SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1179

RESEARCH ARTICLE

PaddyFrog: systematically detecting confused deputy vulnerability in Android applications

Jianliang Wu[1], Tingting Cui[1], Tao Ban[2], Shanqing Guo[1]* and Lizhen Cui[1]

[1] Shandong University, Shunhua Road, Ji'nan, China
[2] National Institute of Information and Communications Technology, Tokyo, Japan

ABSTRACT

An enormous number of applications have been developed for Android in recent years, making it one of the most popular mobile operating systems. However, it is obvious that more vulnerabilities would appear along with the booming amounts of applications. Poorly designed applications may contain security vulnerabilities that can dramatically undermine users' security and privacy. In this paper, we studied a kind of recently reported application vulnerability named confused deputy – a specific type of privilege escalation vulnerability, which can result in unauthorized operations, and so on. We proposed a novel system with code-level static analysis to analyze the applications and automatically detect possible confused deputy vulnerabilities. To tackle analysis challenges imposed by Android's component-based programming paradigm, we employed special control flow graph construction techniques to build call relations among components and function call graph within components. We developed a prototype of this system named PaddyFrog and evaluated with 7190 real world Android applications from two of the most popular markets in China. We found 1240 applications with confused deputy vulnerability and proved to be exploitable. The median execution time of this system on an application is 14.4s, which is fast enough to be used in volumes of applications testing scenarios. Copyright © 2015 John Wiley & Sons, Ltd.

KEYWORDS
static analysis; confused deputy attack; privilege escalation attack; control flow graph

***Correspondence**
Shanqing Guo, Shandong University, Shunhua Road, Ji'nan, China.
E-mail: guoshanqing@sdu.edu.cn

1. INTRODUCTION

A recent research from Nielsen[†] shows that Android now owns 51.8% market share of smartphone users in the US, which is pulling ahead of Apple iOS (34.3%) and RIM Blackberry (8.1%) [1]. At the same time, Google Play[‡] is becoming 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 store [2]. Unfortunately, the increasing adoption of Android brings up the growing prevalence of security issues [3].

The core security mechanisms in Android are application sandboxing, application signing, and the permission framework, which limits the access to the sensitive data (e.g., SMS and contacts), resources (e.g., battery and log files) and system interfaces (e.g., Internet connection and GPS). Once granted by the (end) users, the assigned permissions cannot be changed afterwards, and they are checked by Android's reference monitor at runtime. This approach restricts the potential damage imposed by malicious applications.

Although the Android's security mechanisms fail most attacks, however, there are flaws which impair the security of Android within the permission framework. Confused deputy [4] is a specific type of privilege escalation, which allows a malicious application to acquire more permissions indirectly through benign applications resulting in unauthorized read or write operations on sensitive resources. It is possible to carry out this kind of attack [5] on Android because of the serious defect of the Android security framework that an application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee). Here, we propose a confused deputy vulnerability detecting system to test whether an application is exploitable to confused deputy attack.

[†] http://www.nielsen.com/global/en.html.
[‡] https://play.google.com/store.
[§] https://www.lookout.com/.

A similar system has been presented in [6], however, without code-level detecting, the system only checks the AndroidManifest file that provides indispensable information about the application, which leads to a higher false positive ratio. Our contributions in this paper are as follows: (1) we have designed and implemented a novel code-level tool, PaddyFrog, to detect the confused deputy vulnerability in Android applications. PaddyFrog detects the confused deputy vulnerability based on the AndroidManifest file and the Control Flow Graph (CFG), which describes the invoking relationship among components and function call flow within components, which overcomes the limitations of [6]; (2) we ran PaddyFrog on 7190 applications downloaded from two of the most popular alternative markets in China (HiApk [7] and Anzhi Market [8]). PaddyFrog detected at least 1240 applications that have this vulnerability; (3) we presented two case studies to illustrate why our method can reach lower false positive rate than past works.

The rest of the paper is organized as follows. In Section 2, we give a brief overview of the fundamental of the Android security framework and explain confused deputy attack on Android. The solution we propose is covered in Section 3, and the evaluation results are presented in Section 4. We discuss the limitations of our work in Section 5. Finally, we summarize related work in Section 6 and draw our conclusions in Section 7.

2. PRELIMINARIES AND CONFUSED DEPUTY ATTACK

2.1. Android architecture

Android [9] can be described as a software stack including an operating system, middleware, and key applications. The main part of Android is middleware, which runs on the top of the Linux Kernel and the hardware drivers. It comprises of libraries, runtime environment, and application framework, which offers necessary components and services to the applications. Lastly, at the top of the stack are the applications, which access the underlying functionality through the Application Program Interfaces (APIs) and communicate with one another via its unique lightweight Inter-Process Communication (IPC) mechanism.

2.2. Android application framework

Android's application model [9] is highly flexible. An application is composed of one or more components, which may be invoked as a separate process or under an existing process in the same application. Android defines four types of components:

Activities provide user interfaces. Activities are started via Intents, and they can return data to their invoking components upon completion. All visible portions of applications are Activities.

Services run in the background and do not interact with the user. Downloading a file and decompressing an archive are examples of operations that may take place in a Service. Other components can bind to a Service, which lets the binder invoke methods that are declared in the target Service's interface. Intents are used to start and bind to Services.

Broadcast Receivers receive Intents sent to multiple applications. Receivers are triggered by the reception of appropriate Intents and then run in the background to handle the event. Receivers are typically short-lived and often relay messages to Activities or Services. There are three types of broadcast Intents: normal, sticky, and ordered. Normal broadcasts are sent to all registered Receivers at once, then they disappear. Ordered broadcasts are delivered to one Receiver at a time and any Receiver in the delivery chain of an ordered broadcast can stop its propagation. Broadcast Receivers have the ability to set their priority level for receiving ordered broadcasts. Sticky broadcasts remain accessible after they have been delivered and rebroadcastable to future Receivers.

Content Providers are databases addressable by their application-defined uniform resource identifier (URIs). They are used for both persistent internal data storage and as a mechanism for sharing information between applications.

Components within the same or from different applications can interact through the Binder IPC mechanism. Intents provide a general pattern for components to interact with one another.

2.3. Android security architecture

2.3.1. Sandboxing.

Sandboxing is a mechanism to isolate applications from each other and system resources. Application isolation is carried out by means of assigning a unique user identifier to each application, while the underlying Linux kernel enforces discretionary access control to resources (files and devices) by user ownership. System resources are owned by either system or root. Applications can only access their own files or files of others that are explicitly declared as world-wide readable.

