6-2016

ICCDetector: ICC-based malware detection on Android

Xu KE
Singapore Management University, kexu.2013@phdis.smu.edu.sg

Yingjiu LI
Singapore Management University, yjli@smu.edu.sg

Robert H. DENG
Singapore Management University, robertdeng@smu.edu.sg

DOI: https://doi.org/10.1109/TIFS.2016.2523912

Published in IEEE Transactions on Information Forensics and Security, 2016 June, Volume 11, Issue 6, 1252 - 1264.
http://dx.doi.org/10.1109/TIFS.2016.2523912

ICCDetector: ICC-Based Malware Detection on Android

Ke Xu, Yingjiu Li, and Robert H. Deng

Abstract—**Most existing mobile malware detection methods (e.g., Kirin and DroidMat) are designed based on the resources required by malwares (e.g., permissions, application programming interface (API) calls, and system calls). These methods capture the interactions between mobile apps and Android system, but ignore the communications among components within or cross application boundaries. As a consequence, the majority of the existing methods are less effective in identifying many typical malwares, which require a few or no suspicious resources, but leverage on inter-component communication (ICC) mechanism when launching stealthy attacks. To address this challenge, we propose a new malware detection method, named ICCDetector. ICCDetector outputs a detection model after training with a set of benign apps and a set of malwares, and employs the trained model for malware detection. The performance of ICCDetector is evaluated with 5264 malwares, and 12 026 benign apps. Compared with our benchmark, which is a permission-based method proposed by Peng et al. in 2012 with an accuracy up to 88.2%, ICCDetector achieves an accuracy of 97.4%, roughly 10% higher than the benchmark, with a lower false positive rate of 0.67%, which is only about a half of the benchmark. After manually analyzing false positives, we discover 43 new malwares from the benign data set, and reduce the number of false positives to seven. More importantly, ICCDetector discovers 1708 more advanced malwares than the benchmark, while it misses 220 obvious malwares, which can be easily detected by the benchmark. For the detected malwares, ICCDetector further classifies them into five newly defined malware categories, which help understand the relationship between malicious behaviors and ICC characteristics. We also provide a systemic analysis of ICC patterns of benign apps and malwares.**

Index Terms—**ICC, malware detection, Android.**

I. Introduction

MANY existing malware detection methods are designed to detect malwares based on required resources, such as permissions, suspicious API calls and system calls. For example, Kirin [1] detects malwares by matching their required permissions against pre-defined security rules. DroidMiner [2] and DroidAPIMiner [3] build malware detection models based on API-related features. Most of these methods treat the detected applications as standalone entities in Android platforms.

However, in order to bypass existing detection methods, malwares move to another direction by conducting malicious operations without requiring suspicious resources. As observed by [4]–[7], malwares may conduct multiple attacks (e.g., *Confused Deputy* attacks and *Collusion* attacks) by manipulating other apps. In fact, instead of being independent to each other, Android applications may communicate through the Inter-Component Communication (ICC) mechanism provided by Android, which is designed to reduce the developers' burden and promote functionality reuse [8]. Although ICC facilitates inter-application collaboration, it can be exploited by malwares to obfuscate malicious behaviors and bypass existing detection methods. For example, consider a malware (`com.jx.theme`) which aims to install APK files during runtime. Instead of requiring the corresponding permission (`android.permission.INSTALL_PACKAGE`), which should not be used by third-party apps, and thus can be detected by most existing detection methods, this malware generates an Explicit Intent, sends it to `Package: com.android.packageinstaller, Class: com.android.packageinstaller.PackageInstallerActivity`, and manipulates the latter to install some APK files from SD cards. It is difficult to detect such malwares from their required resources without inspecting the ICC information involved.

As a pioneer to address such challenge, we systemically analyze ICC patterns of benign apps and malwares, and propose **ICCDetector**, an effective and accurate malware detection method, which detects malwares based on not their required resources, but their ICC patterns. The ICC patterns of an app represent how it use the ICC mechanism, and can be extracted from the app's APK file. ICCDetector is trained with the ICC patterns extracted from some benign apps and those from certain malwares before it outputs a detection model. The detection model is used to detect a malware based on its ICC patterns. By looking into the ICC patterns, ICCDetector not only examines the communications between applications and Android system, but also the interactions between applications. Because of this, ICCDetector is especially useful for detecting those "advanced malwares" which invalidate most existing malware detection methods by exploiting the ICC mechanism instead of requiring suspicious resources.

We collect 5,264 recent malwares and 12,026 benign apps to evaluate the effectiveness and accuracy of ICCDetector. For comparison, we choose a highly cited malware detection method [9] as a benchmark, which detects malwares according to their required permissions. With the same dataset, the evaluation result indicates that ICCDetector achieves an accuracy of 97.4%, roughly 10% higher than the benchmark. Furthermore, ICCDetector produces a false positive rate of 0.67%, which is only about one half of the benchmark. After manually analyzing false positives, we discover 43 new malwares from the benign dataset, and reduce the number false positives to seven.

For detected malwares, ICCDetector further classifies them into five new malware categories according to their ICC patterns. The classification clarifies the relationship between malware behaviors and ICC characteristics. In addition, we test the runtime performance of ICCDetector, identify the performance bottleneck and specify the directions for performance improvement.

The rest of the paper is organized as follows. Section II describes ICC patterns of apps. Section III details the system design of ICCDetector. Section IV evaluates ICCDetector in different aspects, including a comparison with a benchmark, a analysis of detection performance, a classification of malwares, and runtime measurement. Section V discusses some recent work on Android malware detection and the limitations of ICCDetector. Section VI summarizes the related work, and Section VII concludes the paper.

II. ICC Patterns

In this section, we identify the ICC patterns of benign apps and malicious apps, provide systemic analysis of ICC patterns, and clarify how ICC patterns can be used to distinguish between benign apps and malicious apps.

A. App Components

An Android application consists of four types of components, *Activity, Service, Broadcast Receiver*, and *Content Provider*. *Activity* provides a screen with which users can interact in order to do something. All visible portions of applications are *Activities*. *Service* can perform long-running operations in the background and does not provide a user interface. *Content Provider* manages access to a structured set of data. *Broadcast Receiver* receives information sent from multiple applications. An application must declare the names of its *Activity, Service* and *Content Provider* components in its manifest file. However, application developers are allowed to register *Broadcast Receiver* at run time to listen for specific broadcasts during a specified period of time. That is, applications can declare `Broadcast Receiver` both in manifest file and in java code.

Number of Components: In reality, malwares tend to register more *Broadcast Receivers* and less *Activities, Services* and *Content Providers* than benign apps do. A large number of *Broadcast Receivers* enables malwares to monitor system-events, such as network connectivity changes and battery changes. Although some malwares provide legitimate functionalities to end users, these functionalities are limited, which means the number of *Activities, Services* and *Content Providers* declared by malwares are relatively small. For example, without declaring any *Activity* or *Content Provider*, a malware (`com.android.update`) registers only one *Service* (`com.android.update.Updater`) to stealthily download and install malicious APK files to its fetched devices.

Name of Components: Since code reuse is common in the development of malwares, the malwares belonging to the same malware family tend to conduct similar malicious behaviors and reuse some components. For example, four malwares with different package names (`BatteryUpgrade-Tap-To-Start`, `Battery_Upgrade--Tap_to_start`, `BatteryUpgrade-Tap-To-Start-2`, `com.extend.battery`) share some malicious components to conduct similar attacks, such as `com.extend.battery.Splash`, `com.extend.battery.BatteryService` and `com.extend.battery.BootReceiver`.

