Meng et al. *Cybersecurity* (2018) 1:4
https://doi.org/10.1186/s42400-018-0006-7

Cybersecurity

RESEARCH Open Access

DroidEcho: an in-depth dissection of malicious behaviors in Android applications

Guozhu Meng[1,2*], Ruitao Feng[2], Guangdong Bai[3], Kai Chen[1,4] and Yang Liu[2]

Abstract

A precise representation for attacks can benefit the detection of malware in both accuracy and efficiency. However, it is still far from expectation to describe attacks precisely on the Android platform. In addition, new features on Android, such as communication mechanisms, introduce new challenges and difficulties for attack detection. In this paper, we propose abstract attack models to precisely capture the semantics of various Android attacks, which include the corresponding targets, involved behaviors as well as their execution dependency. Meanwhile, we construct a novel graph-based model called the inter-component communication graph (ICCG) to describe the internal control flows and inter-component communications of applications. The models take into account more communication channel with a maximized preservation of their program logics. With the guidance of the attack models, we propose a static searching approach to detect attacks hidden in ICCG. To reduce false positive rate, we introduce an additional dynamic confirmation step to check whether the detected attacks are false alarms. Experiments show that DroidEcho can detect attacks in both benchmark and real-world applications effectively and efficiently with a precision of 89.5%.

Keywords: Semantic attack model, Android malware detection, Inter-component communication graph, Privacy leakage

Introduction

Nowadays, Android malware detection is facing two critical challenges: 1) how to design a precise and efficient model to represent malware; 2) how to reduce false alarms and distinguish real malware from benign applications. Android malware varies in many aspects such as attack targets, attack methods, and applied obfuscation techniques. For example, Android malware may steal users' sensitive information (Grace et al. 2012; Arzt et al. 2014a), elevate their privilege (Xing et al. 2014; Gunadi and Tiu 2013), deplete device resources (Vekris et al. 2012; Pathak et al. 2012), and remote control users' devices (Zhou and Jiang 2012). Malware may accomplish attack missions either individually or collaboratively (Octeau et al. 2013; Bosu et al. 2017), perform attacks only once or periodically (Zhou and Jiang 2012), and be triggered by the installation or a broadcast message. In addition, malware may adopt several mechanisms to bypass the detect ion of security analysts and antivirus software, such as ProGuard (ProGuard 2017) and reflection (Zhou and Jiang 2011). All of these raised challenges for the existing detection approaches to reach a desirable precision and scalability simultaneously.

On the other hand, it is challenging to eliminate greyware from malware (Symantec Inc. 2017), especially when they are requesting privileged permissions for accomplishing specific functionalities. For instance, WeChat, one of the top-ranked applications in Google Play, requests permissions of reading SMS messages and accessing network simultaneously. It may raise the concern of security analysts since it is speculated as a potentially malicious behavior which sends SMS messages out to the network. However, the fact is that it only reads the SMS messages from its remote server for the two-factor authentication use. Similar cases are pervasive on Android: weather applications show the weather situation and forecast to users, and thereby, need to read and send out users' location information; social applications may ask for users' contacts to find friends quickly; fitness applications sometimes access the sensors in order to measure users' exercise. Therefore, the detection based

*Correspondence: mengguozhu@gmail.com
[1]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]Nanyang Technological University, Singapore, Singapore
Full list of author information is available at the end of the article

Springer Open

© The Author(s). 2018 **Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

on an imprecise and coarse-grained malicious behavior model would lead to a high false positive rate.

Even with a precise model of malicious behaviors, malware searching in applications with static approaches is not easy. New execution paradigm, system libraries and rich communication features provided by Android have facilitated the development of rich-functionality applications. On the other hand, however, they also make static analysis of application more complicated and difficult, which are summarized below.

- **Implicit Execution Sequence.** Android framework provides a variety of program execution environments, callbacks and control frameworks for each Android component[1]. It is known as *lifecycle*. For example, after an activity is started by the system, it will execute the methods `onCreate()`, `onStart()` and `onResume()` in proper order, which cannot be observed from the application code;
- **Various Triggers for an Application.** There are many ways for an application to interact with the external environment. The application can be triggered or impacted by users' GUI operations (e.g., clicking a button). It can register a broadcast receiver to respond once a broadcast message arrives. In addition, local sensors can drive the application to run in a pre-defined way. On the other hand, an application can be started and driven via remote messages, such as Google Cloud Messaging (GCM), HTTP response, and an incoming SMS or phone call;
- **Complicated Communication Mechanisms.** Although each application is running in a separated sandbox, Android provides them various ways to communicate with each other. For instance, the Intent model (Octeau et al. 2013) is the most compelling method for component communication. Additionally, applications can define bound services, for example, an AIDL (Android Interface Definition Language) interface, and implement a *Binder* or a *Messenger* to accomplish the communication even between different processes or applications.

To overcome the above challenges, we propose an integrated framework called DroidEcho to analyze Android applications. First, we summarize the features of attacks happening on the Android platform, and propose a novel attack model. The model illustrates a variety of attack types at an abstract level, which is platform-independent. In particular, an attack is composed of: *assets*, which are the targets of attacks; *actions*, the execution operations performed on assets, and *triggers*, of which one entrance to the app that leads to the attack behaviors. Then we specialize the attack model into attack instances which are close to the Android platform, and can be utilized to guide our detection of attacks in a precise way.

Meanwhile, we transform Android applications into a comprehensive graph, incorporating call graphs between methods, and control flow graphs as per method. We conduct an in-depth static analysis through the graph with the guidance of attack model, and generate a full path with the trigger and the predicates that guarantee the occurrence of these behaviors. The detected malicious behaviors will be filtered by two conditions: if a seemingly malicious behavior is triggered by the user, it is likely that the behavior is user-intended, which we regard it as being harmless; presence of suspicious behaviors does not mean there is a real attack. It happens because some applications indeed need to carry out several seeming "malicious" behaviors to fulfill their tasks with good purposes. This is learnt and induced by investigating a group of applications under the same category or being similar. We make use of the mined social knowledge to filter out these harmless behaviors with a high level of confidence, i.e., these behaviors are likely a necessary part for applications. It does not only facilitate the efficiency of detection, but also reduce false positive in practice.

After the identification of malicious behaviors, we propose an approach to confirm the detected attacks with the dynamic execution. Our dynamic analysis is driven by the attack traces generated previously, and provides a satisfied condition to guarantee the program to proceed along the trace. The dynamic execution reproduces the occurrence of attacks, and makes the attack detection more precise.

Different from the existing research on static analysis based approaches (Arzt et al. 2014a; Arzt and Bodden 2016; Xu et al. 2016; Wei et al. 2014), our work starts from the comprehension of Android malware by constructing semantic models. To reduce the false positive rate, we propose an approach to confirm attacks complying with the identified executed traces. To sum up, we make the following contributions:

- **Attack model** We propose a novel representation, to characterize malicious behaviors. An attack in the model is constituted of target assets, execution actions, triggers, execution flows and apps' declaimers. It can facilitate the understanding of the essential features of attacks, and the detection of malware.
- **Accurate attack detection approach** We propose a richly descriptive representation, named ICCG, to depict an Android application, with a maximal preservation of information. Based on ICCG, we design a synthetic approach to identify a malicious application by considering both the engineering aspect and the social aspect. A reduced but sufficient static analysis is to prove the presence of suspicious

behaviors, then confirmed with the help of the learnt social knowledge.

- **Attack Confirmation** After the identification of malicious behaviors, we conduct a confirmation process to prove the existence of a real attack with dynamic execution. The dynamic execution is fed with the traces of malicious behaviors generated by DROIDECHO, and further identifies the satisfiable conditions. Then it drives the application to execute along the traces, and thereby reproduces the attacks for confirmation.
- **Evaluation** We have evaluated DROIDECHO on the malware benchmarks (i.e., GENOME and DROIDBENCH), and 7,643 real world applications. It shows that DROIDECHO outperforms the state-of-the-art tool. Moreover, we have found out 444 applications with malicious behaviors in Google Play, and have a competitive edge in precision of 89.5% to the counterpart approaches and tools.

