., 2011). For the revoked permissions, fake or "mock" data are provided to applications which call the corresponding functions. For example, a user might choose to provide a fake constant value when an application tries to retrieve the device ID. Other categories of data which could be "mocked" include location, Internet, SMS/MMS, calendar, contact and broadcast intents.

Hornyack et al. (2011) proposed *AppFence*, which covertly substitutes shadow data to replace private data. Similar to *MockDroid*, when an application attempts to access critical resources such as device ID and location, *AppFence* provides fake data instead. In addition, *AppFence* blocks network transmission of the data which the user makes available to the application for on-device use only by extending the *TaintDroid* information-flow tracking system.

Zhou et al. developed *TISSA* which defines a *private mode* for Android based smart phones (Zhou et al., 2011). The *private mode* enables users to flexibly control what kinds of personal information, *e.g.*, device ID, contracts, call log, and location, are accessible to an application. Moreover, the granted permissions can be dynamically adjusted at run-time so as to better meet user's requirements. For each kind of personal information, *TISSA* supports four options: *empty*, *anonymized*, *bogus*, and *trusted* to protect the information. The *empty* option simply returns an *empty* result to a requesting application, indicating "non-presence" of the requested information. The *anonymized* option provides an *anonymized* version from original personal information, which still allows applications to proceed without necessarily leaking user's information. The *bogus* option provides a fake result for requested information. Finally, the *trusted* option returns the original personal information as requested.

Similar to *Apex*, in 2012, Mueller et al. proposed *Flex-P*, which revises the Android permission system for managing application permissions at run-time (Mueller and Butler, 2011). *Flex-P* allows end-users to grant a subset of permissions at install-time and change the granted permissions at any time even after installation.

Separating advertising libraries. *AdDroid* by Pearce et al. (2012), *AdSplit* by Shekhar et al. (2012), and the work by Leontiadis et al. (2012) are countermeasures to thwart the over-claim of permissions introduced due to using advertising libraries contained in Android applications. Since they work similarly, *AdDroid* will be used as an example in the following analysis.

In 2012, Pearce et al. conducted a study of Google Play Store and discovered that 49% of Android applications contained at least one advertising library and that application developers over-claimed about 46% of advertising-supported applications (23% of all applications) by one or more permissions due to the use of advertising libraries. Further, 56% of applications with advertisements requested for location information (34% of all applications) solely due to the requirement of advertisements (Pearce et al., 2012). To address this issue, *AdDroid* introduces a new advertising API as well as certain advertising permissions for Android platform so that privileged advertising

functionalities can be separated from host applications, allowing applications to show advertisements without requesting for privacy-sensitive permissions.

Comparison

Table 1 summarizes the comparison among countermeasures to over-claim of permissions.

While Felt et al. built *Stowaway* tool to detect the over-claim of permissions in Android applications and provided a quantitative analysis for over-claim of permissions, Wei et al.'s work was based on the same tool but focused on the tendency of over-claim of permissions in the evolution of Android platform and third party applications. In contrast to *Stowaway* which fuzzes Android APIs directly, *PScout* fuzzes the applications which use the APIs (Au et al., 2012). As a result, *PScout* is more complete but less sound than *Stowaway*. However, *Stowaway* and *PScout* are both incomplete because they cannot catch those APIs invoked through Java reflection. In addition, *PScout* works on any version of Android while *Stowaway*'s specification is for Android 2.2 (Au et al., 2012).

When a permission to access a critical resource is revoked by user, *Apex* throws an exception but *MockDroid* provides fake data to requesting applications. As a result, applications are more likely to crash under *Apex*, compared to *MockDroid*. *Flex-P* enables users to grant not only individual permissions but also group permissions, which is an added functionality to *Apex*. However, unlike *Apex*, *Flex-P* lacks the functionality of adding new restriction types to granted permissions, such as limiting a permission to a certain number of uses per day. Compared to *MockDroid*, *AppFence* also protects the data which users makes available to applications for on-device use only from being misappropriated and sent off the device. While Beresford et al. tested whether or not fake data can be provided to applications without causing them to crash, Hornyack et al. measured user-discernable side effects as well (Hornyack et al., 2011).

The difference among *MockDroid*, *AppFence*, *TISSA* and *Apex*, *Flex-P* is that the latter two are restricted to the currently available permissions. In contrast, the privacy setting of *MockDroid*, *AppFence*, and *TISSA* is orthogonal to the currently available Android permissions (Zhou et al., 2011).

Although *AdDroid* (Pearce et al., 2012), *AdSplit* (Shekhar et al., 2012), and Leontiadis et al.'s work (Leontiadis et al., 2012) share similar ideas in combatting the over-claim of permissions through separating advertising libraries from host applications, there are a few subtle differences among them. *AdSplit* runs advertising libraries as separate applications, and allows applications to share the screen of phone. Leontiadis et al.'s work leverages in-application widgets and ICC instead of screen sharing in running advertising libraries (Pearce et al., 2012).

Table 1 – Comparison among the countermeasures to over-claim of permissions.