Interestingly, we discover that malwares are more likely to register some components with similar confusing names so as to fool end users around or evade from detections. For example, `com.gp.geekadoo` is a malware which pretends to be a card game application in markets, and is capable of gaining super user privileges, rewriting system files, and connecting to a command and control server. Specially, `com.gp.geekadoo` includes several components with confusing names, such as `com.google.update.Dialog`, `com.google.update.UpdateService`, and `com.google.update.Receiver`. Without expert knowledge, end users might be confused by these names, and treat this malware as an updated version of Android or a patch of Google.

Moreover, in the Android system, when multiple *Activities* match a single Implicit Intent, the user of the system will be prompted to choose which component should receive and respond to the Intent [10]. With confusing names, the user may be tricked to choose malicious applications.

B. Intents

In the ICC mechanism, Intents are used to link components, and can be sent between *Activities, Services*, and *Broadcast Receivers*. The major functionality of Intents is to start *Activities*, start and stop *Services*, and deliver broadcast information to *Broadcast Receivers*.

Explicit Intents: Explicit Intents specify the components to start with by including targeted package names and class names. Typically, Explicit Intents are used to connect components within the same application and designed for internal application communications [10].

However, malwares can abuse Explicit Intents by sending them to other applications (i.e., external components). For example, a benign application makes its components exposed in order to receive system-generated Intents. In this case, a malware can directly send Explicit Intents to these exposed components. Without strict action check and appropriate permission protections, these benign components will be directly launched and manipulated by malwares.

In order to avoid being detected by traditional malware detection approaches which can capture suspicious permission usages and API calls, malwares find an effective solution by including or dynamically installing additional APK files.

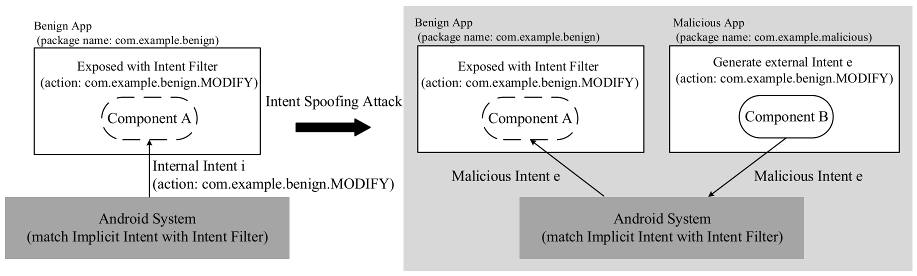
Benign App
(package name: com.example.benign)
Exposed with Intent Filter
(action: com.example.benign.MODIFY)
Component A
Internal Intent i
(action: com.example.benign.MODIFY)
Android System
(match Implicit Intent with Intent Filter)
Intent Spoofing Attack
Benign App
(package name: com.example.benign)
Exposed with Intent Filter
(action: com.example.benign.MODIFY)
Component A
Malicious App
(package name: com.example.malicious)
Generate external Intent e
(action: com.example.benign.MODIFY)
Component B
Malicious Intent e
Malicious Intent e
Android System
(match Implicit Intent with Intent Filter)

Fig. 1. Example of *Intent Spoofing* Attack.

These newly installed APK files are responsible to conduct actual malicious actions. In this case, the original malwares will not be detected since their malicious behaviors cannot be captured by monitoring permissions and API calls. However, these malwares may communicate with the dynamically installed malwares using Explicit Intents, which can be inferred from their ICC patterns.

In addition, malwares tend to send more Explicit Intents to Android system than benign applications do. In our experiments, we discover that many malwares send Explicit Intents to an Android component, `Package: com.android.packageinstaller, Class: com.android.packageinstaller.PackageInstallerActivity`, which is responsible to install APK files saved on SD cards. However, none of the 12,026 popular benign applications which we collected from GooglePlay create such Explicit Intents.

Implicit Intents: Unlike Explicit Intents, Implicit Intents do not name any specific components, but instead declare general actions to perform. When an application creates an Implicit Intent, the Android system finds the appropriate component to start by comparing the contents (i.e., action, category, and data) of the Intent to the declared Intent Filters. If the Intent matches an Intent Filter, the system starts that component and delivers it the Implicit Intent object. There is a variety of system Intent actions and categories defined in the `Intent` class, and applications can define their own actions using their package names as prefixes. This results in two types of actions in Android: `system action` (prefix: `android.-` or `com.android.-`) and `user-defined action` (prefix: `package_name.-`). In particular, it is unusual and suspicious for an application generating Implicit Intent containing actions defined by other applications.

Different from the standard process, malwares may send malicious Implicit Intents to the exposed components of benign applications. These components are exposed to receive system-generated Intents or internal Implicit Intents (which is not recommended by Android due to security consideration). Without necessary action check or permission protections, the exposed components may be launched and manipulated by malwares via Implicit Intents.

For example, a malware may launch an *Intent Spoofing* attack [8] by misusing Implicit Intents as shown in Fig. 1. `Component A` is exposed to receive internal Intents and perform an action (`com.example.benign.MODIFY`). Misusing the Implicit Intent mechanism, malicious `Component B` may trick `Component A` to receive an external Intent (i.e., malicious Intent `e`) instead of the internal Intent (i.e., benign Intent `i`), and perform `com.example.benign.MODIFY` accordingly.

Although it is challenging for normal applications to get the knowledge of other applications' exposed components, it is relatively easy for well-prepared or colluded malwares to attain the knowledge. Malware authors may analyze popular applications and exploit the ICC vulnerabilities in designing their malwares. Also, the same malware authors may develop multiple malwares, which make use of each other's exposed components to perform *Collusion* attacks.

Although Android provides permission checks for sending out Implicit Intents with sensitive action strings (e.g., `android.intent.action.CALL`, and `android.intent.action.REBOOT`) and recommends developers to protect their exposed components (especially *Services*) with permissions, this is not an effective way to prevent Implicit Intents from being misused.

C. Intent Filters

Intent Filters are used to match with Implicit Intents in Android system. The use of Intent Filters in malwares is significantly different from their use in benign applications.

Intercepting Implicit Intents: Malwares may intercept Implicit Intents with Intent Filters in *Component Hijacking* attacks [8], where malicious components are launched in place of the expected benign components. Malwares may also intercept Implicit Intents to read the data included in the Intents, connect to certain applications, or even inject false information into the response returned. As illustrated in Fig. 2, malwares may register appropriate Intent Filters so as to intercept external Intents generated by benign apps.