Organization

Section Semantic model of attack proposes abstract models for various attacks in Android. Section The inter-component communication graph describes a representation for Android applications. Section System design of DroidEcho presents our approach in malware detection. Section Evaluation gives a comprehensive evaluation for our approach. Section Discussion discusses the experiments and limitations of our approach. Section Related work summarizes summarizes relevant literatures, and Section Conclusion concludes this work.

Semantic model of attack

In this section, we first give an in-depth discussion on the attacks happening on the Android platform, and then provide a formal description of these attacks.

Building blocks

An attack on the Android platform has its unique features and characteristics. It has a variety of attack targets, and includes a sequence of actions that often leverage the APIs provided by Android. In order to depict these elements of an attack, we start with introducing the building elements of attacks and their representative examples on Android, in order to construct a general and formal definition of attacks.

Assets

Assets are referred to hardware, software and information on Android devices, which are the targets of attacks. For example, contact information is an important asset, which attackers aim to steal and make use of for malicious purposes; front light is a battery-consuming hardware such that some malicious applications may acquire it without releasing to exhaust battery quickly. On the Android platform, all the assets we concern about can be accessed by invoking certain system APIs. We list the representative examples of these assets on Android as follows.

- **Information Assets**: Identity code, Contact, SMS messages, File system, Location, System setting, etc.
- **Software Assets**: Phone service, SMS service, Package Manager, Download Manager, Broadcast service, etc.
- **Hardware Assets**: Camera, Media, Sensor, etc.

Actions

An attack action is an operation performing on a certain asset with the purpose of acquisition, tampering and interception, e.g., to fetch the IMEI code of the mobile phone.

Category According to the type of the target assets, actions can be categorized into several classes. For example, an action can acquire, edit, or delete some information stored on device; invoke, interrupt or stop a service provided by Android; and occupy or release a hardware resource. Therefore, the semantics of actions can be uniquely specified by the association of the action type and the target assets. In addition, there is a unique kind of actions on Android which are used for communication (see Section The inter-component communication graph for more details). Within communication, there must be at least one sender and one receiver, and the communication can occur between an application and the external environment, or between two components in one application. As a result, we summarize four actions related to communication in the scope of application. Table 1 shows the categories of actions covered in this paper.

Parametrization An action is often implemented by invoking a set of system APIs. These APIs are organized with a certain dependency relationship. For example, the action of retrieving data stored in a content provider can be described as: obtaining an instance of `ContentResolver`; specifying the URI of the target asset; and retrieving the data stored in this content provider. Every action of retrieving data in content provider follows the above processes. And we provide more details about this in Section Action recognition.

As a functional unit in the attack model, an action usually has an input, an output or both. Let α be an action, and β be an asset, then $\alpha(\beta)$ denotes the input of the action α is the asset β, and $\alpha \overset{\beta}{\dashrightarrow}$ denotes the output of the action α is the asset β (refer to Section Flows). A variety of concrete actions are derived from parameterizing these actions with assets. For instance, when acquiring the content of a content provider, we can specify some assets as the target, such as

Table 1 The category of actions on Android

Category	Operation	Action Example	Corresponding Implementation
Information-based	acquire	get SMS message	ContentResolver.query(Inbox)
	insert	insert a contact	ContentResolver.insert(Contact)
	edit	change system setting	Wallpaper.setBitmap(Image)
	delete	delete local files	File.delete()
Software-based	invoke	call a number	startActivity(Intent{tel:num})
	interrupt	block SMS messages	abortBroadcast()
	stop	uninstall an app	startActivity(Intent{pkg:app})
Hardware-based	occupy	hold the wakelock	WakeLock.aquire()
	release	release the wakelock	WakeLock.release()
Communication	e_send	send data to environment	sendTextMessage(SMS)
	e_recv	receive data from environment	getMessagesFromIntent(Intent)
	i_send	send data to other component	startService(Intent)
	i_recv	receive data from other component	getIntent(Intent)

`ContactsContract.Contacts.CONTENT_URI` and `CalendarContract.Events`. As a consequence, two actions are generated to fetch the contact list and events in the calendar, respectively. Table 1 list 9 basic kinds of actions, based on which more actions can be generated by parametrization with explicit target assets.

Triggers

Triggers are events which are taken as inputs to an application and lead to the occurrence of a behavior. Although triggers, which occur during runtime, are unpredictable for applications, the application can provide handlers to subscribe and capture these triggers. Once the application receives a subscribed trigger, it will go into the life cycle and execute specific methods. In light of the awareness of users, we present two sorts of triggers in the following:

- **User Interaction**. This kind of triggers are usually GUI-related, which are visible to the operating users. For example, when the user clicks a button drawn on the screen, the behavior is triggered and starts to execute. From this, the user can learn that the behavior is caused by his/her click operation, and we call it user-awareness. For simplicity, we assume that users can know the behaviors from the context which the user interaction causes.
- **Environmental Inputs**. There is another kind of triggers which can drive the execution of an Android application. The trigger could be the initialization of the application, a broadcast message or registered listeners to sensors. The whole process is free from the involvement of the user, which means that the user is likely unware of the execution of behaviors. As a consequence, we classify malicious behaviors triggered by environmental input as potential attacks for a further analysis.

As suggested by (Yang et al. 2013; Chen et al. 2013), behaviors that would never been executed until they are triggered by the user interaction reflect the "intention" of the user. Therefore, in this work, we assume that user interactions will not trigger any malicious behaviors, i.e., potential attacks that are triggered by user interactions are false positive. However, environmental input triggers can proceed stealthily, preventing users from knowing them. This kind of triggers usually bring in many security risks, which are our main concern in this paper.

Since triggers are external objects that cause the execution of attacks, we can instead recognize *e_recv* (see Table 1) to observe the arrivals of triggers. Specifically, the listeners can be categorized in terms of types of triggers. For example, `onClick(View)`, `onDrag(View,...)` and `onKey(View,...)` are the entry points of program when a user interaction trigger comes. While `onCreate()` and `onReceive()` are the entry points for the boot of applications and a broadcast message, respectively, which are regarded as environmental inputs.

Flows

Actions have a flow relationship in between. It is a kind of dependency relationship which is either directional or contextual. The directional relationship indicates the certain order of execution, which has been defined in the program logic for a specific task[2]; and the contextual relationship can be described as a semantic connection between two actions, for example, the input of an action is the output of the other action[3]. Generally, the contextual

relationship needs a transition of the negotiated data from one participant to the other.

A flow can exist between the environment and an action, and triggers are their negotiated data between them. Take an incoming SMS message for example, if an application registers a *BroadcastReceiver* for SMS messages, once an incoming SMS message arrives, the application will start to execute from the listener, and it can also get the content of the message as input. Therefore, there exists a directional and contextual relationship between the environment and the action *acquire*(*SMS*), i.e.,

$$\overset{SMS}{\dashrightarrow} e_recv$$

A flow can also exist between two actions. After an application gets an incoming SMS message, it can send the message to a remote server via the Internet. In such a case, it is a contextual flow between these two actions. The flow guarantees the two actions perform on the same SMS message. Therefore, we present the flow as:

$$e_recv \overset{\text{SMS}}{\dashrightarrow} e_send \overset{\text{SMS}}{\dashrightarrow}$$

Attack models

Based on the aforementioned building blocks for an attack, we define different attacks in this section. In the remainder of this section, we use the following notations. E is the set of Environmental Input triggers; t is the trigger of the attack and $t \in E$; *Asset* is the set of assets involved in the attack; Let α be an action or a trigger, β be an action, and γ be an asset. A flow is either a control flow denoted as $\alpha \rightarrow \beta$, or a data flow denoted as $\alpha \overset{\gamma}{\dashrightarrow} \beta$.