2.3.2. Application signing.

Android enforces application signing, however, not centrally, that is, developers themselves have to sign the application with the self-certified key. Thus, application signing does not provide protection against malware but helps to establish trust relationships among applications originating from the same developer. Applications signed with the same key may request to share the same user identifier, and they will be placed into the same sandbox.

2.3.3. Android permission framework.

Android permission framework is provided by the middleware layer. It includes a reference monitor which enforces Mandatory Access Control on inter-component communication (ICC) calls. Security sensitive APIs, here referred to as interfaces, are protected by permissions. These are security labels, which can either be required to enforce access control or be granted to allow access.

Granted permissions are assigned to application sandboxes and inherited by all application components. Unlike this, required permissions are assigned to application components. Both required and granted permissions are explicitly specified in an application's AndroidManifest file, which is included in application installation package. Granted permissions are approved at installation time based on user confirmation. Once granted, permissions cannot be modified. At runtime, the reference monitor checks permission assignments.

2.4. Confused deputy and privilege escalation attack on android

In this section, we give the details of the confused deputy – a specific type of privilege escalation attack in Android. The confused deputy problem could occur on several platforms, Linux, Windows, as well as the web. The root cause of it is that the software does not sufficiently preserve the original source of the request before forwarding the request to an external actor that is outside of the software's control sphere [10]. Concentrating on Android, it's a little bit different from traditional platforms. On Android, the fundamental of the confused deputy attack is the exposed component, which performs the privileged task.

In the confused deputy's scenario, the deputy defines some exposed interfaces. A malicious *requester* lacks the permission that the deputy has to perform a malicious operation. The requester invokes the deputy's exposed interfaces, causing the deputy to invoke a privileged system API call. The system API call will be executed because the deputy has the requested permission to perform the system API call. In this way, the requester succeeded to invoke a privileged system API call indirectly without appropriate permissions.

As the example shown in Figure 1, the requester does not have the permission to send text messages while the

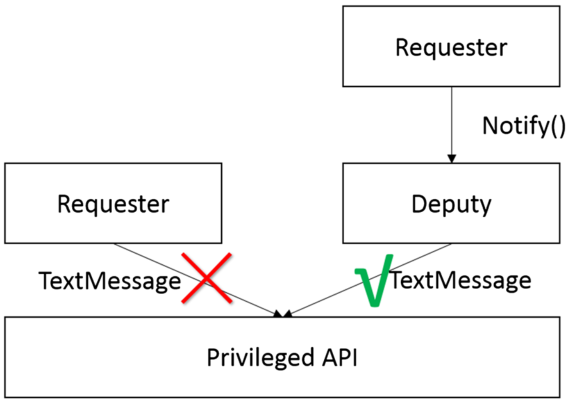


Figure 1. Confused deputy attack [5].

deputy does. The deputy also defines a **Notify** method to invoke a privileged system API call to send text messages. When the requester calls the deputy's **Notify** method, the system API will send text messages because the deputy has the necessary permissions. As a result, the confused deputy attack succeeds.

With a good understanding of the confused deputy attack on Android, it is easy to draw a conclusion that the two preconditions of successfully conducting a confused deputy attack on Android are (1) an exposed component; and (2) this component performs a privileged task, from which we could design our detecting system.

3. SYSTEM DESIGN

3.1. Overview

There are two main features of the applications to be chosen as confused deputies that our system is based on. First, an application with confused deputy vulnerability has some exposed components, which can be invoked by other applications that are in control of attackers. Because it is a prerequisite to conduct a confused deputy attack that the deputy must have exposed interfaces for public use and does not check if the caller has the appropriate permissions. In addition, the applications chosen to be deputies have privileged operations (this is the purpose to conduct confused deputy attack, to perform privileged operations), like voice flow interception, and so on, which are accessible from exposed interfaces. All these features make us solve the following two problems in sequence to check whether a confused deputy vulnerability exists in an application: (1) to detect whether exported components exist or not in an application; (2) to check whether these exported components have privileged operations.

To solve these problems, we have designed a system shown in Figure 2. The *AndroidManifest Analyzer* first parses the AndroidManifest file and obtains the preliminary exported components. As this kind of detecting process only depends on the AndroidManifest file, it is a kind of coarse-grained detecting and also inaccurate to make a decision whether the application is exploitable to confused deputy attacks or not. In order to remedy this defect, we have further designed two modules: Binary Analyzer and Detector. Binary Analyzer constructs CFG, and Detector inspects whether an application is exploitable to the confused deputy attacks based on the constructed CFG in code level.

3.2. Coarse-grained insecure-components discovery

AndroidManifest file specifies the kinds of operations supported by components [11]. Therefore, the component must be declared in the AndroidManifest file using the *exported* flag or Intent Filters if it is able to receive Intents, which makes PaddyFrog possible to obtain the insecure exported components by analyzing AndroidManifest file. In PaddyFrog, the AndroidManifest analyzing process is

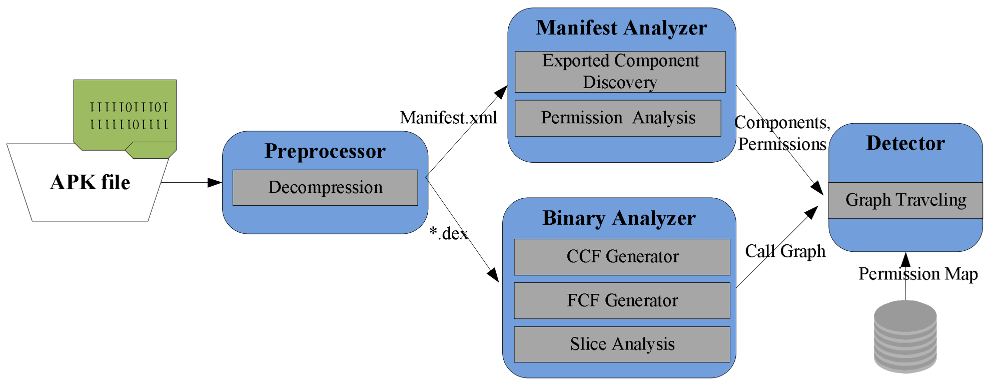


Figure 2. System overview.