	Important feature	Working layer
Stowaway[a]	Detect over-claim of permission	N/A[j]
Wei et al. (2012)	Show tendency of over-claim of permission	N/A[j]
PScout[b]	Detect over-claim of permission	N/A[j]
Apex[c]	Allow revoking of over-claimed permissions	Middleware[k]
MockDroid[d]	Allow revoking of over-claimed permissions	Middleware
AppFence[e]	Allow revoking of over-claimed permissions	Middleware
TISSA[f]	Allow revoking of over-claimed permissions	Middleware
Flex-P[g]	Allow revoking of over-claimed permissions	Middleware
AdDroid[h]	Separate advertising functionality	Middleware
AdSplit[i]	Separate advertising functionality	Middleware
Leontiadis et al. (2012)	Separate advertising functionality	Middleware

[a] Felt et al. (2011c), and Felt et al. (2011b).
[b] Au et al. (2012).
[c] Nauman et al. (2010).
[d] Beresford et al. (2011).
[e] Hornyack et al. (2011).
[f] Zhou et al. (2011).
[g] Mueller and Butler (2011).
[h] Pearce et al. (2012).
[i] Shekhar et al. (2012).
[j] *Stowaway*, Wei et al. (2012), and *PScout* are static analysis tools which do not run on Android OS.
[k] Countermeasures working in the middleware layer usually consist of modifications in the application installer, the reference monitor, the permission database, and the Dalvik virtual machine.

Finer-grained permissions

Overview

Finer-grained permissions can be used to address the issues of coarse-grained permissions. Finer-grained permission implementations can be categorized into install-time policy enforcement and run-time policy enforcement. The install-time policy enforcement aims to stop malicious applications from being installed on user's devices. It provides finer policy enforcement at install-time rather than asking users to decide whether to grant all permissions or nothing (Enck et al., 2008; Ongtang et al., 2011). More factors, such as permission combinations, permission and action string combinations, and signatures of requesting applications, are taken into consideration. If an application violates one of the preset policies, the installation of the application would be aborted. On the other hand, the run-time policy enforcement allows users to set finer-grained restrictions at run-time (Ongtang et al., 2011; Nauman et al., 2010). The finer-grained restrictions can be enforced on the permission configuration of an installed application, such as how many times a critical resource is allowed to be accessed, and what kinds of private information can be accessed.

Finer-grained permissions can also be categorized according to the type of improvement over coarse-grained permissions, including revising the current frameworks and proposing new frameworks. The first category evaluates how and where data are accessed or transferred complying with finer-grained policies (Nauman et al., 2010). This category usually involves the enhancement of the reference monitor module in the Android middleware. The second category provides new Android permission models instead of the

current one (Jeon et al., 2012). A user can specify finer-grained policies in new permission models.

Countermeasures

Install-time policy enforcement. In 2008, Enck et al. proposed *Kirin* (Enck et al., 2008) and in 2009, Enck et al. enhanced *Kirin* (Enck et al., 2009). *Kirin* performs policy enforcement at install-time using a set of predefined security rules. These rules decide whether the permission configuration and action strings listed in an application package's manifest file are secure. *Kirin* not only evaluates an application on a single permission basis, but also takes both permission combinations and, permission and action string combinations into consideration. For example, an eavesdropper on a voice call requires a combination of `READ_PHONE_STATE`, `RECORD_AUDIO` and `INTERNET` permissions to function appropriately. A voice call eavesdropper can be prevented from being installed due to enforcing the following security rule in *Kirin* "*An application must not have the* `PHONE_STATE`*,* `RECORD_AUDIO`*, and* `INTERNET` *permission labels*."

Similar to *Kirin*, Ongtang et al. proposed *Secure Application INTeraction (Saint)* in 2009 whose install-time policies regulate the granting of application defined permissions (Ongtang et al., 2011). In addition to Android's protection level-based permission granting policy, an application can define the conditions under which the permissions defined by the application can be granted to other applications at install-time. An application might require a *signature-based policy* to control how the permissions it declares are granted based on the signature of a requesting application. An application might also require a *configuration-based policy* to control the permission assignments based on the configuration parameters of a requesting application such as application versions. At install time, *Saint*-enhanced installer retrieves the requesting permissions from the manifest file in an application package. For each permission, it queries an *AppPolicy provider*, which maintains a database of all the policies. The *AppPolicy provider* consults its policy database, and returns a decision according to matching rules. If the matched policy conditions pass, the installation proceeds. Otherwise, it is aborted. Upon successful installation, the application's polices are appended in the *AppPolicy provider*'s policy database.

Run-time policy enforcement. In addition to install-time policy enforcement, *Saint* by Ongtang et al. also enforces run-time policies which regulate the communications between applications (Ongtang et al., 2011). With a *configuration-based policy*, an application can define desirable configurations of opponent applications, such as the minimum version and a set of permissions which an opponent application is allowed or disallowed. Moreover, an application may wish to regulate its interactions based on the transient state of phone. A *phone context-based policy* is proposed to govern run-time interactions based on context, such as location, time, Bluetooth connection and connected devices, call state, data state, data connection network, and battery level. At run-time, when a caller application initiates ICC through Android middleware framework, ICC is intercepted by *Saint* policy enforcement code before any Android permission evaluation. *Saint* queries *AppPolicy provider* for policies which match ICC. The *AppPolicy provider* identifies the appropriate policies, evaluates the policy conditions (application state, phone configuration, and *etc.*), and returns a decision. If the conditions are not satisfied, ICC will be blocked; otherwise, ICC will be directed to existing Android permission evaluation enforcement. Then, Android allows or disallows ICC to continue based on traditional Android policies.