Intent Filters With Sensitive Actions: Compared with benign applications, malwares especially care about the system-wide events (i.e., system broadcast information). Some of system broadcasts can only be sent by Android system, but can be received by any components with appropriate Intent Filters. Malwares tend to register more Intent Filters for broadcast information related to phone states, such as `android.intent.action.BOOT_COMPLETED`, and `android.intent.action.SMS_RECEIVED`. Some of the Intent Filters that are often misused by malwares are given in Table I, from which we observe that the percentage of malwares registering such Intent Filters is significantly different from the percentage of benign applications registering such Intent Filters. For example,

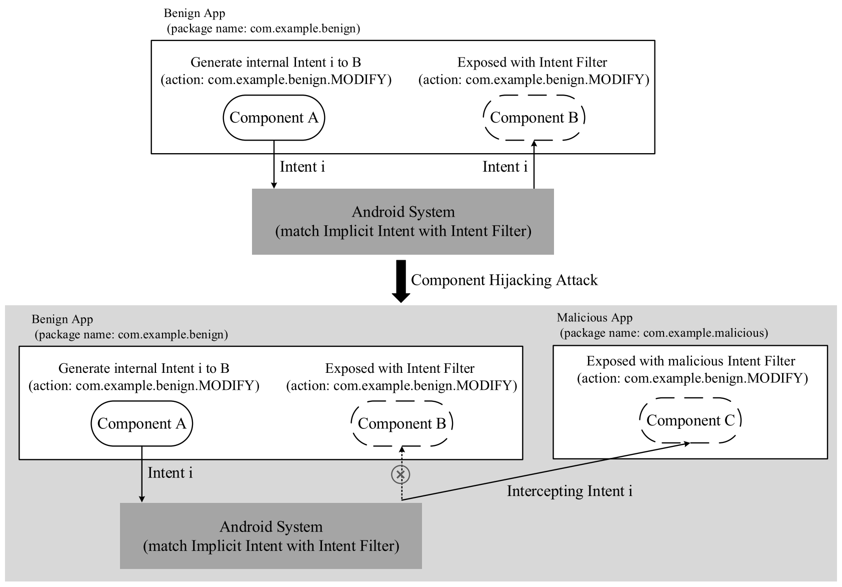
Benign App
(package name: com.example.benign)
Generate internal Intent i to B
(action: com.example.benign.MODIFY)
Component A
Exposed with Intent Filter
(action: com.example.benign.MODIFY)
Component B
Intent i
Intent i
Android System
(match Implicit Intent with Intent Filter)
Component Hijacking Attack
Benign App
(package name: com.example.benign)
Generate internal Intent i to B
(action: com.example.benign.MODIFY)
Component A
Exposed with Intent Filter
(action: com.example.benign.MODIFY)
Component B
Malicious App
(package name: com.example.malicious)
Exposed with malicious Intent Filter
(action: com.example.benign.MODIFY)
Component C
Intent i
Intercepting Intent i
Android System
(match Implicit Intent with Intent Filter)

Fig. 2. Example of *Component Hijacking* Attack.

TABLE I
PERCENTAGE OF BENIGN APPS AND MALICIOUS APPS REGISTERING INTENT FILTERS WITH SENSITIVE ACTIONS

Sensitive Actions	**Benign App% (12,026)**	**Malicious App% (5,264)**
`android.intent.action.BOOT_COMPLETED`	11.9%	60.1 %
`android.intent.action.PACKAGE_ADDED`	3.2%	16.5%
`android.intent.action.POWER_CONNECTED`	2.1%	5.5%
`android.intent.action.SMS_RECEIVED`	1.6%	33.5%
`android.intent.action.PHONE_STATE`	1.2%	10.4%
`android.intent.action.SIG_STR`	0%	12.2%

none of the benign application in the benign dataset (including 12,026 benign applications) register Intent Filter to receive broadcast information related to changes of signal strength (i.e., `android.intent.action.SIG_STR`), while 12.2% of malwares intend to intercept such event.

Number of Intent Filters: Malwares may conduct *Component Hijacking* attacks by exposing their malicious components with Intent Filters. Therefore, it is more likely for malwares to register Intent Filters for their *Activities* and *Services*. In our experiments, 29.32% of malwares declare Intent Filters for *Services*, while only 7.0% of benign apps make *Services* exposed. Furthermore, it is common for malwares to register more Intent Filters, which allows malwares to reliably launch malicious components or payloads.

Registration Mode of Intent Filters: The Registration mode of Intent Filters can serve as an indicator to differentiate between benign apps and malicious apps. The registration of Intent Filters for *Activities* and *Services* must be recorded in the manifest file, while the registration of Intent Filters for *Broadcast Receivers* is flexible, which can be static and dynamic. Android enforces dynamic registration of Intent Filters to keep applications informed with system changes during runtime. However, the dynamic Intent Filters make it possible for malwares to capture specific events in runtime and make necessary responses as required to perform malicious operations.

In our experiments, it is extremely common that malwares dynamically register Intent Filters to capture sensitive broadcasts, such as `android.intent.action.BOOT_COMPLETED`, `android.intent.action.BATTERY_CHANGED` and `android.intent.action.PACKAGE_ADDED`.

Table II summarizes the ICC patterns of benign apps and malicious apps. Benign applications use the ICC mechanism mainly for linking internal components and communicating with the Android system. However, malwares usually manipulate the ICC mechanism for monitoring system events, and creating Intents and Intent Filters to interact with external components.

III. SYSTEM DESIGN

ICCDetector consists of two phases, including Training Phase and Detection Phase as shown in Fig. 3. In the training phase, ICCDetector extracts ICC-related features by analyzing the ICC sources and sinks of certain benign apps and malwares, and generates feature vector for every processed app. A classification method is used to take its input from the generated feature vectors of benign apps and malwares, and outputs a detection model. This detection model can be used to differentiate between benign apps and malwares, and it is transmitted to the detection phase. In the detection phase, ICCDetector generates a feature vector for

TABLE II
ICC PATTERNS OF BENIGN APPS AND MALICIOUS APPS

ICC Patterns	Benign App	Malicious App
Component	Sufficient *Activities, Services, Providers*, and few *Receivers*	Few *Activities, Services, Providers*, and sufficient *Receivers*
Explicit Intent	Send to internal components	Send to internal components and external components
Implicit Intent	Send to internal components, and external components with `system action`	Send to internal components, external components with `system action` and `user defined action`
Intent Filter	Receive internal Implicit Intents, and few system-wide broadcasts	Receive internal Implicit Intents, various system-wide broadcasts and external Implicit Intents

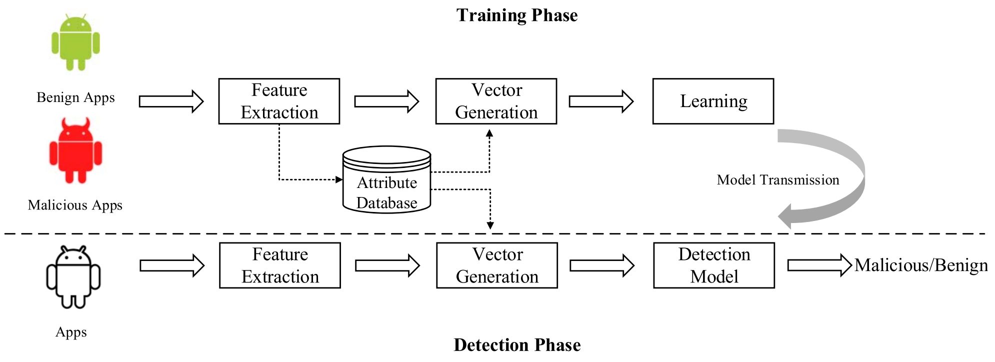

Fig. 3. ICCDetector System Architecture.

each app being detected and feeds the feature vector into the detection model, which outputs whether the detected app is benign or malicious.