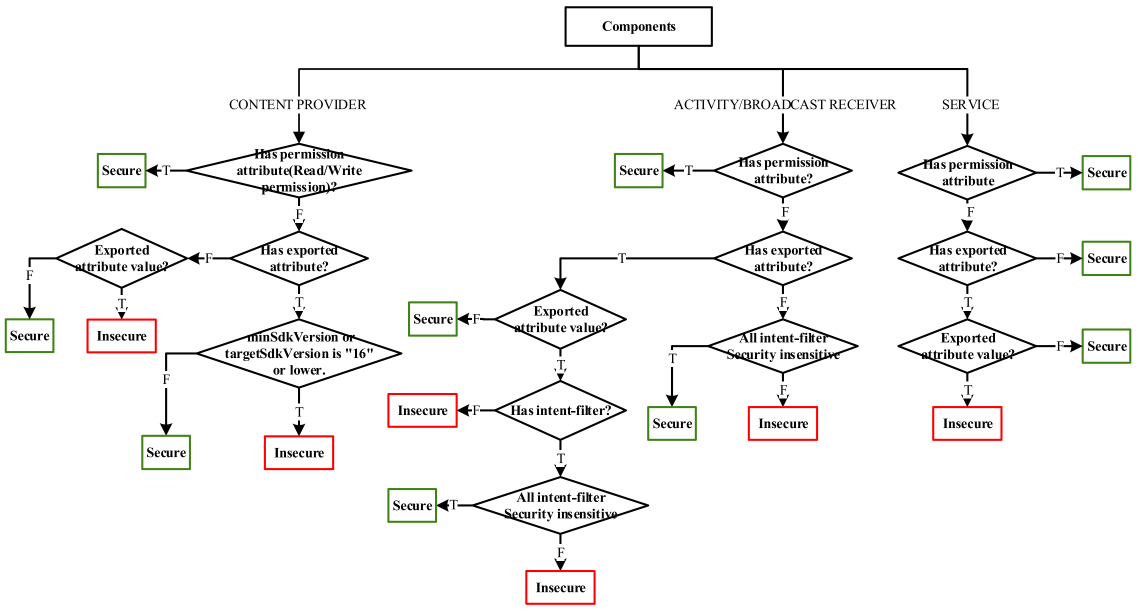

Components
CONTENT PROVIDER
ACTIVITY/BROADCAST RECEIVER
SERVICE
Has permission attribute(Read/Write permission)?
Has exported attribute?
Exported attribute value?
minSdkVersion or targetSdkVersion is "16" or lower.
Has permission attribute?
Has exported attribute?
Exported attribute value?
Has intent-filter?
All intent-filter Security insensitive
Has permission attribute
Exported attribute value?
Secure
Insecure
T
F

Figure 3. Androidmanifest checking process.

divided into two stages: exported components discovery stage and insecure components checking stage.

In the exported components discovery stage, PaddyFrog uses a set of heuristics for inferring whether a component has exposed public interfaces for other applications. These rules' definitions are based on whether the component has exported attribute, whether it is protected with signature permission, and whether it has eligible Intent-filters [12].

To check whether the exported component is insecure, we have proposed some heuristics to filter the insecure components from the exported components in Figure 3. If a component is protected with *android:permission* which declares the permissions that other applications must possess to interact with this component, it means that we can infer whether the component is secure by analyzing the value of *android:permission*. Otherwise, PaddyFrog further deduces some components' security by checking the exported attributes and the related Intent filters following the procedures described in Figure 3. What's more, the term, Intent-filter security insensitive in Figure 3, means that the Intent *action* is not harmful to the security and privacy of the system, and we define this Intent *action* with a whitelist.

The follow-up checking of insecure service components is similar to that of Activity components except that it does not consider whether the Intent filters are security sensitive or not because there is no standard Intent action defined by Android for service components. To the provider component, its exported attributes' default values change before and after that android:targetSdkVersion is 16, which makes PaddyFrog add an additional checking point to check its security.

3.3. Fine-grained insecure-components identification

In this section, we describe how we detect the confused deputy attack in the code level. The core idea of our approach is first to build a CFG and then inspect whether there exists a path from the entries of the exported components to the security sensitive functions on this graph. If paths were found, then we walk through the paths to determine whether dynamic permission checking applied or not. As Android application is a kind of component-based program, the communication between components depends on ICC. This event-trigger mechanism of Android makes the CFG construction become more complicated than the traditional applications on Windows or Linux. In this paper, we divide the CFG construction into two parts: intra-component CFG construction and inter-component CFG construction.

3.3.1. Intra-component control flow graph construction.

We build the CFG for a component from smali files [13], which contain some instructions similar with Dalvik bytecode instructions and most of the techniques used for constructing CFG are quite standard [14]: PaddyFrog finds all the JUMP-like instructions, divides the methods into basic blocks (A basic block is a straight-line sequence of code with only one entry point and only one exit) and generates CFG edges between blocks that contain JUMP-like instructions and blocks that contain JUMP-like instructions' targets.

Although constructing CFG is a well-known topic, the Android-specific aspects make this task complicated. During building the CFG for a component, inheritance and dynamic binding make it impossible to unambiguously determine what concrete a class reference represents, which blurs the target of the indirect call instruction. In our current prototype, we take a conservative approach like [15]. When analyzing an application's smali code, our system maintains a comprehensive class hierarchy, and when an ambiguous reference is encountered, we consider all possible assignable classes.

Additionally, the component-based model of the Android framework determines that there are several entries in a single application, which result in that CFG is not correctly built if we just take one method as the entry. In PaddyFrog, we take all the methods as the beginning entries, while it will be checked if this block has been linked before linking two blocks, which prevents generating duplicate edges.

3.3.2. Inter-component control flow graph construction.

The Android platform uses Intents to communicate with components in an application or with other applications. So, we can build the component switching relationships by tracing the Intent parameters of Start-Another-Component Functions found from Android APIs [16] listed in Table I.

As described in [17], there are two different ways to define an Intent, explicit and implicit. An explicit Intent can specify its particular recipient component by name, whereas an implicit Intent just specifies an action to the system and the actual recipient is determined by system according to the AndroidManifest.xml. In the AndroidManifest.xml, all components (i.e., Activity, Service and Receiver) in this application are specified what Intents they can receive and the specific actions they can perform. For these two kinds of Intents, we adapt different trace-back methods to recognize its target.