In addition to allowing a user to selectively grant permissions, *Apex*, which was first introduced in Section 4.1, allows a user to impose quantitative run-time constraints on the usage of resources, *e.g.*, limiting the number of SMS sent each day (Nauman et al., 2010). At install-time, for each permission, besides the options to grant or deny permissions, *Apex* introduces a third option, conditional allow. The user can specify constraints on permissions, such as the number of times a permission can be used, or the valid time per day during which a permissions should be allowed. Moreover, after install-time, the user might modify the constraints with a shortcut in the settings application. These kinds of constraints are saved in an XML file, along with user's policies of selective permission administration described in Section 4.1.

Aurasium proposed by Xu et al. in 2012 (Xu et al., 2012) is an application hardening service that bypasses the need to modify Android system while enhancing Android's security and privacy controls. *Aurasium* enforces security and privacy policies to an application by repackaging to attach sandboxing codes to the application. For example, after downloading an Android application from an unknown source, a user can put the application into the *Aurasium* black box and obtain a hardened version. *Aurasium* ensures that the interactions of the harden application are closely monitored for malicious activities, and policies protecting the user's privacy and security are actively enforced. *Aurasium* intercepts almost all types of interactions between the application and the operating system. For instance, when an application attempts to access a remote site on the Internet, the IP of the remote server is evaluated against an IP blacklist. When an application attempts to send an SMS message, *Aurasium* evaluates whether the number is a premium number. When an application tries to access private information such as IMEI, IMSI, stored SMS messages, contact information, or services such as camera, voice recorder, or GPS, a policy evaluation is performed to allow or disallow the access. *Aurasium* also monitors I/O operations such as write and read (Xu et al., 2012).

In 2012, Jeon et al. proposed an approach which consists of three tools, *RefineDroid*, *Mr. Hide*, and *Dr. Android* (Jeon et al., 2012). Jeon et al. created a taxonomy with four broad categories that groups standard Android permissions by the behaviors the permissions allowed. For each category, they proposed new fine-grained variants. Joen et al. chose to focus on the `INTERNET` permission, which is pervasive across applications and might be particularly dangerous. Joen et al. developed a fine-grained, whitelisting permission `InternetURL(d)`, which allowed network connections only to domain *d* and its subdomains. To validate the taxonomy, *RefineDroid*, a static analysis tool that infers the fine-grained permission usage in existing applications, was built. The second tool, *Mr. Hide* is a set of Android services that wrap several privileged Android APIs and dynamically enforce a specified set of fine-grained permissions from the taxonomy.

Table 2 – Comparison among finer-grained permission implementations.

	Important feature	Enforcement phase	Require system modification	Working layer
Kirin[a]	Perform finer-grained install-time policy enforcement	Install-time	✓	Middleware
Saint[b]	Perform finer-grained install-time and run-time policy enforcement	Install-time & Run-time	✓	Middleware
Apex[c]	Allow run-time resource usage constraint	Run-time	✓	Middleware
Aurasium[d]	Perform finer-grained policy enforcement by repackaging applications	Run-time	✗	Application[g]
RefineDroid, Mr. Hide, and Dr. Android[e]	Provide new permission system	Run-time	✗	Application
CRePe[f]	Allow finer-grained context-related policy enforcement	Run-time	✓	Middleware

[a] Enck et al. (2008), and Enck et al. (2009).
[b] Ongtang et al. (2011).
[c] Nauman et al. (2010).
[d] Xu et al. (2012).
[e] Jeon et al. (2012).
[f] Conti et al. (2011).
[g] Countermeasures working in the application layer usually involve modifying existing application packages and developing new applications which run as normal Android applications.

Finally, *Dr. Android* is a tool that removes Android permissions in existing application package files without source code and replaces the permissions with a specified set of fine-grained versions that are accessed through *Mr. Hide*. A user can employ *Dr. Android* to retrofit a downloaded application with fine-grained permissions in order to enforce a desired security policy by rewriting the application. During testing, almost all activities of applications written by *Dr. Android* functioned normally, with no observable changes. However, the performance slowdown imposed by *Mr. Hide* was significant, since the interprocess communication required by *Mr. Hide* was quite an expensive operation.

Finer-grained context-related permission models. In 2011, *CRePE* by Conti et al. allows users to set a context-related policy enforcement, where the *context* here can be geological location, time, and temperature (Conti et al., 2011). Suppose in a business scenario, a company wants to restrict a set of applications that can run on smart phones provided by the company to its employees during work activities. The context *during-work* is defined as the locations of the company's buildings and the time of working hours. Policy information only allows a restricted set of applications to run. The context and policy information are stored in *PolicyProvider*, similar to *Saint*'s *AppPolicy provider*. When in the context, *PolicyManager* evaluates if an application which is about to run is in the application set. If it is in the application set, the application can run successfully. Otherwise the application is not allowed to run.

Comparison

Table 2 summarizes the comparison among the finer-grained permission implementations.

The enforcement of install-time policies, such as *Kirin*, is restricted as it relies on application package metadata or the manifest file. Therefore, dynamically created Broadcast Receivers, which are not specified in the manifest file, cannot be put under control (Ongtang et al., 2011). Both *Kirin* and *Saint* enforce install-time policies in a fine grain; a difference is that only *Saint* allows install-time policies to be defined based on signatures and versions. In *Saint*, any install-time policies are defined by applications, while the source of *Kirin* policies is mainly the authors of *Kirin*. On the other hand, the run-time policies in *Saint* are defined by applications, while the source of the *Apex* policies is mainly the end-users.