A. Training Phase

Feature Extraction: In the first step of the training phase, ICCDetector extracts all of the ICC-related features from a given app. To achieve this, we develop a tool named Parser on top of any ICC analysis tool which outputs all ICC sources and sinks from the app's APK file. Examples of such ICC analysis tools include ComDroid [8], Amandroid [11] and EPICC [12]. We choose EPICC for Parser in this work. Parser defines various categories of ICC-related features, and formats of these features. Given an app's APK file, Parser extracts the ICC-related features for each category, and represents the extracted features in corresponding formats. ICC-related features are defined in the following four categories:

1) Components: Given an application, Parser extracts the names and types of its components. For *Broadcast Receiver-s*, Parser also records the registration modes (i.e., static or d-ynamic). After that, Parser represents the ICC-related features in the following format: `component_name(activity/s-ervice/provider)`, `component_name(receiver_static)`, `component _name(receiver_dynamic)`, `num_of_activity/ service/provider`, `num_of_receiver(static)`, `num_of_receiver(dynamic)`. For example, if an app dynamically registers a `Broadcast Rece-iver: com.bwx.bequick.receivers.Airplane ModeReceiver`, then Parser extracts an ICC-related feature `com.bwx.bequick.receivers.Airplane Mode-Receiver(receiver_dynamic)` from this app.

2) Explicit Intents: In this category, Parser records the total number of generated Explicit Intents and the number of external Explicit Intents. As explained in Section II, it is important to check the Explicit Intents' targets, including internal components and external components. Parser labels an Explicit Intent as `external` if it is sent to another app (i.e., the targeted package name is not included in the sender's APK file). The ICC-related features in this category are represented as `num_of_explicitintent`, `num_of_external_explicitintent`, `external_ package_name(external_explicitintent)`. For example, the package name of an app is `com.jx. theme`, which sends out an Explicit Intent to `package: com.android.packageinstaller`, `class: com. android.packageinstaller.PackageInstaller Activity`, then Parser retrieves an ICC-related feature as `com.android.packageinstaller(external _explicitintent)`. Note that, Parser does not record the package names of internal Explicit Intents. Since internal Explicit Intents are designed for intra-application communications, they are not very useful for detecting malwares.

3) Implicit Intents: For each Implicit Intent of an app, Parser matches it with the Intent Filters retrieved from the same app following the process defined by Android. If a match exists, Parser labels the Implicit Intent as `internal`, which is used to connect components within the same app. Otherwise, Parser regards this Implicit Intent as `external`,

TABLE III
SUMMARY OF ICC-RELATED FEATURES

Feature Category	Formats of ICC-related Features
Component	`component_name(activity/service/provider)`, `component_name(receiver_static)`, `component_name(receiver_dynamic)`, `num_of_activity/service/provider`, `num_of_receiver(static)`, `num_of_receiver(dynamic)`
Explicit Intent	`num_of_explicitintent`, `num_of_external_explicitintent`, `external_package_name(external_explicitintent)`
Implicit Intent	`num_of_implicitintent`, `num_of_internal_implicitintent`, `num_of_external_implicitintent(userdefined_action)`, `num_of_external_implicitintent(system_action)`, `action_string(internal)`, `action_string(external_system_action)`, `action_string(external_userdefined_action)`
Intent Filter	`num_of_intentfilter`, `num_of_intentfilter_for_activity/service`, `num_of_intentfilter_for_receiver(static)`, `action_string(for_receiver_dynamic)`, `num_of_intentfilter_for_receiver(dynamic)`, `action_string(for_activity/service)`, `action_string(for_receiver_static)`

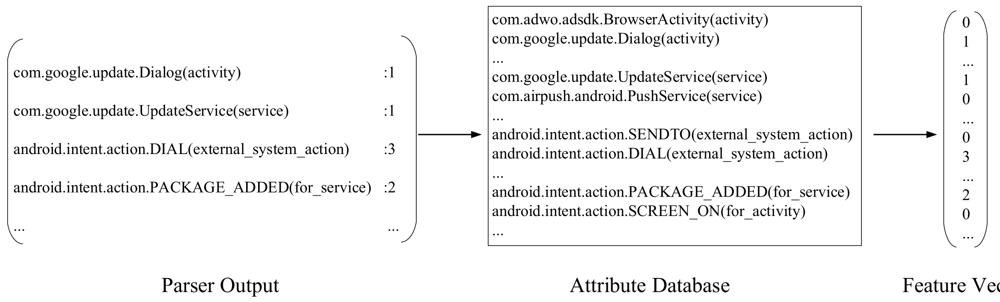
com.google.update.Dialog(activity) :1
com.google.update.UpdateService(service) :1
android.intent.action.DIAL(external_system_action) :3
android.intent.action.PACKAGE_ADDED(for_service) :2
... ...
com.adwo.adsdk.BrowserActivity(activity)
com.google.update.Dialog(activity)
...
com.google.update.UpdateService(service)
com.airpush.android.PushService(service)
...
android.intent.action.SENDTO(external_system_action)
android.intent.action.DIAL(external_system_action)
...
android.intent.action.PACKAGE_ADDED(for_service)
android.intent.action.SCREEN_ON(for_activity)
...
0 1 ... 1 0 ... 0 3 ... 2 0 ...
Parser Output
Attribute Database
Feature Vector

Fig. 4. Process of Feature Vector Generation.

and checks its action field. If the action is defined by other apps (i.e., the action is `user-defined` and its prefix is different from the sender's package name), Parser labels it as `external_userdefined_action`; otherwise, Parser labels it as `external_system_action`.

For each Implicit Intents, Parser retrieves its action string and identifies the potential target, and outputs the following ICC-related features: `num_of_implicitintent`, `num_of_internal_implicitintent`, `num_of_external_implicitintent(userdefined_action)`, `num_of_external_implicitintent(system_action)`, `action_string(internal)`, `action_string(external_userdefined_action)`, and `action_string(external_system_action)`. For instance, a malware (package name: `net.mujee.www`) generates an Implicit Intent with `android.intent.action.DIAL`. However, none of its Intent Filter is registered to receive this Intent. Parser retrieves an ICC-related feature `android.intent.action.DIAl(external_system_action)` in this case.

4) Intent Filters: Similar to Implicit Intents, Intent Filters are represented with the included action strings. In addition, Parser records the types of components which Intent Filters are registered for. Especially, if an Intent Filter is registered for a *Broadcast Receiver*, Parser checks whether the registration is dynamic or static. The ICC-related features in this category include `action_string(for_activity/service)`, `action_string(for_receiver_static)`, `action_string(for_receiver_dynamic)`, `num_of_intentfilter_for_activity/service`, `num_of_intentfilter_for_receiver(static)`, `num_of_intentfilter_for_receiver(dynamic)`, and `num_of_total_intentfilter`. For example, given an application which dynamically registers two Intent Filters with the s-ame action string (`android.provider.Telephony.SMS_RECEIVED`), Parser extracts an ICC-related feature `android.provider.Telephony.SMS_RECEIVED(for_receiver_dynamic)`, and sets its corresponding value to two.

Table III summarizes the formats of ICC-related features. For all the ICC-related features retrieved from benign apps and malwares by Parser, ICCDetector stores them separately in the *Attribute Database*. In addition, ICCDetector stores the output of Parser for each analyzed application, which includes the extracted ICC-related features and the corresponding values.

Feature Vector Generation: In the training phase, ICCDetector leverages any two-class classification method (e.g., SVM [13], Decision Tree [14] and Random Forest [15]) to learn the ICC patterns from benign apps and malwares, respectively. In particular, ICCDetector treats each of the extracted ICC-related features as a detection feature. Therefore, the number of detection features is equal to the size of *Attribute Database*. If the size of *Attribute Database* is X, ICCDetector defines an X-dimensional vector space. For each app, ICCDetector constructs a feature vector by mapping its Parser output to the X-dimensional vector space as shown in Fig. 4.