Attack taxonomy

We conduct a comprehensive investigation of existing attacks of malicious behaviors (Enck et al. 2009; Shabtai et al. 2010; Enck et al. 2011; Zhou and Jiang 2011; 2012), and propose a taxonomy of attacks in terms of these building blocks and semantic information as follows.

Privacy leakage Privacy leakage (Enck et al. 2010; Grace et al. 2012; Zhang and Yin 2014) refers to the exposure of sensitive information on devices. As discussed in the action part, such kind of information can be acquired by specifying an *acquire* action, which is regarded as *source* in the attack of privacy leakage. If there exists a data flow from the return value of the acquire action to the data sent out to the external environment by a communication action, usually called *sink*, privacy leakage happens. In addition, the attack needs to happen without users' awareness, and it is not necessary for the trigger to have a dataflow relationship with these two actions. As a result, the formal attack model of privacy leakage can be defined as:

$$PL \overset{t}{\Longrightarrow} e_recv \rightarrow acquireacquire \overset{\gamma}{\dashrightarrow} e_send(\gamma) \overset{\gamma}{\dashrightarrow}$$

Information interception Mobile devices can interact with the external environment in many ways. However, malicious applications intercept the communication, suspend, or even break off the communication. The common attacks include blocking an incoming SMS messages and phone calls. For such kind of attacks, malicious applications need to register a listener (i.e., *e_recv*) for broadcast messages of incoming messages and calls, which stops the spreading (i.e.,*intercept*) to avoid the messages from reaching to other applications or the user.

$$II =\overset{t}{\dashrightarrow} e_recv \rightarrow intercept(\gamma)$$

Content tampering Malicious applications may tamper content on mobile devices, such as contact, SMS, account, and system settings. It can cause severe damages to the user. Usually, an application can insert, update and delete an item in a content provider with specific permissions. In addition, it can change system settings such as network connection, wallpaper and sleep time. We use *insert*, *edit* and *delete* to describe such kind of behaviors. The trigger of this attack will not give rise to users' attention and does not have any data flow relationship with these actions. The attack is defined as follows:

$$CT \overset{t}{\Longrightarrow} e_recv \rightarrow \alpha(\gamma),\ where\ \alpha \in \{insert, edit, delete\}.$$

Service abuse Malicious applications may abuse the services provided by Android (Luo et al. 2013). According to our investigation, the most prevailing services which are abused include *phone service*, *SMS service*, *package manager*, and *download manager*. For example, if an application possesses the permission of sending SMS messages, it can subscribe a premium-rate mobile service which causes users' financial charge. Let α be the kind of actions which abuses services, and the attack model can be presented as:

$$SA \overset{t}{\Longrightarrow} e_recv \rightarrow \alpha(\gamma),\ where\ \alpha \in \{invoke, stop\}.$$

Resource depletion Due to portability and simplicity, mobile devices usually carry low-frequency CPU, RAM of limited size and small capacity battery. Mobile devices thereby can only provide a limited computation capability, storage and energy. It would make worse if any installed applications occupy these resources immoderately, which can influence other applications, and even the battery life of the device. Either intentionally or unintentionally, applications keep consuming resources (Pathak et al.; 2012; Vekris et al. 2012) or carry on useless and endless works (Oliner et al. 2012; Hao et al. 2013), while never release or stop them. Let *occupy* be the kind of actions which exhausts resources, and *release* be the kind

of actions which releases resources. And we use $\nrightarrow$ to show a missing flow between these two actions. The attack model is given in the following:

$$RD \stackrel{t}{=\rightarrow} e_recv \rightarrow occupy(\gamma) \nrightarrow release(\gamma)$$

Discussion

The taxonomy of attacks is based on the 102 malware families we have studied. However, there are some attacks out of detection of our approach, such as *fishing*, *adware* and *privilege escalation*. Fishing is a kind of attacks in which one application disguises an authentic and legitimate application, and induces users to enter their credentials of, for example, bank account (Prince). Adware is a program that displays advertisements to its users, which is annoying rather than harmful at most of time (F-Secure Lab 2013). Some applications may exploit the vulnerabilities of Android, such as Exploid, RATC/Zimperlich and Ginger Break (Xuxian and Yajin 2013), to elevate the privilege once installed on device; *Pilup* (Xing et al. 2014) is a newfound flaw in Package Management Service which can be exploited by malicious applications only during the phase of upgrading the Android OS. At last, *side channel* attacks (Schlegel et al. 2011; Hilgers et al. 2014; Chen et al. 2014), which collect memory information or timing information, are not our scope of attack detection.

The insufficiency of DROIDECHO comes from two aspects: 1) our static analysis is carried on Java code, and does not go inside the native code. Many of malware of privilege escalation utilize native code to elevate the privilege; 2) we try to avoid to make a subjective judgement, but prefer to detect an objective existence of malicious behaviors. That is, fishing and adware just deceive and bother users respectively, which do not violate security policies (Enck et al. 2009) of Android precisely. We give the statistics of attacks mentioned previously in Table 2, indicating that our approach can detect up to 90.4% of attacks in theory.

Disclaimers

There is a significant exception for determining an attack - disclaimers. A disclaimer is a white list for an application in which some behaviors are excluded from consideration for the determination of attacks. The violation of certain security properties cannot imply the occurrence of attacks. Some applications may need to carry on some suspicious looking behaviors which they already claimed the potential security violation explicitly. We conclude that the users who install their applications would like to undertake the introduced risks by default. Therefore, in this work, we filter out the "attacks" that are allowed by the users, and remove them from the generated attack report.

Table 2 The category of attacks on Android

Attack	Percent	Supported by DroidEcho
Privilege Leakage	31.4	✓
Information Interception	11.6	✓
Content Tampering	13.4	✓
Service Abuse	31.4	✓
Resource Depletion	1.8	✓
Fishing	1.7	✗
Adware	2.3	✗
Privilege Escalation	6.4	✗

The inter-component communication graph

For an accurate representation of Android applications and the convenience of attack detection, this section presents the proposed the inter-component communication graph (ICCG) to capture all possible communications between components and threads inside Android applications.

Android communication medium

Medium is a special data structure used for communications. The communications can occur between either two components (i.e., activity, service, broadcast receiver and content provider), or two isolated processes. Medium is playing a critical role in the behavior of Android applications. Besides the frequently-talked *Inter-Component Communication*(ICC) (Orthacker et al. 2011; Schlegel et al. 2011), which is based on the Intent medium, there are three other mediums which can be also used during the communication. Here we provide the different types of mediums existing on the Android platform.

Intent Intents are the main vehicle for communication. One intent can be either explicit or implicit. Explicit intents have a specific class to start, while implicit intents do not specify the corresponding class, and the system will select the most well-suited class or application to execute. An explicit intent can only invoke a specific component, which is defined in the constructor, or by calling `setComponent(ComponentName)` or `setClass(Context, Class)`; an implicit intent can be received by many well-suited components. It appoints potential receivers by setting an action in the constructor or `setAction(String)` (Meanwhile, it can be instrumented with a data type to restrict its receivers) (Feng et al. 2014). Intent can influence the execution order (a.k.a., control flow) of the application, and also impact on the data flow if enclosed with *extras*.

Message *Message* is a concise data structure for arbitrary data. Two isolated processes or threads can communicate with each other by transferring a message. In general, the message receiver has to create a *Messenger* to handle the

received messages. On the sender side, it needs to obtain the reference to this Messenger, and sends its crafted message by invoking `send(Message message)` of the *Messenger*. In order to send a message, for example, to a daemon service, the component can first bind to this service via `bindService()`, and then fetch the reference to the *Messenger* from the returned *Binder* object.