The run-time monitors which *Aurasium* inserted into an application execute in the same process as the application, and hence are potentially subject to circumvention (Jeon et al., 2012). *Dr. Android* and *Mr. Hide* enhance Android security by removing the original permissions from an application and perform fine-grained permission evaluating in the *Mr. Hide* service, which runs in a separate process. *Saint*, *Apex*, *Aurasium*, and *CRePE* all require the modification of Android system or rooting devices. *Dr. Android* and *Mr. Hide* can run on unmodified Android devices, but require the modification of applications, which makes the process of installing a new application much more complex (Jeon et al., 2012).

One difference between the countermeasures in Section 4.1 and the countermeasures in this section is that: the countermeasures in Section 4.1 are made based on available Android permissions or specific types of data, while the countermeasures in this section provide fine-grained control for users. In other words, the control discussed in this section can be made based on permissions, signatures, action strings, or even new permission frameworks.

Countermeasures to permission escalation attack

Overview

The countermeasures to the permission escalation attack usually involve tracking and controlling the information flows through ICC between applications (Enck et al., 2010; Dietz et al., 2011; Jia et al., 2013). If the ICC violates one of the predefined policies, the ICC is stopped or restricted. In addition,

various permission escalation attacks are highlighted in (Schlegel et al., 2011; Davi et al., 2011) to raise users' concerns.

Countermeasures

Demonstration of permission escalation attack. In 2011, Schlegel et al. presented *Soundcomber*, a Trojan with few and innocuous permissions that can extract targeted private information from the audio sensor of a phone (Schlegel et al., 2011). *Soundcomber* is a typical example of malicious applications which can exploit covert channels in Android systems. Davi et al. demonstrated how to send text messages to a telephone number without explicitly requiring a permission (Davi et al., 2011).

In 2012, Grace et al. developed a tool called *Woodpecker* which identifies the vulnerabilities of the permission escalation attack in eight popular Android smart phones (Grace et al., 2012). They discovered that the stock Android system did not properly enforce the permission model. Specifically, several permissions that protect accesses to sensitive resources were unsafely exposed to other applications, which did not need to request these permissions for actual usages.

Defense mechanisms. Along with presenting *Soundcomber*, Schlegel et al. proposed a defense mechanism (Schlegel et al., 2011), which maintains a list of critical numbers and blocks all applications from accessing the audio data during a sensitive phone call.

In 2010, Enck et al. proposed *TaintDroid*, which tracks information flows by labeling (tainting) data from privacy-sensitive sources and transitively applying labels as sensitive data propagated through program variables, files, and interprocess messages (Enck et al., 2010). If any tainted data aim to leave the system at a taint sink (*e.g.*, network interface), a user is alerted about the application leaking the tainted data.

QUIRE, proposed by Dietz et al. in 2011, is a lightweight provenance system (Dietz et al., 2011). *QUIRE* transparently tracks and records the specific ICC call chain so that the recipient can observe the full call chain associated with a request. Moreover, *QUIRE* also extends to the network module in the Linux kernel to analyze remote procedure calls. Since the ICC recipient is aware of the initiator and even the entire call chain, *QUIRE* can prevent ICC calls from untrusted agents and fraudulent requests. To defend against the *confused deputy attack*, *QUIRE* would deny an ICC request if the originating application has not been granted with the corresponding permission explicitly.

Similar to *QUIRE*, the *IPC Inspection* proposed by Felt et al. tracks the information flows through ICC. When an application receives a message from another application with less privileges, *IPC Inspection* mitigates the *confused deputy attack* by reducing the permissions of the application to the intersection between the recipient's permissions and the requester's permissions (Felt et al., 2011a). In addition to proposing *IPC Inspection*, Felt et al. discovered several severe attacks against Android's system applications and demonstrated that many pre-installed applications were vulnerable to the *confused deputy attack*.

XManDroid (eXtended Monitoring on Android) by Bugiel et al. tracks and analyzes the communication links among applications at run-time, and ensures that the application comply to a desired policy (Bugiel et al., 2011a). *XManDroid* inspects data transferred over ICC and makes policy decisions based on the content of intents. Moreover, *XManDroid* policies may request user confirmation in order to allow or deny specified ICC calls. *XManDroid* is invoked when the default Android reference monitor grants an ICC call, and verifies whether the requested ICC call complies to predefined security policies, *e.g.*, "An application that is notified about incoming or outgoing calls and can record audio must not communicate to an application with network access". To mitigate the *collusion attack* over overt and covert channels, Bugiel et al. extended the `ActivityManager` of Android system which detects and registers all installed services and content providers in the system. *XManDroid* tags each row in the databases of system content providers with the UIDs of writers. Upon writing data, these tags are updated. Upon reading, *XManDroid* verifies for each row if the corresponding reader–writer pairs constitutes a policy violation. Similarly to system content providers, Bugiel et al. tagged each value of the system services with the UID of writers. Upon reading from the services, *XManDroid* performs a policy evaluation if the reading will establish a policy violating channel.