Usually, a typical Android app generates roughly 100 none-zero ICC-related features; therefore, its feature vector is sparse. ICCDetector represents the feature vectors sparsely using hash tables [16]. In comparison, existing works [2], [17] use Boolean expression to represent feature vectors, which indicates whether an app includes a feature or not. Since ICCDetector records the exact value for each ICC-related feature in generating feature vectors, ICCDetector has more accurate information to differentiate between benign apps and malicious apps.

Learning: As described in *Feature Extraction* and *Feature Vector Generation*, ICCDetector extracts ICC-related features as many as possible from given apps and constructs feature vectors by mapping Parser out to the vector space. Some of these extracted ICC-related features, however, are correlated to each other. Note that the number of extracted ICC-related features decides the dimensionality of feature vectors. If ICCDetector extracts too many ICC-related features in *Feature Extraction*, it leads to high dimensional feature vectors in *Feature Vector Generation*, which may contain a high degree of irrelevant and redundant information, and thus degrade the performance of learning algorithms. As a preprocessing step to machine learning, feature selection is effective in reducing dimensionality, removing irrelevant and redundant features, and mitigating overfitting. In this work, we apply a well-known feature selection method, Correlation-based Feature Selection (CFS) [18], in ICCDetector. CFS identifies and removes irrelevant and redundant features according to the correlation between features. After the process, CFS keeps a subset of original features, which are sufficient for the classification of Android applications. Consequently, CFS effectively reduces the dimensionality of feature vectors.

Given the input of reduced-dimension feature vectors generated from benign apps and malwares, respectively, ICCDetector applies any two-class classification method and outputs a detection model, which separates the feature vectors from benign and malicious. The detection model is then transmitted to detection phase.

B. Detection Phase

In the detection phase, ICCDetector extracts the ICC-related features from an app being detected, generated its feature vector, and feeds the feature vector to the detection model. The detection model decides whether the detected app is benign or malicious.

IV. Evaluation

We evaluate the performance of ICCDetector in different aspects with real data, including comparison with a benchmark, analysis of detection performance, classification of detected malwares, and runtime measurement.

A. Data Collection

We built an initial dataset of 14,264 benign apps by crawling GooglePlay from July 2014 to August 2014. To exclude potential malicious apps from this initial dataset, we sent each app to VirusTotal [19], which is an antivirus service with fifty-four antivirus scanners. We labeled an app in the original dataset as benign if and only if no antivirus scanner raises any alarm for the app. We also excluded potential malicious apps such as adwares, and spywares from the initial dataset so as to generate the benign dataset, which consists of 12,026 apps.

An existing malware dataset [17], which consists of 5,264 malwares is used as the ground truth in our evaluation. This malware set is one of the largest and newest datasets of Android malwares which are publicly available today.

B. Feature Selection and Analysis

From 12,026 benign apps and 5,264 malwares, ICCDetector extracts 121,621 ICC-related features in total. Since the extracted features contain redundant information due to correlation between features, we apply CFS to identify and remove the redundant features according to the correlation between features. After the process, CFS chooses 5,000 ICC-related features, which are subsequently used for the classification of malicious and benign applications.

A majority of the ICC-related features that are removed by CFS belongs to *Component*. When design names for components, app developers usually include package names in components names. For example, an application with package name `com.bwx.bequick` includes several components named as `com.bwx.bequick.EulaActivity`, `com.bwx.bequick.ShowSettingsActivity`, and `com.bwx.bequick.MainSettingsActivity`. Since it is common to include package names in component names in mobile application development, our ICCDetector extracts numerous unique ICC-related features from component names. We notice that a majority of ICC-related features in this category only appear once in single applications and are correlated to some other features. Consequently, CFS can effectively reduce the dimensionality of feature vectors after removing these redundant features.

While CFS filters out a majority of ICC-related features belonging to *Component*, it keeps the features in this category that are useful for distinguishing malwares from benign apps. For example, CFS keeps an ICC-related feature `com.allen.txtxcb.Settings(activity)`, which is extracted from an *Activity* named `com.allen.txtxcb.Setting`. This *Activity* can only be found in a malware family called DroidKungFu, and is shared by the malwares in this family. Another example is `com.android.installer.-full.AndroidInstaller2Activity(activity)`, which is an ICC-related feature selected by CFS. This feature, which is extracted from an *Activity* called `com.android.-installer.full.AndroidInstaller2Activity`, is shared by several Russian malwares. These malwares pretend to be legitimate package installers provided by Android, but are capable of manipulating SMS, taking pictures, and directly installing arbitrary applications. This *Activity* has a confusing name so as to fool end users and evade from detections, which is normal in malwares but rare in benign applications. It is thus helpful to keep these ICC-related features for differentiating between malwares and benign applications.

We also analyze the selected ICC-related features belonging to other categories. In general, a majority of the selected features can be used to describe the communications among components within or cross application boundaries. Due to the differences between malicious ICC patterns and benign ICC patterns, these selected features can be used to distinguish benign apps and malwares. For example, `android.intent.action.PACKAGE_CHANGED(external_system_action)` is a selected ICC-related features extracted from an external Implicit Intents `android.intent.action.PACKAGE_CHANGED`. This Implicit Intent is widely used by malwares for monitoring system events that are related to package and downloading additional APK files during runtime; however, it is barely generated by benign applications. In the Training phase, our model learns from the training dataset that this selected feature usually appears in malicious feature vectors but rarely appears in benign feature vectors. In the Detection phase, all of the selected ICC-related features are used to distinguish between malicious and benign applications.

C. Experiment Result

We compare the detection performance of ICCDetector with a benchmark, which is a highly cited Android malware detection method proposed in recent years [9], using the same dataset. The benchmark is a typical Android malware detection method, which detects malwares based on their required permissions and its accuracy is up to 88.2% using the dataset mentioned in Section IV-A.

In the experiments, ICCDetector leverages on a widely used two-class classification method, *Support Vector Machine (SVM)* [13], to train a detection model. *SVM* is suitable for processing multidimensional data like the feature vectors and capable of producing a model efficiently. Given the feature vectors of benign apps and malwares, *SVM* discovers the hyperplane to separate them with the maximum margin, where the margin is the sum of (i) the minimum distance between the hyperplane and the boundary of benign feature vectors, and (ii) the minimum distance between the hyperplane and the boundary of malicious feature vectors.

We conduct a series of experiments using *ten-fold cross validation* [20] to measure the performance of ICCDetector and the benchmark. In particular, we randomly split the benign dataset and the malicious dataset into ten subsets, respectively. The detection model is trained and tested in ten rounds. In each round, we mix one benign subset and one malicious subset as the testing dataset (i.e., unknown dataset), and the remaining subsets as the training dataset (i.e., known dataset). The testing dataset is tested using the classifier trained on the training dataset. In each round, there is no overlap between the testing dataset and the training dataset. Each application of the whole dataset is classified once so the accuracy of cross validation is the percentage of the applications that are correctly classified. We evaluate the performance of ICCDetector using three metrics, True Positive Rate (TPR), False Positive Rate (FPR), and Accuracy, where TPR is the percentage of malwares being detected correctly, FPR is the percentage of benign apps being detected as malwares, and Accuracy is the percentage of all apps being detected correctly in our experiments.