Binder Binder is used for a component to talk to a daemon service. The component which attempts to bind to a service needs to invoke `bindService()` and implement `ServiceConnection`, which establishes the connection with the service. On the service side, it needs to provide an inherited class of Binder, exposing public methods to customers; or design an AIDL interface as well as the implementation. After that, the component can obtain a binder object, which is a remotable object for a lightweight remote procedure call. In addition, AIDL can be exposed to other applications for remote invocations.

Persistent storage On Android, applications may exchange data through persistent storage. There are three types of persistent storage: *File, Shared Preferences and SQLite database*. They can be used for applications or components to exchange data, that is, they provide an implicit data flow from one component to another.

Inter-component communication graph

Definition 1. *Let M be the communication mediums existing on Android. An ICCG is a directed graph defined as* $G = \{V, E_f, E_c\}$*,where V is a set of nodes;* $E_f : V \times V$ *is a set of flow edges; and* $E_c : V \times M \times V$ *is a set of communication edges.*

The nodes of a graph are the methods contained in the application, which come with two levels of granularity. The coarse-grained nodes only represent the signature of the functions, and help to express the relationship between functions in the system level. We can learn the method invocation relationship and possible communications between different functions. In the fine-grained level of granularity, a node is in-depth dissected and shows the internal logic, i.e., control flow. When we are identifying the elements of attacks, especially behaviors, we need to go in deep at the code level, and recognize the different patterns of behaviors.

We employ two different kinds of edges to denote the relationship between nodes - *call* relationship and *communication* relationship. Flow edges reflect the call relationship among nodes. This is the primary concept in the program analysis, which consists of *explicit calls* and *implicit calls*. Here we emphasize the unique implicit calls, i.e., *Android Lifecycle*, existing on Android. An Android lifecycle indicates an implicit function invocation between different methods or classes. The implicit calls are either callbacks passed to a concrete method, or control frameworks specifying a call sequence. Besides the lifecycle features of standard Java, e.g., the method `void start()` of one thread instance will implicitly call the override method `void run()`, Android has included many libraries to support an amount of implicit calls. For each component of Android, it has a unique call sequence pre-defined by Android. In addition, all GUI components on Android allow developers to pass a callback to execute functionalities when the corresponding event occurs.

The communication edges are connecting between nodes and mediums. As defined previously, there are four kinds of mediums used for communication, and it is worth mentioning that the communications are not only showing the logic order of execution, some of them also enclose data which can be transferred from one node to another node.

We use the DroidKungFu malware[4] as an example to explain the ICCG. As shown in Fig. 1 (a), there are an activity and a service, which communicate via an Intent medium. The activity obtains sensitive data (refer to ① in `onStart`), and passes the data to the service. Then the service sends the data out at ② in `onCreate`. Figure 1

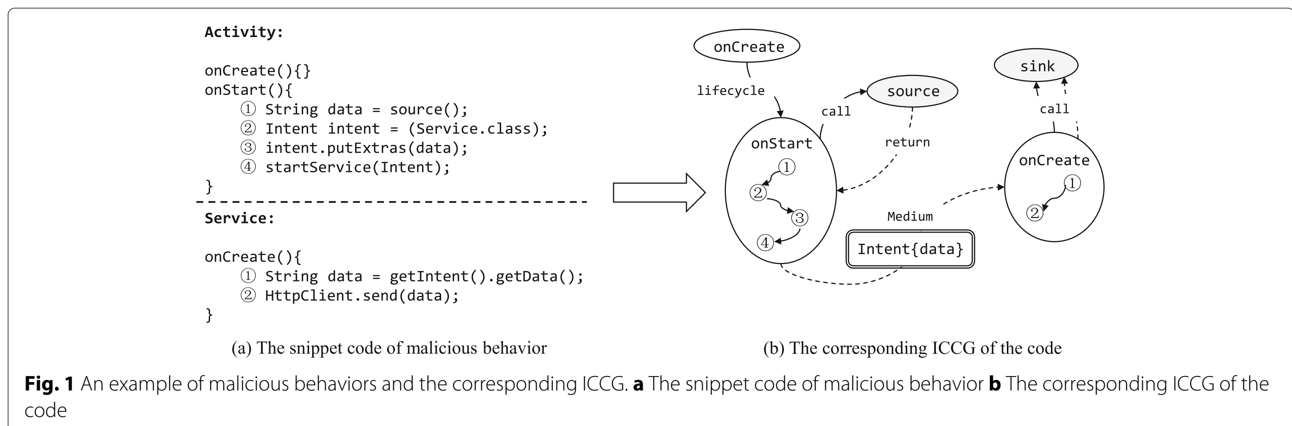

Fig. 1 An example of malicious behaviors and the corresponding ICCG. **a** The snippet code of malicious behavior **b** The corresponding ICCG of the code

(b) shows the constructed ICCG based on the code. As discussed in the previous section, each node represents a method of the application, and contains a control flow graph. The nodes are connected by two kinds of edges: Android mediums (e.g,. the Intent object) and method invocations either implicit invocations (e.g., lifecycle) or explicit invocations.

Sufficiency of ICCG

We construct ICCG for representing the overall structure of functions in the application, and search if any attack model is hidden in the graph. As the attack model proposed in Section Semantic model of attack is general and platform-independent, we show the sufficiency of ICCG to detect attacks below.

As modeled in Section Semantic model of attack, an attack is a set of operations which the attacker performs to achieve a certain objective, and it is composed of 5 essential elements. ICCG retains almost all program information, and we can extract a number of call sequences from it. By checking each call sequence, we can recognize actions which are attack related, identify the trigger of it, and perform data flow analysis on the call sequence. Hence, we could find a mapping from the attach model to the ICCG, which means that ICCG contains sufficient information to detect an attack inside.

System design of DroidEcho

This section presents the design of DroidEcho. As shown in Fig. 2, DroidEcho takes as input an Android application, which contains the class files, the manifest file and the description of its functionality. DroidEcho will generate an attack report which contains identified malicious behaviors and the corresponding traces of these behaviors for forensic use. DroidEcho leverages the *attack model* which is presented in Section Semantic model of attack as the guidance for attack detection, and proceeds in four phases: *disclaimer learning*, *ICCG construction*, *attack detection* and *attack confirmation*. The first phase *disclaimer learning* receives the descriptive text of applications as input, and generates a white list of "necessary" behaviors (a.k.a., disclaimer of the application) in a supervised manner. The white list will be used to exclude the detection for the claimed functionality of the application. Second, *ICCG construction* takes class files and the manifest file of the application as input, and constructs an ICCG, which is then passed to the third phase. *Attack detection* can find out, if any, existing attacks and the corresponding traces which cause these attacks in the application. At last, *attack confirmation* receives the candidate attacks, and determines whether one attack candidate is a false positive or not by a trace-guided dynamic execution.

Disclaimer learning

Some Android applications may perform seemingly suspicious behaviors while they are actually demanded to accomplish the functionality. The demanded functionality and the risks it may bring are usually claimed in their descriptive text. We regard this as a benign behavior (henceforth disclaimer), and it will not be considered as an attack candidate. For example, *TripAdvisor* is a travel application, which can provide the nearby restaurants and hotels when the user is travelling. For ease of use, it acquires the permission *FINE_LOCATION* to learn the user's location such that it can provide the most suitable information for the customers. Although we detect that TripAdisor has a privacy issue, which sends the user's location to a remote server from time to time, we regard this as being benign and harmless.