In two works of Bugiel et al. (2011b) and Bugiel et al. (2012), Bugiel et al. extended the *XManDroid* framework with a kernel-level module by adapting and tweaking TOMOYO Linux. With the additional kernel module, Bugiel et al. implemented a mandatory access control on the file system (including UNIX domain sockets) and local Internet sockets. To support the run-time dynamic low-level policy enforcement, a callback channel between the kernel and the middleware was also provided.

In 2012, Fragkaki designed and implemented *Sorbet*, which allows developers to define policies to mitigate undesired information flows and *confused deputy attack*. *Sorbet* tracks the permissions of all components on a call stack. When a component *A* is called, and *A* is protected by the permission *P*, *Sorbet* evaluates if every component on the call stack has *P* (Fragkaki et al., 2012).

Marforio et al. explored possible overt and covert channels on Android and measured the throughput of each channel (Marforio et al., 2012). The analysis showed that even covert channels with low throughput were still sufficient to exchange possibly private information. Moreover, Marforio et al. proposed four different techniques to remove the taint from tainted variables. Marforio et al. also evaluated the effectiveness of *TaintDroid* and *XManDroid*. In the evaluation, *TaintDroid* was able to correctly report the transmission of sensitive data through two out of four overt channels. As was expected, the covert channels were undetected by *TaintDroid*. On the other hand, *XManDroid* detected all the overt channels except the *system log* channel. It also successfully detected or blocked the *type of intents*, *UNIX socket discovery*, *reading* `/proc/stat` and *threads enumeration* covert channels. However, a small subset of covert channels were not detected by *XManDroid* such as *free space on filesystem*, *processor frequency*. The evaluation also showed that *XManDroid* suffered from the limitation of false-positive results when two non-malicious applications tried to share legitimate data. The communication in *XManDroid* was even blocked even if non-sensitive data was shared.

Comparison

The defense mechanism proposed by Schlegel et al. was specific to *Soundcomber*. That is, this mechanism only worked for colluding malwares which extracted information from phone calls (Schlegel et al., 2011).

Table 3 summarizes the comparison among *TaintDroid*, *QUIRE*, *IPC Inspection*, *XManDroid*, and *Sorbet*. As shown in Table 3, *TaintDroid*, *QUIRE*, *IPC Inspection*, *Sorbet*, and *XManDroid* all addressed the *confused deputy attack* through tracking the information flow between applications, but none of them fully addressed the *collusion attack*. Although *XManDroid* extended the `ActivityManager` of Android system which detects and registers all installed services and content providers in the system specially for mitigating the *collusion attack* over overt and covert channels, there were still one overt channel and a small subset of covert channels not detected by *XManDroid* according to Marforio et al.'s evaluation (Marforio et al., 2012). While *Sorbet* shares many similarities with *QUIRE* and *IPC Inspection*, *Sorbet*'s novelty is to allow developers to specify policies on a per-application basis, which is not showed in Table 3.

Facilitating permission administration

Overview

Three roles are usually involved in the Android permission administration: developers declare which permissions the application will request; application marketers verify whether the application is legitimate or not by an automatic tool or manual review; users decide whether to approve the permission requests. These three roles are usually performed by those who are not well-trained in policy based management (Han et al., 2013). To facilitate the permission administration of Android, researchers have proposed several methods and tools to assist application developers (Vidas et al., 2011; Sarma et al., 2012; Han et al., 2013), application marketers (Google, 2012), and application users (Sarma et al., 2012; Han et al., 2013).

Table 3 – Comparison among mitigating solutions to permission escalation attack.

	Effects			Working layer
	Address confused deputy attack	Address collusion attack		
		Overt channel	Covert channel	
TaintDroid[a]	✓	Partially	✗	Middleware
QUIRE[b]	✓	✗	✗	Middleware & Kernel[f]
IPC Inspection[c]	✓	✗	✗	Middleware
XManDroid[d]	✓	Partially	Partially	Middleware & Kernel
Sorbet[e]	✓	✗	✗	Middleware & Kernel

[a] Enck et al. (2010).
[b] Dietz et al. (2011).
[c] Felt et al. (2011a).
[d] Bugiel et al. (2011a, 2011b, 2012).
[e] Fragkaki et al. (2012).
[f] Countermeasures working in the kernel layer usually involve modifications in the Linux kernel.

Countermeasures

In 2011, Vidas et al. built an Eclipse IDE plugin, *Permission Check Tool*, which assists developers in specifying a minimum set of permissions required for a given Android application (Vidas et al., 2011). *Permission Check Tool* analyzes application source code and automatically informs the developer on the minimum set of permissions required to run the application properly.

In 2012, Google announced that a service named *Bouncer* had been deployed on Google Play Store (Google, 2012). *Bouncer* scans Google Play automatically for potentially malicious applications without disrupting the user experience of Google Play Store or requiring developers to go through an application approval process.

Sarma et al. proposed to better inform policy administrators whether the risks of installing an application was commensurate with its expected benefit (Sarma et al., 2012). Specially, Sarma et al. proposed to capture the benefit of an application by using the category and sub-category of the application, and capture the risk by using the usage percentage of the permissions among the applications in the same category. When a user sees a warning triggered by an application, he or she should be more cautious about the risk of the application being installed. When a developer sees a warning triggered by his or her application, he or she should consider how to make the application avoid triggering the signal.