For an explicit Intent, its target name can be assigned via the constructors of Intent or three methods, *setClass*, *setCLassName,* and *setComponent*, one of whose parameters is the explicit component name in Android API. Obviously, it is easy to know its target component by back tracing the component name parameter of related invoked functions. Therefore the invoked component is determined.

For an implicit Intent, however, it behaves in a different way. The invoked component of the Intent is assigned in the implicit way, and the Android system will search the global AndroidManifest table to find the components to start whose action field of the Intent-filter matches with that assigned in the Intent [17]. Hence, we back trace the *Action, data,* and so on field of the Intent set by the functions it calls. Then find the components declared in the AndroidManifest that match with this Intent. As a result, the implicit invocation target is determined.

We devised the following algorithm to obtain all the switching relationships among components, and the whole algorithm is described in Algorithm1.

In Algorithm 1, the function FindTarget(r) takes the object reference storing the Intent as input which is a parameter in the "start-another-component-functions" and the set of the target components as output. In SIPSF(r)(SIPSF is short for SearchIntentParameter-

Table I. Start-Another-Component functions.

Starts another Activity	Starts another Service	Starts another Receiver
startActivity	bindService	sendBroadcast
startActivityForResult	startService	sendOrderedBroadcast
startNextMatchingActivity	stopService	sendStickyBroadcast
startActivityIfNeeded		sendStickyOrderedBroadcast
startActivityFromChild		
startActivityFromFragment		

Algorithm 1 Inter-Components Call Graph Construction

```
Require: s is the set of functions invoked by the Intent r; r is the register which keeps the reference of Intent; u is the set of Action; b is the block that the Intent belongs to; n is the component name parameter of the functions which set the component name to the Intent; p is the Action parameter of the functions which set the Action field of the Intent
Ensure: t is the set of the target component
1:
2: function FINDTARGET($r$)
3:     $t \leftarrow \phi$;
4:     $u \leftarrow \phi$;
5:     $s \leftarrow SIPSF(r)$;
6:     if s contains constructors of Intent or functions that set a specific component to the Intent then   /* handlingexplicitIntents */
7:         TraceParameterValue($n, t, b$);
8:     else                 /* handlingimplicitIntents */
9:         TraceParameterValue ($p, u, b$);
10:        FindMatchComponent($u, t$);
11:    end if
12:    return $t$
13: end function
14: function SIPSF($r$)
15:    $s \leftarrow \phi$;
16:    for each instruction $i$ do
17:        if $i$ is INVOKE then
18:            if $i$ is invoked by $r$ then
19:                $s \leftarrow s \cup i$;
20:            end if
21:        end if
22:    end for
23:    return $s$
24: end function
25: function TRACEPARAMETERVALUE($r, t, b$)
26:    for each instruction $i$ in block $b$ do
27:        if $i$ is Move-op $r, n$ then          /* $r \leftarrow n$ */
28:            TraceParameterValue($n, t, b$);
29:        else if $i$ is const-op $r, C$ then     /* $r \leftarrow C$ */
30:            $t \leftarrow t \cup C$;
31:            return
32:        else
33:            for each block link to $b$ do
34:                TraceParameterValue($r, t, block$);
35:            end for
36:        end if
37:    end for
38: end function
39: function FINDMATCHCOMPONENT($u, t$)
40:    for each action act in $u$ do
41:        for each component com in AndroidAndroid-Manifest do
42:            if act match with the intent-filter of com then
43:                $t \leftarrow t \cup com$;
44:            end if
45:        end for
46:    end for
47: end function
```

SettingFunctions), the algorithm searches every possible branch backwards to find the most "close" function which sets the attributes of the Intent (i.e., the constructors , setClass(), setClassName(), setComponent(), and setAction() invoked by the Intent). If the invoked function set contains the constructor which needs a specific component name as a parameter or setClass() or setClassName() or setComponent(), it means the target component is invoked in the explicit way, and we use TraceParameterValue (r, t, b) to back trace the component name. In TraceParameterValue (r, t, b), the algorithm scans each instruction in the block. If it's move-op r, n, invoke TraceParameterValue(n, t, b) recursively; if it's const-op r,C, add C to the set t and return; if cannot determine the value of the register in the block, run TraceParameterValue() on each of the blocks that invokes this block. There are also limitations of TraceParameterValue() that we cannot obtain the right result if component's name is not constant, which will be solved in the future. If the set is empty which means there are no functions that assign the specific component name to the Intent in the set, it implies that the target component is invoked in implicit way, and the *Action* field must be set. We back trace the parameter of the constructor which needs *Action* as a parameter or setActon() function to obtain the value of the Action field of the Intent by TraceRegisterValue(). After Actions are found, FindMatchComponent() is called to find the matched components.

3.3.3. Permission check identification.

We have searched all the functions that need extra permissions throughout Android APIs and found 73 functions that are able to acquire users' privacy information (e.g., system log and Mandatory Access Control address) and 10 functions that are able to perform some dangerous actions (e.g., send a test message, make a call). We labeled these functions as "sensitive" functions and formed Table II. The constructed CFG includes a tremendous number of function invoking paths. Among all these possible paths, not all of them mean a confused deputy vulnerability for the following reasons: (1) There is no sensitive (privileged) functions like the functions in the Table II invoked on the possible paths; (2) Some permission checking functions (i.e., Context.checkCallingPermission() and Context.checkPermission()) may appear on the possible paths, which ensure that the application caller has the required permissions. However, we cannot obtain this information by solely analyzing the AndroidManifest file.

Hence, we designed the following CFG-based checking algorithm to reduce the false positive(or negative) ratio caused by relying solely on the AndroidManifest file.

Before running the algorithm, we scan the whole smali code to locate all the sensitive function invocations. We search backwards in the CFG from the blocks that contain sensitive function invocations to determine if there is a path between this block and the exported insecure components. We define the APK as secure if there is no path found. If there is a path, we check whether the 16 permission-check functions (i.e., checkCallingPermis-

Table II. Different sensitive function categories.

Sensitive function categories	Function number	Descriptions
Telephony identifiers	23	IMEI, IMSI, MCC, MNC, LAC, CID, etc.
Account information	19	account name, account password, etc.
GPS coordination	4	current GPS Geo location
Web browser information	11	browser history
WIFI connection information	14	WiFi credentials, MAC address, etc.
Audio video flow	2	call recording, video capture, etc.
Telephony services abuse	3	premium SMS sending, phone call composition...
Arbitrary code execution	7	native code...