As shown in Fig. 3, we obtain the descriptions of applications and perform a description-to-permission fidelity analysis (Qu et al. 2014). The fidelity analysis builds a description-to-permission relatedness model in which one permission is associated with a list of noun phrases. For the description of a given application, we

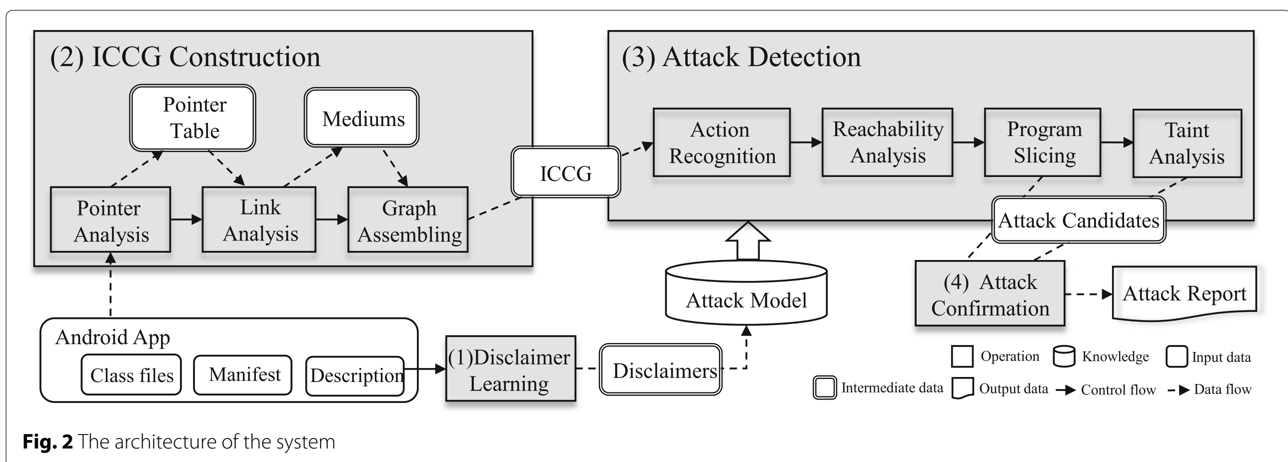

Fig. 2 The architecture of the system

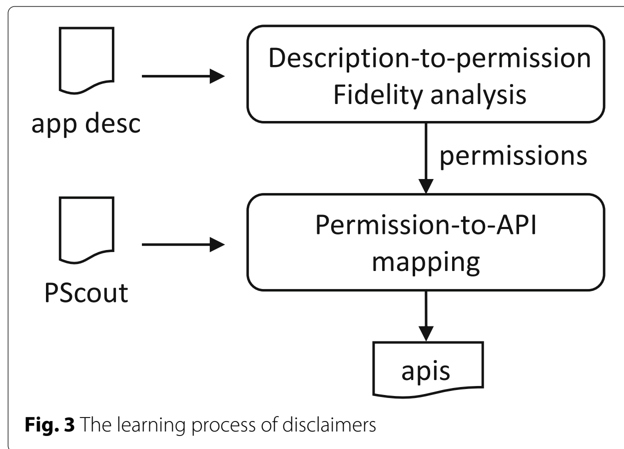

Fig. 3 The learning process of disclaimers

can leverage this model to produce a list of requested permissions. Then, we employ PScout (Au et al. 2012) to elicit the corresponding APIs that request permissions. For example, the sentence "Your location: These permissions are needed to obtain your location so we can help you discover hotels, restaurants, and attractions around you" in app TripAdvisor implies that it requests for recognizing users' current location the permission android.permission.ACCESS_COARSE_LOCATION and android.permission.ACCESS_FINE_LOCATION. Therefore, 21 Android APIs (e.g., void requestLocationUpdates(float, LocationListener) and Location getLastKnownLocation(String)) are regarded as being necessary to invoke by permission-to-api mapping.

The produced Android APIs serve as disclaimers to refine the attack model. During attack detection (see Section Attack detection), these APIs will not be considered as attack actions.

ICCG construction

The construction of ICCG takes class files and the manifest file of the application to be checked as inputs. Primarily, DROIDECHO employs *Soot* (Vallée-Rai et al. 1999) to generate a rough call graph of the whole application and a control flow graph for each method. Given that, DROIDECHO proceeds in three steps successively: *pointer analysis*, *link analysis* and *graph assembling*. The first two steps can provide all auxiliary information to assemble an ICCG.

Pointer analysis

Pointer analysis is a static analysis to infer which variables are pointed to by pointer references or heap references. In this step, we want to identify all references which are pointing to variables in the application, and all possible values which the variables can be assigned to. The result of this step is a *PointerTable*, which contains mappings from variables to concrete values: **Set**(*variables*) $\rightarrow$ **Set**(*values*). **Set**(*variables*) denotes a set of variables which are pointed to with the same reference at a time, and **Set**(*values*) denotes a set of possible values to which the variables can be assigned. *PointerTable* plays a critical role in the step of *link analysis* and *action recognition*. During the step link analysis, *PointerTable* is used to infer the actions and classes of an Intent object, thereby DROIDECHO can identify which components are able to receive this Intent. And DROIDECHO needs the *PointerTable* to recognize the semantics of actions during the action recognition. For example, when DROIDECHO encounters an operation to query a content provider, it needs to learn the value of the argument URI, to distinguish different content providers.

Parts of our pointer analysis are based on SPARK (Lhoták and Hendren 2003), which is a pointer analysis framework. It can cluster the variables into several sets, i.e., **Set**(*variables*), where all variables in the same set have been pointed to with same reference at a time. Since we have got a rough call graph and control flow graphs of all methods, we traverse the call graph and go inside control flow graphs to perform value inference. We evaluate each node in a control flow graph, and infer the possible values of the variables. The value inference can handle basic arithmetic and *String* operations. In addition, we do not evaluate all types of variables, which are both computation expensive and useless to our attack detection. We only pay attention to the valuation of primary types (e.g., *boolean*, *int*, *double*), *String*, *ComponentName*, URI/URL and Intent. It is worth mentioning that the values of *ComponentName* and URI/URL objects can be expressed by a *String*, while we construct a more complicated structure for Intent objects, which basically contains four fields: action, class, data and category.

The pointer analysis used in this work is type-sensitive, however, flow-insensitive. That is, every variable in the same set needs to share the same data type with others. In order to reduce the expense of storage and computation, we store all possible values which the set of variables can be assigned to rather than only parts of them after a certain statement.

Link analysis

Link analysis is to establish all links between methods or components in an application, i.e., the edges in ICCG. Primarily, the call graph generated by Soot only contains the call relationship between Java methods. As introduced in Section The inter-component communication graph, there are implicit invocations and a variety of communication mechanisms on Android. On the basis of the call graph, we analyze all links between methods and build a complete communication graph for the application.

There are two kinds of links between two methods, invocation links (either explicit or implicit) and communication links via Android medium (e.g., Intent and message). We first build call chains for the lifecycle

of Android components. For example, one of the call chains of Android Activity is `onCreate` → `onStart` → `onResume`, which shows the implicit invocations after the start of the Activity. As a result, the above methods in the call graph will be linked with an invocation edge, respectively. For communication links, we recognize the mediums as well as their attributes existing in the methods, and identify which components or methods can receive these mediums. Take the Intent medium as an example, if we find an action which starts activities, like `startActivity(Intent)`, we retrieve the attributes (e.g., class and action) of the Intent object and identify which activities can be triggered by this Intent object. As a result, we add a new link between the method which sends out the Intent and the constructor method of the target activities.

Graph assembling

By far, we have obtained the control flow graph for each method of the application, and all links between these methods. We take the control flow graphs as nodes, the links as edges, and assemble them into an ICCG. The graph depicts the execution order and communications between different methods at the system level, and illustrates the control flow at the method level. Combined with *PointerTable*, ICCG is passed to the attack detection phase. Attack detection will search the graph and find out any existing attack.

Attack detection

To reduce the search space of attack detection, we will not analyze the program from its entry points. In converse, we first recognize attack-related actions existing in the program in a fast way, and perform a bidirectional flow analysis from behaviors, which can effectively speedup the search process.