In 2013, to simplify the tasks of permission administration for both developers and users, Han et al. proposed the *Collaborative Policy Administration (CPA)* framework (Han et al., 2013). The essential idea of *CPA* is that applications with similar functionalities shall have similar policies. The similarity measure methods are designed to judge the similarity of different applications and retrieving similar policy sets from a policy bas. These similarity measure methods may be choosing applications belonging to the same category on Google Play Store, or performing text mining on the applications' description. Then end-users can verify or specify the permission configuration of an Android application by leveraging *CPA*.

Comparison

The ideas proposed by Sarma et al. (2012) and Han et al.'s (2013) CPA are similar in that they both took the functionality of an application into consideration, along with the permissions which the application requested, to better inform policy administrators. In addition, they both proposed to judge the functionality of an application by its category. However, Han et al. also proposed to perform text mining on the application's description to find similar applications and the evaluation showed that the latter approach had a better performance than the first one which is based on category.

Future work

In this section, we identify promising directions to improve the permission framework on Android. We point out that

there are many important issues to address in this field, including designing more efficient detection methods for permission escalation attack, identifying more vulnerabilities on current permission framework, and enforcing mandatory access control policies (Smalley and Craig, 2013; Bugiel et al., 2013).

Data driven methods to strengthen Android framework

A meaningful direction is to collect and analyze a large volume of security-relevant data on Android usage such as descriptions, implementations (APK files), usage log, reviews and ratings of Android applications. Collecting and analyzing such data can help researchers discover intrinsic trends and patterns relevant to Android security, including the ecosystem of malicious applications. Researchers can leverage such data to propose novel methods for strengthening the Android framework. Currently, such data are used to discover the characteristics of malicious codes, and detect malicious or unsuitable applications according to permission configurations (Han et al., 2013) and implementation settings (Bugiel et al., 2011a). Also, such data can be used to evaluate role mining algorithms, and boost up the development of role based access control mechanisms (Zhang et al., 2013).

It would be challenging to collect a large volume of permission related data on Android. Such data can be collected from application markets (*e.g.*, descriptions and implementations) and social network services such as Twitter and Facebook (*e.g.*, reviews, usage log, characteristics of applications' users, and relationships among these users). For example, the characteristics of application users and relationships among these users can be leveraged to enforce the following principle more accurately: *Similar users may configure similar permissions for similar applications*. This can help unskillful permission administrators design and verify permission configurations of Android applications.

Consistency between application intentions and system implementation

Many researchers have identified the problem of inconsistency among descriptions, permission configurations, and implemented functions. Leveraging this inconsistency, an adversary can trap end-users into installing a malicious application (Vennon, 2010) in a way similar to the phishing attack on the Web. For example, a standalone game may request for INTERNET permission; this could be useless in its implementation, or it can be used as an advertisement functionality without prompt; the standalone game may secretly implement a paid functionality via accessing some tolled websites. Researchers may propose effective tools which can remove such redundant permissions, prompt advertisement usage, and warn potential malicious usage.

We argue that it is also important to verify the consistency among descriptions, permission configurations, and implemented functions. It is also important to create permission configuration or brief description of an application based on its implementation. The biggest challenge to achieve this is to understand the dynamic features in the implementation of an Android application.

In addition, the development in policy management for refining high level policies, such as goal policies or utility policies, can help developers of Android applications maintain consistency between application intentions and system implementation.

Flexible and fine-grained permission models

The current model of Android permissions is coarse-grained and inflexible. It is difficult to specify the context of permissions. For example, INTERNET permission is required by most of applications (Barrera et al., 2010); however, it is not possible to specify the permission in varied context of application usage. As a result, an end-user is unlikely to care about the prompt of permission requests, because he or she has encountered too many such requests (Sarma et al., 2012). Furthermore, traditional access control models are usually linked to different objects, such as *reading* a *file* in the disk. However, the current permission model on Android is specified with respect to all objects in the same category.

We argue that some context-aware models (Conti et al., 2011; Chen et al., 2013) of Android permissions can help developers make more accurate permission requests for their applications. The content of the context should cover some usage environment variables and attributes of objects. Furthermore, it would be meaningful to employ standard access control policy languages (Han and Lei, 2012) such as XACML to express Android permissions. Even though this would lead to performance degradation, the benefit on portability and flexibility due to the use of such languages may overweight the performance loss.

Conclusion

The security issues and countermeasures of Android systems have been rigorously studied since the first Android device was shipped to the market. In recent years, the problem of Android security has become even more severe, partially due to the vulnerabilities in the design of Android systems, and partially due to the huge success of Android devices in the market. Motivated to provide a systematic overview of the current research on Android security, we identify six issues in Android security, including coarse granularity of permissions, incompetent permission administrators, insufficient permission documentation, over-claim of permissions, permission escalation attack, and TOCTOU attack. The former three are indirect issues, while the latter three are direct issues, which may lead to financial loss or privacy leakage directly.

In this paper, we investigated the countermeasures to address the issues in Android security. In particular, we showed the development of these countermeasures and compared them systematically according to the technical features of these countermeasures.

Although the situation of Android security is severe, we identified plenty of opportunities to improve Android security, especially when more security relevant data such as permission configurations and secure system implementations are publicly available. Based on comprehensive analysis on the issues and countermeasures, we argued that Android security

can be improved by developing data-driven tools to strengthen Android framework, and maintaining consistency between application intentions and implementations.