IMEI, International Mobile Equipment Identity; IMSI, international mobile subscriber identity; MCC, mobile country code; MNC, mobile network codes; LAC, location area code; CID, Customer identity; MAC, mandatory access control.

Algorithm 2 Checking Algorithm

```
Require: b is the block which contains sensitive function
    invocation; s is the set of possible paths
Ensure: whether this sensitive function is exploitable
 1: function IsVulnerable(b)
 2:     s ← $\phi$
 3:     p ← $\phi$
 4:     PathSearch(b, s)
 5:     for each path p in s do
 6:         if NOT HasPermissionCheck(p) then
 7:             return true
 8:         end if
 9:     end for
10:     return false
11: end function
12: function HasPermissionCheck(p)
13:     for each node n in p do
14:         if n invokes permission-check functions then
15:             if the permission checked matches with the
    permission required then
16:                 return true
17:             end if
18:         end if
19:     end for
20:     return false
21: end function
```

sion and checkCallingUriPermission) are invoked on the path. If one of them is called, we inspect that whether the checked permission is what the privileged function requires, which determines if it is secure or not; otherwise, we define the application as vulnerable.

4. EVALUATION AND RESULTS

The whole system is written in Java with more than 8600 lines of code. APKtool is needed to decompile applications from APK format to smali files which is the format of PaddyFrog's input before processed by PaddyFrog. There are several versions of APKtool, and here we choose the newest version at this time, 1.4.1. With experimental computer equipped with an Intel Core i5-3320 CPU of 2.6GHz and 4GB of RAM, we carried out an in-depth evaluation on PaddyFrog in terms of its effectiveness and performance.

Table III. Result 1 of PaddyFrog's analysis.

Total apps#	7018
Apps with public exported interfaces# [1]	1929
Potential[2] vulnerable apps#	1250
Vulnerable[3] apps#	1240

[1] This is the result by only Androidmanifest analysis.
[2] "Potential" means applications with exported interfaces and sensitive function invocations. What's more, there are paths detected from exported interfaces to sensitive functions.
[3] "Vulnerable" means applications with exported interfaces, sensitive function invocations and there are paths from exported interfaces to sensitive functions WITHOUT any dynamic permission check on the paths.

4.1. Dataset preparation

We have crawled 7190 real-world applications from two of the most popular markets in China, HiApk [7] and Anzhi market [8](5400 from HiApk and 1790 from Anzhi market) from 1 to 15 October 2012 to make our evaluation more "real". Over 7000 real-world applications are enough to perform a test on PaddyFrog in all aspects for what we do is to evaluate our system and make our results as typical as possible at the same time rather than to take a survey of markets applications. Besides, the applications we have downloaded are all top 100 downloads of their types, which makes our results more typical. These applications are from different categories including SMS applications, web browsers, office applications and games. The overall experiments are conducted on these 7190 applications stated before. In these 7190 applications, 172 applications cannot be disassembled, for the limitations of APKtool. Therefore, we tested our system on the remaining 7018 applications.

4.2. Results and analysis

We have evaluated our methods on the downloaded 7018 applications. The results are listed in Table III.

Table IV. Result 2 of PaddyFrog's analysis.

Apps with public exported interfaces#	1929
Apps with public exported interfaces but no sensitive function invocations# [1]	679
Apps with public exported interfaces & sensitive function invocations# [2]	1250

[1] "No sensitive function invocations" means that there is no path found from exported interfaces to sensitive functions.
[2] There are paths found from exported interfaces to sensitive functions.

Table V. Result 3 of PaddyFrog's analysis.

Apps with public exported interfaces & sensitive function invocations#	1250
dynamic permission check#	10
Vulnerable apps#	1240

From Table III, we know that 1929 applications with public exported interfaces were discovered from the 7018 applications, and their percentage is about 27.4% when we only make use of the AndroidManifest file analysis. In these 1929 applications, 1250 are detected with exported interfaces and sensitive function invocations from these interfaces. However, there are only 1240 applications that are really vulnerable, which accounts for less than 18% of the overall applications detected by PaddyFrog.

Table IV tells us that there are 679 applications that do not call any sensitive functions listed in Table II from exported interfaces, which cannot endanger our mobiles but reported as vulnerable in [6] in these 1929 applications with exported interfaces. The other 1250 applications have public exported interfaces and invoke sensitive functions from these interfaces, which may endanger our phones. However, not all of them are vulnerable to confused deputy attack.

For the remaining 1250 applications, they have public exported interfaces and sensitive function invocations. Will all these applications be exploitable for the confused deputy attack? Table V shows us that in these 1250 applications, only 1240 applications are really vulnerable. As to the other 10 applications, they are integrated with some dynamic permission-checking functions in code level, which can prevent them from being invoked illegally. It's noteworthy that there are 9 applications, which adapt the dynamic permission-checking functions in some vulnerable paths but not in all paths, so they are still not able to prevent confused deputy attack. We also checked the 10 applications with permission-checking functions manually. And we have found that one application is still exploitable even with permission-checking functions. As we dig deeper, we have figured out the scenario that the author of the application do checks the permission before this privileged operation, but he leaves the result of checking alone and continue to perform the operation. That is, the operation will be performed no matter whether the permission is met or not. Thus, PaddyFrog cannot prevent false negative.

Table VI. Accuracy of PaddyFrog.

Vulnerable apps detected by PaddyFrog#	1240
Apps checked manually#	124
Vulnerability detected apps#	124

We define an application as vulnerable only if there is a path that exists from the public exported interfaces to the privileged operations and no dynamic permission check functions called on the path, which means that the callers with less permissions can perform privileged operations and is proved to be effective. Under this definition, the 1240 applications are vulnerable, and there is no false positive for that we have found at least one vulnerable path for each of the applications, and we have proved that (actually, we have verified part of them).

4.3. Effectiveness and performance

We have selected 10% percent (namely 124) of the applications detected vulnerable by PaddyFrog randomly to be checked manually due to the large amount of detected applications to evaluate the effectiveness of PaddyFrog, and the result is shown in Table VI. It's shown that all of the selected 124 applications are proved vulnerable, and there is no false positive. Actually we do not "exploit" the applications during the manual checking process. We write little simple applications to invoke the vulnerable components detected by PaddyFrog to verify whether these simple applications could perform privileged operations without related permissions. It is worthy to note that being able to perform unauthorized operations is also a kind of exploit. Although there is no false positive, PaddyFrog cannot prevent false negative because of code obfuscation as stated before.