TABLE IV
EXPERIMENT RESULT

Metrics	True Positive Rate	False Positive Rate	Accuracy
ICCDetector	93.1%	0.67%	97.4%
Benchmark	65.0%	1.71%	88.2%

Table IV shows the evaluation results of ICCDetector and the benchmark. The accuracy of the benchmark is up to 88.2%, while ICCDetector achieves an accuracy of 97.4%, roughly 10% higher than the benchmark, with a lower false positive rate of 0.67%, which is only a half of the benchmark. Through manually analyzing detected false positives, we discover that only seven benign applications are falsely identified as malware. The true positive rate of ICCDetector is also considerably better than the benchmark, roughly 30% higher than the benchmark. More importantly, ICCDetector discovers 1,708 more "advanced malwares" than the benchmark (i.e., these malwares can only be detected by ICCDetector), while it misses 220 "obvious malwares" which can be easily detected by the benchmark.

D. True Positive Analysis

Looking into the 1,708 "advanced malwares" which are correctly detected by ICCDetector but not by the benchmark, we discover that the differences between their permission usage patterns and those of benign apps are not very significant. Table V shows the percentage of benign apps and malwares which require some sensitive permissions. Since the permission patterns are similar, it is difficult for the benchmark to distinguish between benign apps and malwares.

In comparison, the ICC patterns can be used to distinguish benign apps and malwares in such case. In general, benign apps mainly use ICC for internal communications, in a sense that Intents and Intent Filters are mainly used to link the components within the same apps. However, malwares tend to interact with external components and monitor Android system via the ICC mechanism. For example, benign apps barely register any Intent Filters for package-related information, while the malwares usually register several such Intent Filters in order to properly download APK files at runtime.

One example of the "advanced malwares" is `com.safesys.viruskiller`, which pretends to be antivirus app in markets. Without requiring any sensitive permissions, `com.safesys.viruskiller` downloads APK files by generating external Implicit Intents and monitoring the system events related to `package`, such as `android.intent.action.PACKAGE_ADDED` and `android.intent.action.PACKAGE_CHANGED`. Another example is `com.accutracking`, which is a malware intercepting private information and accessing personal files. One characteristic of `com.accutracking` is its rich variants. Although its variants are given different package names, they share the same ICC-related features and same

TABLE V
PERCENTAGE OF BENIGN APPS AND MALICIOUS APPS REQUIRING SENSITIVE PERMISSIONS

Permission	**Benign App% (12,026)**	**Malicious App% (5,264)**
android.permission.INTERNET	92.5%	97.5%
android.permission.ACCESS_NETWORK_STATE	81.4%	67.4%
android.permission.ACCESS_COARSE_LOCATION	23.6%	32.9%
android.permission.CAMERA	14.2%	4.2%
android.permission.CALL_PHONE	11.3%	13.4%

ICC patterns, which can be easily detected by ICCDetector. These "advanced malwares" are still available in alternative markets, such as *kekaku* [21], *coolAPK* [22], *appchina* [23], and *amazon* [24].

E. False Negative Analysis

ICCDetector misses 364 malwares, while 220 of them can be easily detected by the benchmark. After manually checking how these malwares use ICC and permissions, we discover that these malwares barely use ICC. Instead of stealthily conducting malicious actions, these malwares attack in a straightforward way by simply requiring a bunch of sensitive permissions, and sometimes even permissions not for use by third-party apps. For example, with only four non-zero ICC-related features, a malware requires three permissions `android.permission.READ_LOGS`, `android.permission.INSTALL_PACKAGES`, and `android.permission.MODIFY_PHONE_STATE`, which should not be used by third-party apps. Since benign apps merely require such system-level permissions, the benchmark can easily detect such malwares based on their required permissions.

It is obvious that ICCDetector and the benchmark are complementary. A hybrid approach combining ICCDetector and the benchmark would produce better results.

F. False Positive Analysis

In the experiment, ICCDetector labels 81 benign applications as malicious (i.e., false positives). After manually analyzing these false positives, we discover that 31 of them were mislabeled by VirusTotal before. Besides them, 43 of other false positives are manually identified as malicious because they are capable of conducting malicious actions. In the end, the false positives of ICCDetector boil down to seven benign applications falsely classified as malware. We classify the false positives into three categories as follows:

Mislabeled by VirusTotal: In order to construct benign dataset, we excluded potential malwares by sending each app in the original dataset to VirusTotal, and labeled an app as benign if and only if no antivirus scanner raises any alarm for the app. However, the detection result of VirusTotal should be updated and corrected over time. After resending the 81 false positives to VirusTotal, we discover that 31 applications, which had been labeled as benign when constructing benign dataset in August 2014, received alarms from at least one antivirus scanners in May 2015. For these 31 applications, the detection results of ICCDetector and those of the updated version of VirusTotal are consistent.

Since VirusTotal has not released any technical details related to its updating process, it remains unknown that how these 31 malwares bypassed the scanning of VirusTotal in August 2014. Fortunately, these malwares can be easily detected by ICCDetector according to their ICC patterns. For example, `com.tobyyaa.superbattery` is an application which includes several malicious components, such as `com.millenialmedia.-`, `com.admob.-`, `com.flurry.-` and `com.appbrain.-`. This application is capable of manipulating SMS, and making phone calls by generating suspicious Intents such as `android.intent.action.DIAL`, and `android.intent.action.CALL`, and certain Intent Filters such as `android.intent.action.NEW_OUTGOING_CALL`, and `android.provider.Telephony.SMS_RECEIVED`. These suspicious ICC patterns can be captured by ICCDetector in malware detection.

New Identified Malwares: Not only can ICCDetector product consistent results with the updated version of VirusTotal, but also can identify new malwares. After manually analyzing the 81 false positives, 43 of them are identified as malicious, which have not been identified by VirusTotal before. Most of these newly identified malwares are capable of leaking private information, manipulating SMS, connecting to remote servers, and monitoring system state. For example, `com.tunewiki.lyricplayer.android.quicklaunch` is a newly identified malware discovered in GooglePlay which is designed to make phone calls, monitor and leak phone states and system settings. To achieve its goal, this application generates an Intent `android.intent.action.DIAL` to make phone call, and registers several Intent Filters to monitor any phone state and setting changes.

During the manual analysis, we discover that these newly identified malwares not only manipulate the ICC mechanism, but also abuse sensitive permissions. For instance, `com.thukhakyaw.calllocator` is a newly identified malware which may make phone calls, send out SMS, and modify system states. In particular, this malware requires several permissions which are not allowed to use by any third-party applications, such as `android.permission.MODIFY_PHONE_STATE` and `android.permission.UPDATE_DEVICE_STATS`. This discovery further demonstrates the accuracy of ICCDetector.

Benign Applications: Among the 81 false positives, seven benign applications are falsely identified as malware. After manually analyzing these benign applications, we discover that they barely use any ICC mechanism, therefore it is difficult for ICCDetector to correctly identify them.

G. Classifications

The classifications of malwares in different malware families facilitate better understanding and analyzing of malwares [2], [17]. On the other hand, some of the existing classifications have the following limitations:

- Some malware families are named by different mobile security software vendors and researchers. The naming scheme is confusing and inconsistent. For example, *BaseBridge* is also named as *AdSMS*, and *LeNa* is a variant of *DroidKungFu* [25].
- Some malwares belonging to different families have similar malicious behaviors. For example, the malwares in *Lovetrap* and *NickyBot* are similar in terms of sending premium SMSes and starting malicious services right after Android system boot-up.
- Some malware families contain too few samples. For example, *FakeInstaller* contains about 1,000 malicious samples, while *GGTracker, DroidCoupon* and *GamblerSMS* only have one malicious sample. More importantly, the majority of existing malware families (i.e., more than 200 malware families) contain less than thirty samples per family.