Algorithm 1 shows the whole process to check whether one attack is contained by the application or not. The algorithm takes ICCG of an application, and one attack model as the input, and outputs whether the attack model exists in the ICCG. Line 1-3 show that it recognizes all actions existing in the ICCG. If any of actions in the attack is not contained in the ICCG, DROIDECHO concludes that the application does not contain this attack. In our implementation, we conduct an one-time retrieval of the ICCG for each application and store all recognized actions. By comparing the included actions in each attack, we can quickly eliminate some attacks which will definitely not happen.

If all actions in the attack model are found in ICCG, we proceed the reachability analysis and program slicing. Since there are two kinds of flows (referred to control flow and data flow in program analysis, respectively) defined in our attack model, we carry on *ForwardControlFlowAnalysis* (Line 10) and *TaintAnalysis* (Line 6) to determine

Algorithm 1: Model-based attack detection

Input: *ICCG* of the application
Input: *Attack* model $\{(action_i, action_{i+1}, \text{data}|\text{control})\}$, where $0 \le i < n-1$
Output: if *ICCG* contains attack

```
1  for action ∈ attack do
2      if !(ICCG contains action) then
3          return false;
4  for i = 0 to #actions − 2 do
5      if flow(action_i, action_{i+1}) == data then
6          data_flow = TaintAnalysis(action_i, action_{i+1}, asset);
7          if data_flow is not satisfied then
8              return false;
9      if flow(action_i, action_{i+1}) == control then
10         control_flow = ForwardControlFlowAnalysis(action_i,
           action_{i+1});
11         if control_flow is not satisfied then
12             return false;
13 trigger := BackwardControlFlowAnalysis(action_0);
14 if trigger ∈ {Environmental Input} then
15     return true;
16 else
17     return false;
```

whether the flows are satisfied or not. At last, we get the trigger causing this attack (Line 13), and check if it is a kind of *environmental input*, e.g., the initialization of application, system broadcast message and a timer task. In the following, we will give a more detailed description for each step.

Action recognition

We use actions to describe the basic elements in an attack, which is semantic but domain-independent. However, we need to define a system of notations in a specific domain (here Android), to capture these actions and triggers in ICCG. On Android, we recognize an action by the corresponding constraints. Here we define three kinds of predicates to express APIs and constraints in these actions we met in the code: **sig**(*api*), **type**(*arg*), and **value**(*arg*), where *api* is an Android API, *arg* is a variable, and these predicates will return a comparable constant value. As a consequence, action recognition can be transformed into a satisfiability problem,

$$action \models \mathbf{sig}(api) \tag{1}$$

$$\mathbf{sig}(api) \models \mathbf{type}(arg) \cap \mathbf{value}(arg) \tag{2}$$

One action is recognized if we detect some APIs which satisfies the above constraints progressively. Equation 1 shows the action can be recognized with an API with the specific signature, and moreover, the arguments or the base, if any, need to satisfy two kinds of predicates, **type** and **value**. As shown in Eq. 2, *arg* is either the base of the API (static methods do not have a base), or the arguments. Specially, *arg* may be another invocation of API, i.e., **sig**. Therefore, we will recursively solve the constraints until

the action is recognized. Taking the example of obtaining contacts, the essential code at language level of this action can be described as follows:

$$\frac{sig(api) = obj.query(uri, *)}{obtain\ contact} \tag{3}$$

$$\frac{\begin{array}{l} type(obj) = ContentResolver, \\ type(uri) = Uri, \\ value(uri) = \text{“}content : //contacts\text{"} \end{array}}{sig(api) = obj.query(uri, *)} \tag{4}$$

As shown in Eq. 3, we first need to find a pivotal function whose signature matches `obj.query(uri, *)`, and the methods need to meet three constraints: the base of the invocation *obj* needs to be an object of the class `android.content.ContentResolver`, the type of *uri* needs to be an object of `android.net.Uri`, and its value needs to be `content://contacts` as shown in Eq. 4. The code statements, which together form a behavior, might have dependency relationship or follow an execution order in between. We deal with it as a constraint satisfaction problem, and recognize a behavior with reasoning. The benefits are that we do not need to care about the execution order of code in a behavior, and hence our approach is more general so as to identify more variations.

Reachability analysis & slicing

If the ICCG contains all necessary elements for one attack, we start to do program slicing from these elements. The slicing consists of backward and forward control flow analysis. The backward control flow analysis aims to complete three tasks: 1) find the root cause that lead to such action, i.e., its entry points. Based on the entry points, we can infer the type of the triggers. Then we know whether the attack is triggered by a user interaction or environmental inputs; 2) obtain all conditions in a trace from the entry points to the action. The conditions are used in *attack confirmation* to guide the dynamic execution of the application; 3) identify the search space for potential taint analysis.

The forward control flow analysis aims to complete two tasks: 1) determine the occurrence of the subsequent actions in an attack model; 2) similar to the backward control flow analysis, identify the search space for the taint analysis. As a result, we will not search the entire ICCG during the taint analysis, which is computationally expensive.

Taint analysis

Taint analysis can track the flow of data during detection. Taking privacy leakage as an example, we need to carry on taint analysis to track the flow of data, and if the data is flowed to a sink action and sent out eventually. During the taint analysis, we get a domain set in a control-flow order $SearchDomain = D_1 \rightarrow D_2, ... \rightarrow Dn$, and the source action is located at D_{sr} after the above steps. Then we perform a forward data flow analysis on the domain set *SearchDomain*. Figure 4 illustrates the ways how the data can be tainted cross domains. First of all, data in the domain D_s can influence the data in its previous domain by three methods: return the data at the call site in the previous domains, referring to ①; the data flow ② shows how the data in the latter domain influences the data in its previous domains; and we can assign the data to one commonly shared variable between the domain D_s as shown in ③. There are three possible ways for the data in domain D_s to influence the data in the successive domains: enclose communication medium with data and pass it to the next domains as shown by the data flow ④; pass the data as an argument to its successive domains, which are used in these domains, referring to ⑤; assign the data to a commonly shared variable in between as shown by the data flow ⑥. In addition, we take a coarse-grained aliasing analysis in this paper, i.e., if for example a string variable

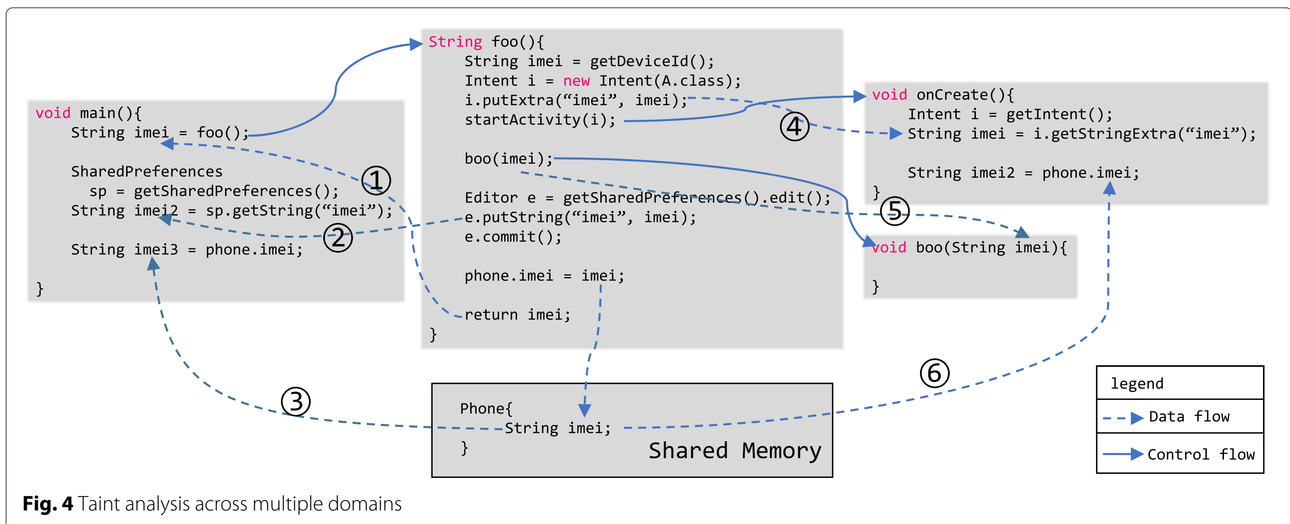

Fig. 4 Taint analysis across multiple domains

is passed to a function, and this function will encrypt the string and return a new encrypted value with a cryptographic scheme. Although we do not know how to convert the original string to the encrypted one (we do not infer the meaning of cryptographic schemes), we can definitely ensure the operation is reversible, and the returned data is also of sensitive information.