Acknowledgment

This paper is supported by the projects of National Key Science and Technology Program (2012ZX01039-004-20), Natural Science Foundation of Shanghai (12ZR1402600), 12th Five-Year National Development Foundation for Cryptography (MMJJ201301008), Innovation Foundation of STCSM (Grant No. 12511504200), and the open projects funded by CNNIC DNSLab and Key Lab of Information Network Security, Ministry of Public Security (C13612). We thank Miss Xinyi Zhang for her English polish. We also thank anonymous reviewers for their comments. Weili Han is the corresponding author.

REFERENCES

Android. Gmail – android market https://web.archive.org/web/20110328201807/https://market.android.com/details?id=com.google.android.gm; 2011.

Android http://developer.android.com/guide/topics/manifest/permission-element.html; 2013a.

Android. Manifest.permission http://developer.android.com/reference/android/Manifest.permission.html; 2013b.

Au KWY, Zhou YF, Huang Z, Lie D. Pscout: analyzing the android permission specification. In: Proc. of ACM CCS. ACM; 2012. pp. 217–28.

Barrera D, Kayacik HG, van Oorschot PC, Somayaji A. A methodology for empirical analysis of permission-based security models and its application to android. In: Proc. of ACM CCS. ACM; 2010. pp. 73–84.

Beresford AR, Rice A, Skehin N, Sohan R. Mockdroid: trading privacy for application functionality on smartphones. In: Proc. of HotMobile. ACM; 2011. pp. 49–54.

Bugiel S, Davi L, Dmitrienko A, Fischer T, Sadeghi A. XManDroid: a new Android evolution to mitigate privilege escalation attacks. Technische Universität Darmstadt; 2011a [Technical Report; Technical Report TR-2011-04].

Bugiel S, Davi L, Dmitrienko A, Fischer T, Sadeghi A, Shastry B. Towards taming privilege-escalation attacks on android. In: Proc. of NDSS; 2012.

Bugiel S, Davi L, Dmitrienko A, Fischer T, Sadeghi AR, Shastry B. Poster: the quest for security against privilege escalation attacks on android. In: Proc. of ACM CCS. ACM; 2011b. pp. 741–4.

Bugiel S, Heuser S, Sadeghi AR. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In: Usenix security; 2013.

Chen KZ, Johnson NM, D'Silva V, Dai S, MacNamara K, Magrino T, et al. Contextual policy enforcement in android applications with permission event graphs. In: NDSS; 2013.

Conti M, Nguyen V, Crispo B. Crepe: context-related policy enforcement for android. Inf Secur; 2011:331–45.

Davi L, Dmitrienko A, Sadeghi A, Winandy M. Privilege escalation attacks on android. Inf Secur; 2011:346–60.

Dietz M, Shekhar S, Pisetsky Y, Shu A, Wallach D. Quire: lightweight provenance for smart phone operating systems. In: Proc. of USENIX security; 2011.

Enck W, Gilbert P, Chun BG, Cox LP, Jung J, McDaniel P, et al. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proc. of USENIX OSDI; 2010. pp. 1–6.

Enck W, Octeau D, McDaniel P, Chaudhuri S. A study of android application security. In: Proc. of USENIX security, 2011; 2011.

Enck W, Ongtang M, McDaniel P. Mitigating android software misuse before it happens; 2008.

Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification. In: Proc. of ACM CCS. ACM; 2009. pp. 235–45.

F-Secure. Mobile threat report January–March 2013 http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q1_2013.pdf; 2013.

Felt A, Wang H, Moshchuk A, Hanna S, Chin E. Permission re-delegation: attacks and defenses. In: Proc. of USENIX security; 2011. pp. 22–37.

Felt AP, Chin E, Hanna S, Song D, Wagner D. Android permissions demystified. In: Proc. of ACM CCS. ACM; 2011b. pp. 627–38.

Felt AP, Greenwood K, Wagner D. The effectiveness of application permissions. In: Proc. of USENIX WebApps. USENIX Association; 2011c. p. 7.

Felt AP, Ha E, Egelman S, Haney A, Chin E, Wagner D. Android permissions: user attention, comprehension, and behavior. In: Proc. of SOUPS. ACM; 2012. p. 3.

Fragkaki E, Bauer L, Jia L, Swasey D. Modeling and enhancing android's permission system. In: Computer Security–ESORICS 2012. Springer; 2012. pp. 1–18.

Google. Android and security http://googlemobile.blogspot.com/2012/02/android-and-security.html; 2012.

Gozalvez J. First googles android phone launched. IEEE Veh Technol Mag 2008;3(4):3–9.

Grace M, Zhou Y, Wang Z, Jiang X. Systematic detection of capability leaks in stock android smartphones. In: Proc. of NDSS; 2012.

Han W, Fang Z, Yang LT, Pan G, Wu Z. Collaborative policy administration. IEEE TPDS 2013;24(1):1.

Han W, Lei C. A survey on policy languages in network and security management. Comput Networks 2012;56(1):477–89.

Hornyack P, Han S, Jung J, Schechter S, Wetherall D. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: Proc. of ACM CCS. ACM; 2011. pp. 639–52.