Motivated to overcome such limitations, we propose five new malware categories based on ICC patterns and classify detected malwares into corresponding categories. In order to conduct certain malicious operations, malwares need to use the ICC mechanism accordingly. Therefore, these newly defined malware categories are closely related to malware behaviors.

Server Connector: Malwares in this category mainly conduct malicious actions by connecting to command and control servers, dynamically downloading and installing APK files, and executing remote commands. These malwares usually register several *Broadcast Receivers* and *Services* to receive `c2dm` (Cloud to Device Messaging [26]) related Intents and to execute received commands. Especially, in order to effectively download and install APK files from the remote servers, these malwares leverage Intent Filters to monitor events related to `package`, such as `android.intent.action.PACKAGE_CHANGED`, `android.intent.action.PACKAGE_ADDED`, and `android.intent.action.PACKAGE_REMOVED`. Several wellknown malware families, including `DroidKungFu`, `DroidRooter`, `RootSmart` and `ExploitLinuxLotor`, belong to this category.

Telephony Abuser: This category includes malwares which conduct attacks targeting at telephonic functionalities, such as making phone calls, blocking incoming phone calls and SMSes, and sending SMSes to premium numbers. To effectively manipulate telephonic functionalities, malwares need to generate corresponding Intents, such as `android.intent.action.DIAL`, and register certain Intent Filters to intercept SMS-related information and new outgoing calls, such as `android.provider.telephony.SMS_RECEIVED` and `android.intent.action.NEW_OUTGOING_CALL`. M-alware families in this category include `Opfake`, `Dialer`, `MobileSpy`, and etc.

System Monitor: Malwares in this category especially care about system-wide broadcast information that is relevant to phone states and settings, such as battery state, power state, and connectivity setting. From phone states and settings, these malwares can infer whether a phone is in use or not, and pick the appropriate time to perform malicious actions without user's awareness. To achieve their malicious objectives, these malwares tend to register several Intent Filters to capture broadcasts with special actions, such as `android.settings.SIG_STR`, `android.net.conn.CONNECTIVITY_CHANGE`, `android.intent.action.POWER_CONNECTED` and `android.intent.action.PHONE_STATE`. Moreover, some malwares in this category generate external Intents so as to change phone states and settings.

Effective Launcher: This category contains malwares which leverage a special system-wide Intent with action `android.intent.action.BOOT_COMPLETED` to effectively launch their malicious *Activities* and *Services* when the Android system completes its booting process. Moreover, some malwares in this category can immediately bootstrap their *Services* before starting the host app's primary *Activity* by intercepting an Intent with action `android.intent.action.MAIN`.

Advertiser: Instead of including dangerous and sensitive Intent Filters or Intents, malwares in this category usually include more than one advertisement libraries, which are mainly used by malwares. Such libraries include `Airpush`, `LeadBolt`, `Appenda` and `SendDroid`.

Table VI summarizes the categories we defined according to ICC characteristics, which demonstrates similar malicious behaviors within each category.

We also looked into the 1,708 "advanced malwares" which can be detected by ICCDetector but not the benchmark, and the 43 newly identified malwares detected from false positives. We discovered that most of them belong to *Server Connector, Telephony Abuser* and *System Monitor*, which are more dangerous than the other two categories.

H. Runtime Measurement

We ran our experiments on a machine with 4×3.20GHz Intel-Core and 12 GB of RAM, and measured the runtime of ICCDetector. In each of ten rounds in our experiments, ICCDetector is trained with 15,561 applications (i.e., 90% of datasets), and tested with a mix of 1,203 benign apps and 526 malwares. In the training phase, an ICC analysis tool, EPICC, is used to analyze the APK file of each app, which outputs all ICC sources and sinks. Our Parser is then used to extract all ICC-related features, including their names and values. After processing all the apps in training dataset, and storing all ICC-related features in the *Attribute Database*, ICCDetector generates a feature vector for each processed app. ICCDetector outputs a *SVM* detection model given all feature vectors in the training phase. In the detection phase, each app is processed using APK analysis, feature generation, and vector generation as in the training phase. In addition, the detection model labels an app being detected as "benign" or "malicious" based on its feature vector.

TABLE VI
MALWARE CATEGORIES BASED ON ICC PATTERNS

Category	Num of Apps	Malicious Behaviors	ICC Characteristics	Example of Malware Family
Server Connector	1258	Connect to command and control servers, dynamically install APK files, and execute remote commands	Register Intent Filters for package-related information, register *Broadcast Receivers* for `c2dm`, and register *Services* for executing commands	DroidKungFu, DroidRooter, RootSmart, ExploitLinuxLotoor
Telephony Abuser	2270	Conduct attacks aiming at telephonic functionalities, such as making phone calls and sending SMS	Register *Services* for controlling SMS, and register Intent Filters for monitoring SMS-related information and new-outgoing calls	Opfake, MobileSpy, SendPay, Dialer, SMSreg
System Monitor	348	Monitor system-wide broadcasts relevant to phone state changes	Register Intent Filters for monitoring phone states, such as battery states and power states, and generate external Intents to change connectivity	AccuTrack, Anti, Gamex, SafeKidZone
Effective Launcher	1016	Leverage ICC mechanism to effectively launch malicious *Services* and *Activities*	Register Intent Filters and Intents with `android.intent.action.BOOT_COMPLETED` and `android.intent.action.MAIN`	Boxer, Fakelogo, Jifake, Iconeyss, Adrd
Advertiser	8	Include more than one malicious advertisement libraries	Register *Services* and *Activities* related to advertisement libraries, such as *Airpush, LeadBolt, Appenda, SendDroid.*	Opfake, Fidall, Stiniter, Kidlogger

TABLE VII
TIME FOR PROCESSING AN APP

Step	APK Analysis	Feature Extraction	Vector Generation	Model Detection
Average	36.00s	2.6×10^{-3}s	0.79s	1.2×10^{-3}s

Table VII shows the average time for processing each app in our experiments. The performance bottleneck is at the APK analysis. In the future, the performance of ICCDetector would be improved with the development of more efficient ICC analysis tools.

V. DISCUSSIONS

Besides the benchmark, ICCDetector is also compared with some recent malware detection methods, including Drebin [17], and DroidMiner [2]. Since the source codes and datasets of Drebin and DroidMiner are not open to the public, we provide our comparison qualitatively.

Drebin [17] is a lightweight malware detection method directly working on smartphones. Due to the limited resources of mobile devices, Drebin conducts a broad static analysis to gather many detection features, including permissions, APIs, network addresses, app component names, and Intent Filters extracted from manifest files. In comparison, ICCDetector extracts more ICC-related features, including the number of Intents, the names and actions of each Intent, the potential internal and external receivers of each Intent, and the Intent Filters extracted from bytecode. Therefore, ICCDetector is more accurate in capturing ICC-related features and patterns in malware detection.

On the other hand, DroidMiner [2] detects malwares based on not only the frequency and names of sensitive APIs, but also the connections of multiple sensitive APIs. Unlike ICCDetector, DroidMiner does not inspect any ICC-related functions. Therefore, it is less effective to capture the communications and interactions between components within or cross application boundaries.