Dynamic attack confirmation

As discussed before, DROIDECHO's ICCG construction and attack detection are based on static program analysis, which is less precise than dynamic analysis. As a result, the attacks reported by DROIDECHO may be false positives. Therefore, we introduce a confirmation step to reduce false positives, and the attack confirmation is based on the technique of dynamic testing.

An attack candidate, which is passed from the *attack detection* phase to the *attack confirmation* phase, contains an attack trace and the conditions that guarantee the occurrence of attacks. Given that, we simulate the inputs to drive the dynamic execution of the application and check whether the attack trace can occur in the real execution. In order to activate the attack candidate and capture malicious behaviors, we first instrument Android OS by hooking specific Android APIs which are included in our attack model, and then generate the triggers which are used to activate the contained malicious behaviors.

- **Instrumentation.** Since the actions in attack model are recognized as the invocations of specific Android APIs, we instrument Android OS to monitor the invocation behaviors. In this paper, we leverage TaintDroid (Enck et al. 2010) to determine whether these APIs are invoked.
- **Triggers.** We leverage IntelliDroid (Wong and Lie 2016) to generate all triggers leading to specific malicious behaviors, and subsequently schedule these triggers to drive the execution of the application. We simply feeds the application with all possible trigger sequences, and in order to eliminate the impossible sequences (which never occur during the real executions), we exploit the "*happen-before*" relations among these triggers to generate sequences.

Obtaining these inputs, DROIDECHO is able to execute the suspicious applications to determine if the attack is reachable. In order to make the exploration faster, DROIDECHO prunes the paths which rarely lead to the attack trace, which can significantly reduce the search space of the program.

Evaluation

We implement an automatic platform DROIDECHO to facilitate the detection, accordingly. DROIDECHO is written with 17,038 lines of Java, and 163 lines of scripts (Python and Shell). The dynamic confirmation is implemented based on TaintDroid (Enck et al. 2010) and IntelliDroid (Wong and Lie 2016). TaintDroid enables us to track the information flows of applications. In addition, we customized TaintDroid for two purposes. First, we intercept the APIs in our *Action* set to monitor whether they are invoked by the tested applications. Second, we intercept the APIs providing the applications with environmental inputs, such as location and time information, where we can return the applications values that would activate the target behaviors. During the confirmation, we employ IntelliDroid to generate the call paths for specific Android APIs as well as conditions that enable the paths. Then the driver script takes them as input to automatically drive the execution of the suspicious applications. To estimate the overall performance of DROIDECHO, we conduct the experiments from three aspects: *evaluation on malware benchmark*, *evaluation on real apps* and *evaluation on performance*.

Evaluation on Malware Benchmark

To evaluate the performance of our approach on the infamous malware, we conduct an experiment on 1260 samples of malware of the collection (Zhou and Jiang 2011). According to the types of malware, we filter out 108 of them (e.g., Asroot, DroidCoupon and DroidDeluxe) which only use native code to launch attacks. At last, we successfully detect 940 (89.5%) samples, and also show the attack type. There are mainly two reasons for the missing malware: 1) some malware use reflection to dynamically invoke malicious code. For example, AnserverBot loads an executable file in its asset folder, retrieves the included classes and runs the code. 2) some of them leverage complicated obfuscation and encryption to confuse AV tools. For example, Geinimi leverages several cryptographic schemes (e.g., DES) to encrypt the communication and strings.

In addition, we conduct an experiment to compare DROIDECHO's capability of attack detection with FlowDroid (Arzt et al. 2014b), which is a static tool in detecting privacy leakage. The subjects of this experiments include a set of open-source Android applications named DroidBench[5], of which the applications may contain the attacks of privacy leakage.

DROIDECHO successfully detects 34 samples of malware, while fails to find 8 malicious samples. We provide Table 3 to illustrate the comparison results, actually only the different results, with FlowDroid. As shown in Table 3, DROIDECHO has an edge in detecting the first six kinds of privacy leakage, but cannot detect the last three kind of privacy leakage. PrivateDataLeak-1&2 are two applications which steal the text in a password field of an Android GUI view. Since the data on GUI components

Table 3 Comparison with FlowDroid

App	DroidEcho	FlowDroid
ArrayAccess-1&2	TP	FP
HashMapAccess1	TP	FP
ListAccess1	TP	FP
Ordering1	TP	FP
Unregister1	TP	FP
Exception-1&4	TP	FP
PrivateDataLeak-1&2	FN	TP
ImplicitFlow-1&2&3&4	FN	FN
Reflection-3&4	FN	FN

are hard to be determined to be sensitive, in addition, applications which need authentication have to send credentials, such as user input from keyboard, to the remote server for authentication. As a result, DroidEcho does not track the flow of the data on GUI components. And last, DroidEcho and FlowDroid both cannot cope with the last two kinds of applications, where ImplicitFlows are samples which leverage obfuscation techniques to confuse the analysis, and Reflections are two samples which use reflection to dynamically invoke methods or fetch fields to complete the process of privacy leakage.

Evaluation on real Apps

We have collected 7643 applications from Google Play, which are hot and free application in their respective categories. By running DroidEcho, we find out 444 applications which have malicious behaviors. In addition, we have done a statistics of behaviors which are user-awared or already claimed by the description of applications. We compare DroidEcho with other anti-virus (AV) tools, by uploading *apk* files into VirusTotal (www.virustotal.com). Although AV tools have detected 1541 (20.2%) samples of malware, most of them are Adware, of which the number is up to 1217 (79.0%). Due to the restriction of our approach, we do not provide a detection for Adware. By filtering these applications of Adware, we can also find 149 more applications which have malicious behaviors.

We investigate the 149 applications which contain malicious behaviors, of which 131 applications have privacy leakages, while the remaining applications have other four kinds of malicious behaviors. In particular, 10 applications contain service abuse attacks, i.e., sending SMS messages without users' consent; 6 applications contain content tampering attacks, i.e., deleting SMS messages from the inbox; 2 applications are depleting battery by holding Screen lock for a long time. By investigating the code of these applications, we find that many of them are employing a third-party library which has exposed sensitive information. The third-party libraries may do a measurement for the usage of applications, e.g., Flurry and Crittercism, diagnose the crash of applications, e.g., Crashlytics, or advertise, e.g, Umeng and Google Ads. Table 4 shows third-party libraries that are contained in the applications.

False positive analysis To evaluate DroidEcho's accuracy, we randomly selected 50 samples, and manually identified 4 false positives. Two false positives are because DroidEcho cannot well handle collection objects such as array, list, and map. If any element in a collection is tainted, DroidEcho determines the whole collection object is tainted. One false positive is due to the ignorance of execution conditions of flows. The execution condition may not be satisfied during runtime leading the malicious behaviors cannot be practically triggered. The last false positive is attributed to the insufficient modelling of persistent storage. As an alternative communication channel, persistent storage (e.g., file, database) might contains multiple dimensional data. It is non-trivial to track the flow of data in the persistent storage, which will be further studies in future.