IDC. Android and ios combine for 92.3operating system shipments in the first quarter while windows phone leapfrogs blackberry, according to idc http://www.idc.com/getdoc.jsp?containerId=prUS24108913; 2013.

Jeon J, Micinski KK, Vaughan JA, Fogel A, Reddy N, Foster JS, et al. Dr. android and mr. hide: fine-grained permissions in android applications. In: Proc. of ACM SPSM. ACM; 2012. pp. 3–14.

Jia L, Aljuraidan J, Fragkaki E, Bauer L, Stroucken M, Fukushima K, et al. Run-time enforcement of information-flow properties on android – (extended abstract). In: ESORICS; 2013. pp. 775–92.

Kemmerer RA. A practical approach to identifying storage and timing channels: twenty years later. In: Proc. of ACSAC. IEEE; 2002. pp. 109–18.

Leontiadis I, Efstratiou C, Picone M, Mascolo C. Don't kill my ads!: balancing privacy in an ad-supported mobile application market. In: Proc. of HotMobile. ACM; 2012. p. 2.

Marforio C, Francillon A, Capkun S. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. ETH Zurich; 2011 [Technical Report; Technical Report 724].

Marforio C, Ritzdorf H, Francillon A, Capkun S. Analysis of the communication between colluding applications on modern smartphones. In: Proc. of ACSAC. ACM; 2012. pp. 51–60.

Mueller K, Butler K. Poster: Flex-p: flexible android permissions. In: Proc. of IEEE S&P; 2011.

Nauman M, Khan S, Zhang X. Apex: extending android permission model and enforcement with user-defined runtime constraints. In: Proc. of ACM ASIACCS. ACM; 2010. pp. 328–32.
Ongtang M, McLaughlin S, Enck W, McDaniel P. Semantically rich application-centric security in android. Secur Commun Networks 2011;5(6):658–73.
Pearce P, Felt AP, Nunez G, Wagner D. Addroid: privilege separation for applications and advertisers in android. In: Proc. of ACM ASIACCS. ACM; 2012. pp. 71–2.
Saltzer JH. Protection and the control of information sharing in multics. Commun ACM 1974;17(7):388–402.
Sarma BP, Li N, Gates C, Potharaju R, Nita-Rotaru C, Molloy I. Android permissions: a perspective combining risks and benefits. In: Proc. of ACM SACMAT. ACM; 2012. pp. 13–22.
Schlegel R, Zhang K, Zhou X, Intwala M, Kapadia A, Wang X. Soundcomber: a stealthy and context-aware sound trojan for smartphones. In: Proc. of NDSS; 2011. pp. 17–33.
Schmidt A, Schmidt H, Clausen J, Yuksel K, Kiraz O, Camtepe A, et al. Enhancing security of linux-based android devices. In: Proc. of 15th International Linux Kongress. Lehmann; 2008.
Shekhar S, Dietz M, Wallach DS. Adsplit: separating smartphone advertising from applications. CoRR 2012;28 [abs/12024030].
Shin W, Kwak S, Kiyomoto S, Fukushima K, Tanaka T. A small but non-negligible flaw in the android permission scheme. In: IEEE POLICY. IEEE; 2010. pp. 107–10.
Smalley S, Craig R. Security enhanced (se) android: Bringing flexible mac to android. In: NDSS; 2013.
Vennon T. Android malware. a study of known and potential malware threats; February 2010. p. 24 [Online].
Vidas T, Christin N, Cranor L. Curbing android permission creep. In: Proc. of the Web, vol 2; 2011.
Wei X, Gomez L, Neamtiu I, Faloutsos M. Permission evolution in the android ecosystem. In: Proc. of ACSAC. ACM; 2012. pp. 31–40.
Welch C. Google: 900 million android activations to date, 48 billion app installs. The Verge; 2013.
Xu R, Saïdi H, Anderson R. Aurasium: practical policy enforcement for android applications. In: Proc. of USENIX security. USENIX Association; 2012. p. 27.
Zhang X, Han W, Fang Z, Yin Y, Mustafa H. Role mining algorithm evaluation and improvement in large volume android applications. In: Proceedings of the first international workshop on Security in embedded systems and smartphones; 2013. pp. 19–26.
Zhou Y, Zhang X, Jiang X, Freeh VW. Taming information-stealing smartphone applications (on android). In: Proc. of TRUST. Springer; 2011. pp. 93–107.

Zheran Fang is a graduate student with Software School at Fudan University now. He is currently a member of the Laboratory of Cryptography and Information Security, Software School, Fudan University. His research interests mainly include information security, policy based management.

Weili Han is an associate professor with Software School, Fudan University. His research interests are mainly in the fields of policy based management, IoT security. He received his Ph.D. of Computer Science and Technology at Zhejiang University in 2003. Then, he joined the faculty of Software School at Fudan University. From 2008 to 2009, he visited Purdue University as a visiting scholar funded by China Scholarship Council and Purdue University. Weili Han has served in several leading conferences and journals as PC members and reviewers. Weili Han is a member of ACM, IEEE and CCF.

Yingjiu Li is currently an Associate Professor in the School of Information Systems at Singapore Management University. He received his Ph.D. degree in Information Technology from George Mason University in 2003. His research interests include RFID security and privacy, applied cryptography and system security, and data applications security. He has published over 100 technical papers in international conferences and journals. Yingjiu Li is a senior member of the ACM and a member of the IEEE Computer Society.