Limitations: Lacking dynamic inspection of malware behaviors, ICCDetector may be bypassed by malwares using Java reflection and bytecode encryption [27]. This encourages us to incorporate dynamic analysis in future versions of ICCDetector. Another limitation of ICCDetector, which is due to the use of classification methods, is its vulnerability to mimicry and pollution attacks [28], where malwares may include more benign features and poison the training dataset to lower their suspicions.

VI. RELATED WORKS

Mobile Malware Detection: Static malware detection methods analyze app codes and manifest files without running the apps. For instance, Kirin [1] detects malwares based on the permissions required by the Android apps which break certain pre-defined security rules. Stowaway [29] detects overprivileges in Android apps by mapping API calls to permissions. Peng et al. [9] proposed a malware detection model based on app categories and declared permissions. RiskRanker [30] captures risky apps based on known malicious behaviors and existing vulnerabilities in Android, and detects malwares from risky apps based on manual efforts. DroidMiner [2] and DroidAPIMiner [3] use sensitive API calls in detecting malwares, while DroidMat [31] and Drebin [17] use not only sensitive API calls but also other information extracted from manifest files as detection features. These previous works capture the communications between apps and Android system based on the required resources of detected apps, while ICCDetector captures not only the communications between apps and system, but also the interactions among apps based on ICC-related features.

Another class of malware detection, including TaintDroid [32], DroidScope [33], CrowDroid [34], Paranoid Android [35], and DroidRanger [36], employs dynamic analysis to detect malwares at runtime. These dynamic approaches are complementary to the static analysis based approaches, including ICCDetector.

ICC Analysis: Much work has been done on ICC analysis. For example, ComDroid [8] investigates the attack surfaces related to ICC. CHEX [37] focuses on detecting *Component Hijacking* attacks by analyzing information flows. AppSealer [38] generates vulnerability-related patches for preventing *Component Hijacking* attacks. Epicc [12] is a static analysis tool for identify ICC precisely and scalably. Amandroid [11] conducts static analysis for security vetting of Android apps based on inter-component control and data flows. Pscout [39] produces a permission specification, which is a set of mappings between API calls (including ICC APIs) and permissions. These works focus on identifying ICC-related attack surfaces for Android apps, while ICCDetector focuses on detecting malwares based on ICC-related features. The ICC analysis tools developed in these works can be applied by ICCDetector in constructing its Parser.

VII. Conclusion

ICCDetector detects malwares based on ICC-related features which capture the interaction between components within or cross application boundaries. The performance of ICCDetector is better than the benchmark in our experiments. The malwares detected by ICCDetector are classified into five new malware categories according to their ICC characteristics, which clarifies the relationship between malware behaviors and ICC patterns. Furthermore, after manually analyzing false positives, we discover 43 new malwares from the benign dataset. In the future, we plan to apply ICCDetector to detect new malwares in various application markets. We also plan to build a dataset which can be used to evaluate and compare different malware detection methods on a common platform.

References

[1] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. 16th ACM Conf. Comput. Commun. Secur.*, 2009, pp. 235–245.

[2] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications," in *Proc. 19th Eur. Symp. Res. Comput. Secur. (ESORICS)*, Wroclaw, Poland, Sep. 2014, pp. 163–182. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11203-9-10

[3] Y. Aafer, W. Du, and H. Yin, "DroidAPIMiner: Mining API-level features for robust malware detection in Android," in *Proc. 9th Int. ICST Conf. Secur. Privacy Commun. Netw. (SecureComm)*, Sydney, NSW, Australia, Sep. 2013, pp. 86–103. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-04283-1-6

[4] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on Android," in *Proc. 13th Int. Conf. Inf. Secur. (ISC)*, Boca Raton, FL, USA, Oct. 2011, pp. 346–360. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-18178-8-30

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

[6] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Proc. USENIX Secur. Symp.*, 2011, pp. 1–16.

[7] R. Schlegel, K. Zhang, X.-Y. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A stealthy and context-aware sound trojan for smartphones," in *Proc. NDSS*, vol. 11. 2011, pp. 17–33.

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. 9th Int. Conf. Mobile Syst., Appl., Services*, 2011, pp. 239–252.

[9] H. Peng *et al.*, "Using probabilistic generative models for ranking risks of Android apps," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 241–252.

[10] *Android*. [Online]. Available: http://developer.android.com/guide/components/intents-filters/

[11] F. Wei, S. Roy, X. Ou, and R., "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1329–1341.

[12] D. Octeau *et al.*, "Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis," in *Proc. 22nd USENIX Secur. Symp.*, 2013, pp. 543–558.

[13] C. J. C. Burges, "A tutorial on support vector machines for pattern recognition," *Data Mining Knowl. Discovery*, vol. 2, no. 2, pp. 121–167, 1998.

[14] D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Trans. Syst., Man, Cybern.*, vol. 21, no. 3, pp. 660–674, May/Jun. 1991.

[15] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R News*, vol. 2, no. 3, pp. 18–22, 2002.

[16] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[17] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proc. NDSS*, 2014, pp. 1–15.

[18] M. A. Hall, "Correlation-based feature selection for machine learning," Ph.D. dissertation, Dept. Comput. Sci., Univ. Waikato, Hamilton, New Zealand, 1999.

[19] *VirusTotal*. [Online]. Available: https://www.virustotal.com/

[20] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proc. IJCAI*, vol. 14. 1995, pp. 1137–1145.

[21] *Kekaku*. [Online]. Available: http://m.kekaku.com/

[22] *CoolAPK*. [Online]. Available: http://www.coolapk.com/apk/com.coolapk.market

[23] *Appchina*. [Online]. Available: http://www.appchina.com/

[24] *Amamon*. [Online]. Available: http://www.amazon.com/gp/feature.html?docId=1000625601

[25] Y. Zhou and X. Jiang, "Dissecting Android malware: Characterization and evolution," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2012, pp. 95–109.

[26] *Google*. [Online]. Available: https://developers.google.com/android/c2dm/

[27] V. Rastogi, Y. Chen, and X. Jiang, "DroidChameleon: Evaluating Android anti-malware against transformation attacks," in *Proc. 8th ACM SIGSAC Symp. Inf., Comput. Commun. Secur.*, 2013, pp. 329–334.

[28] S. Venkataraman, A. Blum, and D. Song, "Limits of learning-based signature generation with adversaries," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, Feb. 2008. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/08/papers/18limits learning-based.pdf

[29] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, 2011, pp. 627–638.

[30] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day Android malware detection," in *Proc. 10th Int. Conf. Mobile Syst., Appl., Services*, 2012, pp. 281–294.

[31] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: Android malware detection through manifest and API calls tracing," in *Proc. 7th Asia Joint Conf. Inf. Secur. (Asia JCIS)*, Aug. 2012, pp. 62–69.

[32] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, p. 5, Jun. 2014.

[33] L.-K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. USENIX Secur. Symp.*, 2012, pp. 569–584.

[34] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. 1st ACM workshop Secur. Privacy Smartphones Mobile Devices*, 2011, pp. 15–26.

[35] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, “Paranoid Android: Versatile protection for smartphones,” in *Proc. 26th Annu. Comput. Secur. Appl. Conf.*, 2010, pp. 347–356.

[36] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, “Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets,” in *Proc. NDSS*, 2012, pp. 1–13.

[37] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “CHEX: Statically vetting Android apps for component hijacking vulnerabilities,” in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 229–240.

[38] M. Zhang and H. Yin, “Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in Android applications,” in *Proc. 21th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2014, pp. 1–15.

[39] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, “Pscout: Analyzing the Android permission specification,” in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 217–228.