In regard to performance, we instrumented PaddyFrog to measure its execution time while it detects all the applications. The median total processing time for an application is 14.4s, with median processing time for AndroidManifest 1.5s, disassembling 8.3s and smali analysis 4.6s, which suggests that PaddyFrog can quickly inspect a large amount of applications for confused deputy vulnerability.

4.4. Case study

To understand the nature of our tool, we talk about two scenarios in depth with examples. These scenarios were selected to illustrate some of the patterns we encountered in practice, as well as how our system was able to handle them.

4.4.1. UC Browser.

The simplest scenario, which would be assumed as vulnerable by [6], is that there are public exported interfaces in the application, while these interfaces do NOT invoke any

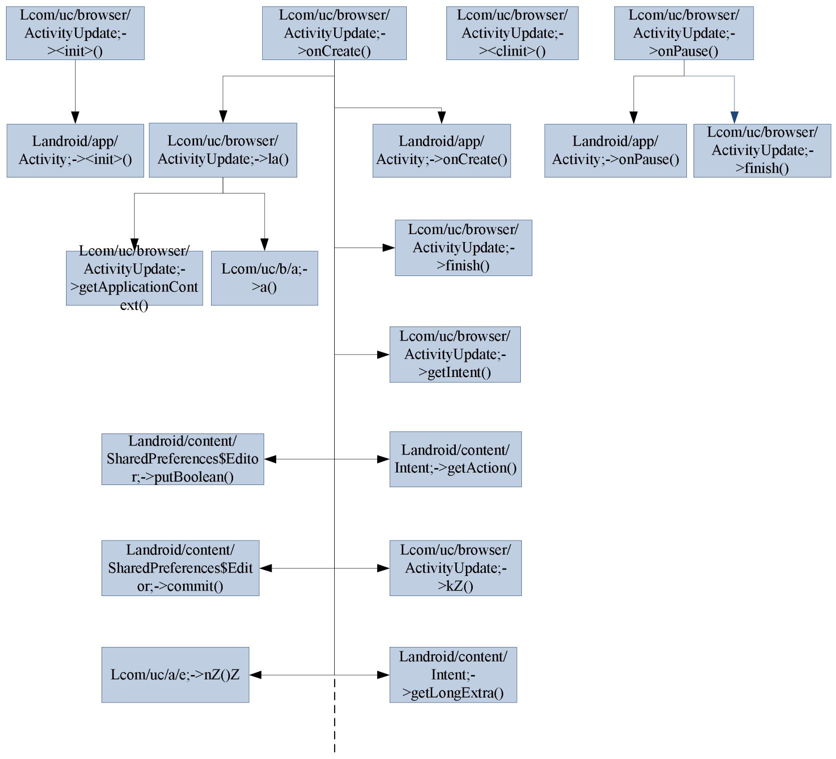

Lcom/uc/browser/ActivityUpdate;-><init>()
Lcom/uc/browser/ActivityUpdate;->onCreate()
Lcom/uc/browser/ActivityUpdate;-><clinit>()
Lcom/uc/browser/ActivityUpdate;->onPause()
Landroid/app/Activity;-><init>()
Lcom/uc/browser/ActivityUpdate;->la()
Landroid/app/Activity;->onCreate()
Landroid/app/Activity;->onPause()
Lcom/uc/browser/ActivityUpdate;->finish()
Lcom/uc/browser/ActivityUpdate;->getApplicationContext()
Lcom/uc/b/a;->a()
Lcom/uc/browser/ActivityUpdate;->finish()
Lcom/uc/browser/ActivityUpdate;->getIntent()
Landroid/content/SharedPreferences$Editor;->putBoolean()
Landroid/content/Intent;->getAction()
Landroid/content/SharedPreferences$Editor;->commit()
Lcom/uc/browser/ActivityUpdate;->kZ()
Lcom/uc/a/e;->nZ()Z
Landroid/content/Intent;->getLongExtra()

Figure 4. UC Browser.

sensitive functions directly or indirectly. Actually, these components can do little harm to our phones. In PaddyFrog, it first searches for all methods $F()$ invoked by the public exported interfaces directly or indirectly in the smali files. After all invoked methods are found, examines each method $F()$ to determine if it is a sensitive function or not. The application is defined as secure if all the methods $F()$ are not sensitive functions.

Take the APK UC Browser-7.9.4 as an example. The component, com.uc.browser.ActivityUpdate is inspected exported by manifest analysis. The call graph inside the component is described in Figure 4.

According to the graph, all the functions invoked from the exported component, com.uc.browser.ActivityUpdate are common functions which requires no extra permissions. No privileged functions found invoked by the exported component means this component is not exploitable to confused deputy attack although it could be used as a confused deputy. Thus, it can do little harm to the phone although it is exported.

4.4.2. LBE Privacy Guard.

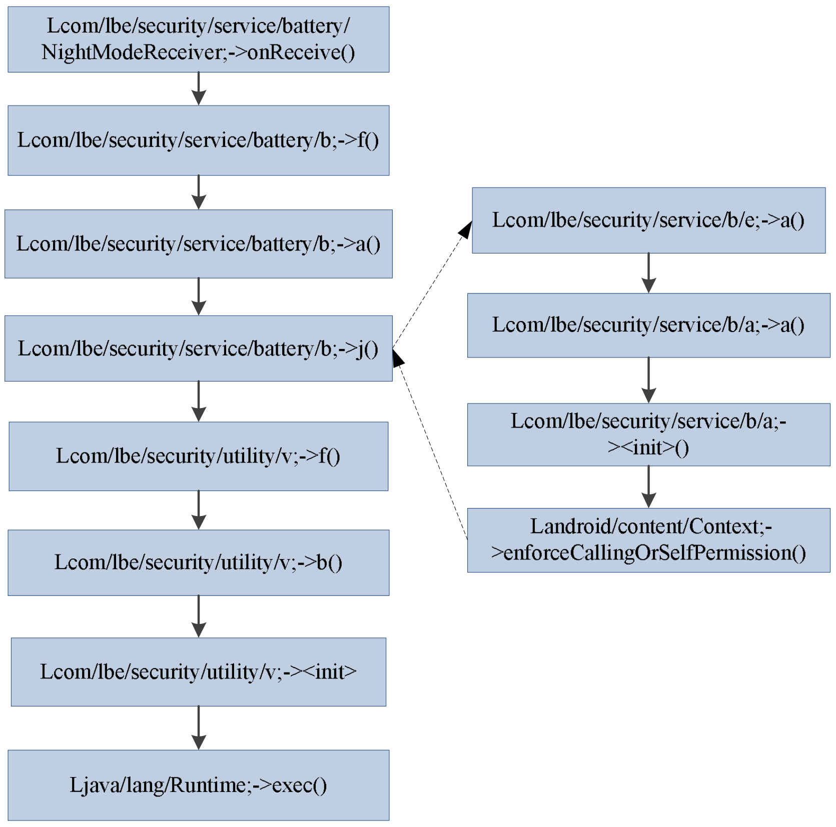
We define the APK as secure if there is a permission check before the invocation of the sensitive functions although there is a path between the sensitive functions and the exported interfaces. Take the APK LBE-Privacy-Guard-4.0.1894 as an example. The exported interface acquired by the AndroidManifest analysis is com.lbe.security.service.battery.NightM odeReceiver. The path between the sensitive function Ljava/lang/Runtime;->exec() and the exported interface is showed in Figure 5.

As showed in the graph, the sensitive function Ljava/lang/Runtime;->exec() could be invoked indirectly from the exported interface com.lbe.security.service .battery.NightMode- Receiver. However, before the sensitive function is invoked, the application called the check-permission function via the path, com.lbe.security.service.b.e.a(), com.lbe.security.service.b.a.a(), com.lbe.security.service.b.a.<init>(). Therefore, the permission is checked before the sensitive function is invoked, which can prevent confused deputy attack. The same as the case in Section 4.4.1, it's detected as vulnerable according to [6], however, the attack could be prevented by the permission-checking process. Thus, our tool could reduce the false positive rate introduced by [6].

5. DISCUSSION

Our evaluation results show that our prototype can effectively detect confused deputy vulnerabilities. In this section, we further talk about possible limitations in our system and explore ways for future improvements.

Firstly, the checking results depend on the sensitive function set (as shown in Table II) and the checking-caller-permission functions. If they are incomplete, it will take great bad affects to the results. However, PaddyFrog is



Lcom/lbe/security/service/battery/NightModeReceiver;->onReceive()
Lcom/lbe/security/service/battery/b;->f()
Lcom/lbe/security/service/battery/b;->a()
Lcom/lbe/security/service/battery/b;->j()
Lcom/lbe/security/utility/v;->f()
Lcom/lbe/security/utility/v;->b()
Lcom/lbe/security/utility/v;-><init>
Ljava/lang/Runtime;->exec()
Lcom/lbe/security/service/b/e;->a()
Lcom/lbe/security/service/b/a;->a()
Lcom/lbe/security/service/b/a;-><init>()
Landroid/content/Context;->enforceCallingOrSelfPermission()

Figure 5. LBE Privacy Guard.

only a testing prototype and when needed, we can add additional relevant functions into PaddyFrog to improve its performance.

Secondly, PaddyFrog could perform unpredictably when it comes to code obfuscation, which is also a very big challenge faced by static analysis. We will apply dynamic analysis techniques to detecting in the future work due to the shortcoming of static analysis. Besides, dynamically determining which component to start at runtime can also fool PaddyFrog.

Thirdly, we consider ICC channels as the only mechanism available in Android to establish communications between applications. However, in some cases applications may communicate by other methods rather than ICC. For example, applications may establish communication channels via Internet socket, file system, and so on, which completely bypass Android's middleware layer but can be used to implement confused deputy attack. To take these mechanisms into consideration, we can extend the construction of CFG.

Lastly, Java language has the possibility to load classes dynamically, and if there are some sensitive functions existed in the loaded classes, it is possible to be used to implement confused deputy attack. However, the characteristics of dynamic loading makes static analysis impossible because the loaded classes are only known at runtime. Up to now, PaddyFrog cannot deal with it, but in the future, we try to solve them by analyzing the parameters of functions *loadClass*.

6. RELATED WORK

Researchers have analyzed the security model of Android's permission system [18], assessed the framework of Android [19], surveyed attack formats in Android [20], developed analysis tools for Android applications [21,22], and proposed new protection mechanisms (e.g., [23,24]). PaddyFrog is different from these efforts with its unique focus on a flaw in the Android's permission system, which makes application exploitable for the confused deputy attack.

Several works have pointed out flaws of the current Android permission system. One weakness is the lack of global properties: Android's permission system does not prevent privilege escalation. Davi *et al.* [25] and Felt *et al.* [5] have studied privilege-escalation attacks in detail. ComDroid [26] uncovers possible unintended consequences of exposing certain application components. At the same time, some researchers investigated different mitigation mechanisms. Bugiel *et al.* [27] developed a system that monitors interactions between applications at runtime and has the capacity to mitigate a wide range of privilege escalation attacks. Quire [28] similarly addresses the permission delegation problem by proposing an IPC call chain tracking mechanism to identify the provenance of these IPC requests and enforce certain policy. Dietz *et al.* [28] proposed a framework for provenance tracking to mitigate the confused deputy problem. Bugiel *et al.* [6] assigned trust levels to applications, allowing applications to communicate only if they are at the same level. All of these works focus on implementation mechanism of privilege escalation attack or how to prevent this kind of attack. However, our work focuses on how to use static program analysis method to detect the existence of the privilege escalation vulnerability in an application.

From another perspective, these works have a relation with the privacy leakage and several works have identified the problem of privacy leaks in Android [29,30]. Projects such as TaintDroid [29], ScanDroid [31] and PiOS [32] aim to automatically detect and prevent dangerous private information leaks in Android. PiOS [32] develops a static analysis tool to spot possible information leaks in iOS applications. In our work, we only learn how to define the privacy information from these works.

To the best of our knowledge, the closest work to ours is Chan *et al.* [6]. They all proposed a vulnerability checking system to detect benign applications, which fail to enforce the additional checks on permissions granted. However, Chan [6] *et al.* looks into the AndroidManifest file only, but our works use the missing information at the code level and overcome many limitations described by Chan *et al.* [6]. Hence, their tool is more coarse-grained and produces more false positives.

7. CONCLUSIONS

In this research, we first designed a static analyzer, PaddyFrog, to detect the confused deputy vulnerability in code level. In doing this, we introduced how to find the public exposed interfaces based on the AndroidManifest file, build the CFG inside and outside of the components, and detect confused deputy vulnerability based on the CFG. PaddyFrog prototype was implemented based on the disassembler APKtool and was evaluated with 7190 real-world applications. The empirical experiment demonstrated a satisfactory scalability and performance of our analysis method, as well as we provided two insight cases to illustrate why this method is more precise effective.

ACKNOWLEDGEMENTS

This work is partially supported by National Natural Science Foundation of China (61173068, 61173139), Program for New Century Excellent Talents in the University of the Ministry of Education, the Key Science Technology Project of Shandong Province (2010GGX10117, 2014GGD01063), the Independent Innovation Foundation of Shandong Province (2014CGZH1106), and the Shandong Provincial Natural Science Foundation (ZR2014FM020).

REFERENCES

1. Nielsen. Who is winning the U.S. smartphone battle? 2011. http://blog.nielsen.com/nielsenwire/onlinemobile/who-is-winning-the-u-s-smartphone-battle/ [accessed on 12 October 2012].
2. Lookout. App genome report, 2011. https://www.mylookout.com/ [accessed on 12 October 2012].
3. Quality of android market apps is pathetically low, 2011. http://www.huffingtonpost.com/2011/06/20/android-market-quality_n_880478.html [accessed on 12 October 2012].
4. Hardy N. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review* 1988; **22**(4): 36–38.
5. Felt AP, Wang HJ, Moshchuk A, Hanna S, Chin E. Permission re-delegation: attacks and defenses, *Proceedings of the 20th USENIX Conference on Security,* SEC'11, USENIX Association, Berkeley, CA, USA, 2011; 22–22.
6. Chan PP, Hui LC, Yiu S. A privilege escalation vulnerability checking system for android applications, *Proceedings of 13th IEEE International Conference on Communication Techonologies,* ICCT '11, IEEE Computer Society, Jinan, China, 2011.
7. *Hiapk*. http://apk.hiapk.com/ [accessed on 2 October 2012].
8. *Anzhi market*. http://www.anzhi.com/ [accessed on 2 October 2012].
9. *Android developer's guide*, 2011. http://developer.android.com/guide/ [accessed on 12 November 2012].
10. *Confused deputy*. http://cwe.mitre.org/data/definitions/\penalty-\@M441.html [accessed on 20 September 2012].
11. *Androidmanifest*, 2012. http://developer.android.com/guide/topics/manifest/manifest-intro.html.
12. Kantola D, Chin E, He W, Wagner D. Reducing attack surfaces for intra-application communication in android, *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices,* SPSM '12, ACM, New York, NY, USA, 2012; 69–80.
13. *apktool*, 2012. http://code.google.com/p/android-apktool/ [accessed on 20 October 2012].
14. Theiling H. Extracting safe and precise control flow from binaries, *Proceedings of the Seventh International Conference on Real-Time Systems and Applications,* RTCSA '00, IEEE Computer Society, Washington, DC, USA, 2000; 23–30.
15. Michael Grace ZW, Yajin Z, Jiang X. Systematic detection of capability leaks in stock android

smartphones, *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '12, 2012.

16. *Android ap*, 2011. http://developer.android.com/reference/packages.html [accessed on 15 October 2012].
17. *Intents and intent filters*, 2011. http://developer.android.com/guide/components/intents-filters.html [accessed on 15 October 2012].
18. Barrera D, Kayacik HG, van Oorschot PC, Somayaji A. A methodology for empirical analysis of permission-based security models and its application to android, *Proceedings of the 17th ACM Conference on Computer and Communications Security,* CCS '10, ACM, New York, NY, USA, 2010; 73–84.
19. Shabtai A, Fledel Y, Kanonov U, Elovici Y, Dolev S, Glezer C. Google android: a comprehensive security assessment. *Security & Privacy, IEEE* 2010; **8**(2): 35–44.
20. Vidas T, Votipka D, Christin N. All your droid are belong to us: a survey of current android attacks. *Woot* 2011; 81–90.
21. Felt AP, Chin E, Hanna S, Song D, Wagner D. Android permissions demystified, *Proceedings of the 18th ACM Conference on Computer and Communications Security,* CCS '11, ACM, New York, NY, USA, 2011; 627–638.
22. Michael Grace QZ, Yajin Z, Zou S, Jiang X. Riskranker: scalable and accurate zero-day android malware detection, *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, ACM, Low Wood Bay, Lake District, United Kingdom, 2012; 281–294.
23. Nauman M, Khan S, Zhang X. Apex: extending android permission model and enforcement with user-defined runtime constraints, *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security,* ASIACCS '10, ACM, New York, NY, USA, 2010; 328–332.
24. Ongtang M, McLaughlin S, Enck W, McDaniel P. Semantically rich application-centric security in android, *Proceedings of the 2009 Annual Computer Security Applications Conference,* ACSAC '09, IEEE Computer Society, Washington, DC, USA, 2009; 340–349.
25. Davi L, Dmitrienko A, Sadeghi A-R, Winandy M. Privilege escalation attacks on android, *Proceedings of the 13th International Conference on Information Security,* ISC'10, Springer-Verlag, Berlin, Heidelberg, 2011; 346–360.
26. Chin E, Felt AP, Greenwood K, Wagner D. Analyzing inter-application communication in android, *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services,* MobiSys '11, ACM, New York, NY, USA, 2011; 239–252.
27. Bugiel S, Davi L, Dmitrienko A, Fischer T, Sadeghi A-R, Shastry B. Towards taming privilege-escalation attacks on android, *19th Annual Network & Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2012.
28. Dietz M, Shekhar S, Pisetsky Y, Shu A, Wallach DS. Quire: lightweight provenance for smart phone operating systems. *CoRR* 2011; **1102.2445**.
29. Enck W, Gilbert P, Chun B-G, Cox LP, Jung J, McDaniel P, Sheth AN. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones, *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation,* OSDI'10, USENIX Association, Berkeley, CA, USA, 2010; 1–6.
30. Enck W, Octeau D, McDaniel P, Chaudhuri S. A study of android application security, *Proceedings of the 20th USENIX Conference on Security,* SEC'11, USENIX Association, Berkeley, CA, USA, 2011; 21–21.
31. Fuchs AP, Chaudhuri A, Foster JS. Scandroid: automated security certification of android applications. *CS-TR-4991*, Department of Computer Science, University of Maryland, College Park, 2009.
32. Egele M, Kruegel C, Kirda E, Vigna G. Pios: detecting privacy leaks in ios applications, *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2011.