Evaluation on performance

In order to evaluate the efficiency and scalability of DroidEcho, we measured runtime parameters in the previous experiments. The runtime parameters consist of the complexity of applications and runtime for each phase of DroidEcho. And the experiments are conducted on a Linux Ubuntu 14.04 machine, carrying 12 cores of Intel Xeon(R) CPU E5-16500, and 16G Memory. We depict the complexity of applications from four aspects: the file size of application, the number of nodes, edges and mediums of the ICCG. We have measured the runtime for pointer analysis, link analysis, action recognition and attack detection, respectively. The detailed data can be found in Table 5. As shown in Column *Runtime(ms) of DroidEcho*, DroidEcho is very effective in detecting

Table 4 Privacy leakage via 3rd-libraries

Library	Description	Num	Behaviors
Adobe	Measurement of Usage	1	Identity Code, etc.
Flurry	Measurement of Usage	20	Identity Code, Location, etc.
Conversant	Measurement of Usage	1	Identity Code, Location, etc.
Crashlytics	Diagnosis of Crash	8	Identity Code, Sys. Info, etc.
Map Service	Map Service	5	Location, etc.
Crittercism	Optimization Tool	1	Identify Code, etc.
Umeng	Advertisement	4	Identity Code, Location, etc.
Google Ads	Advertisement	3	Identity Code, Location, etc.
Amazon Ads	Advertisement	1	Identity Code, Locatoin, etc.
Millennialmedia	Advertisement	2	Identity Code, Location, etc.

Table 5 Evaluation on performance of DroidEcho

	Size (K)	ICCG			Runtime(ms) of DroidEcho						Runtime(ms) of Soot
		#N	#E	#M	Pointer	Link	Assembling	Recognition	Detection	Total	
DroidBench	186	15	1	0	46	3	14	11	40	114	24,702
Malware	893	1,327	6,070	5	4,818	108	55	747	2,358	8,086	65,241
Real Apps	5,392	3,900	75,117	10	17,114	611	453	3,742	13,301	35,221	135,763

attacks, with the average time of about 35s to complete the analysis of a real application. In addition, since we leverage Soot to generate the rough call graph and control flow graphs for each method of applications, the runtime of Soot should also be considered to complete the whole detection. Soot performs a heavy work of reverse engineering, i.e., converting Android *.dex* code into Java bytecode, the time spent on that is hence much larger than the runtime of DROIDECHO.

Discussion

Our attack detection is guided with the semantic attack models, which describe the essential attack elements combined in a logic order. In this way, our approach is general such that we could detect several types of attacks as well as their variations on Android. Although it is hard to include exhaustive attack types, considering that *zeroday* attacks occur from time to time, each augmentation of the attack model can enhance and increase the ability of detecting attacks significantly. On the other hand, we have improved the conventional static analysis on Java with taking into account the new features provided by the Android platform. It helps to produce a more complete and comprehensive communication graph for Android applications, and thereby makes the attack detection more accurate and effective. However, considering the flaws of static analysis and the experiment results we got, DROIDECHO still has some shortcomings in detecting attacks:

Transformation attacks

It is a kind of attacks against anti-malware tools and approaches, with transforming a malware into different forms, but reserving the original logic (Rastogi et al. 2013). Our approach has a sufficient resistance against *trivial transformation attacks* and *transformation attacks detectable by static analysis (DSA)*. However, *transformation attacks non-detectable by static analysis (NSA)*, e.g., *reflection* and *bytecode encryption*, can paralyze our approach, which is also a common issue in static analysis.

Vulnerability exploits

We put more attentions on the attacks which invoke Android APIs. There exist a kind of attacks which exploit the vulnerabilities of Android, and trigger the vulnerabilities by crafting a special input or executing some code in a certain order. It is more difficult when the exploits are written in native code. To date, our work only accepts Java bytecode as the analysis object.

The limitations of our approach can be largely ascribed to the expressive ability of the attack model. Since the detection is based on static analysis, the attack model proposed in this paper only contains static features of attacks. As a result, we can detect more attacks by enriching and enhancing the attack model, for example, taking into account dynamic features of attacks.

Related work

Attack representation

Chen et al. (Chen et al. 2013) present permission event graphs (PEG) to depict API- and permission-related behaviors occurring on Android. In addition, to express the sequence of occurrence of events, they add the temporal order and leverage the LTL to depict a policy specification. Combining static analysis, model checking and runtime monitoring, they are able to detect the violation of contextual policies of Android applications. Gunadi and Tiu (Gunadi and Tiu 2013) propose a security policy specification language to describe privilege escalation on Android. The language is based on metric linear-time temporal logic (MTL) plus an extension of recursive definitions. It can help to figure out the context-sensitive privilege for one application. By monitoring the chain of privilege in runtime, they manage to find out the elevated privilege and detect collusion attacks. Aiming at privacy issues on Android, Arzt et al. (Arzt et al. 2014b) reduce them into an IFDS (Reps et al. 1995) problem, and construct a flow- and context-sensitive graph to present the entire behavioral system by static analysis. Graph reachability and value evaluation are performed to figure out whether the messages being sent out are tainted as sensitive information. Yang et al. (Yang et al. 2014) propose a two-level behavioral graph (Component Dependency Graph and Component Behavior Graph) to express the program logic. At first, they leverage an unsupervised mining approach to mine the program logic in malware automatically. Based on the mined graphs, they search crawled applications from marketplaces whether they contain any of malicious behaviors or not. Mariconti et al. (Mariconti et al. 2016) propose to use Markov Chain to represent malicious behaviors in Android malware, and employ static analysis to identify malicious behaviors. AppContext (Yang et al. 2015) proposes two heuristics

(i.e., activating and guarding conditions) to identify malware, while not classifying malware in terms of attack targets.

A handful of works are devoted to identifying user-intended behaviors. In a PEG, Chen et al. (Chen et al. 2013) define pre-conditions either with or without users' consents. Although it only focuses on GUI operations, it provides a new prospective of learning the essential characteristics of malware. AppIntent, proposed by Yang et al. (Yang et al. 2013), is another work to extract a sequence of GUI operations which causes data transmission. They first reduce the search space by static analysis to avoid time-consuming, but useless, searching; then the event sequence is generated after running symbolic execution guided by the reduced space.

Our attack model combines the program-level behaviors and external inputs (i.e., triggers) to model attacks. First, on the program level, we consider the combination of assets, actions and flows to model a complete attack behavior. In addition, our model is not on an abstract level as most of the previous studies do. It thus can be directly mapped to the real implementation of the tested applications, without missing critical details of the attacks. Second, the triggers are taken into our consideration, which can effectively differentiate the benign behaviors from malicious ones.

Attack detection

Attack detection via *program analysis* can be roughly divided into two categories: dynamic analysis and static analysis. TaintDroid (Enck et al. 2010) tracks the propagation of sensitive information on a customized Android, and determines whether there exists any attack of privacy leakage. DroidScope (Yan and Yin 2012) and VetDroid (Zhang et al. 2013) both reconstruct malicious behaviors by collecting information during the dynamic analysis. However, the difficulty of the deployment of the monitor system restricts the scale of attack detection; and exhaustive test inputs are nearly impossible, which means attacks may not be triggered and detected sometimes due to insufficient inputs.

As a result, more researchers focus on detecting attacks via static analysis. FlowDroid (Arzt et al. 2014b) performs static analysis, specifically dataflow analysis, on the code of applications to check if they contain behaviors of privacy leakage. IccTa (Li et al. 2015) incorporates ICC analysis to achieve a more complete and accurate detection of privacy leakage. However, these two approaches are only focusing on the attack detection of privacy leakage. DroidSIFT (Zhang et al. 2014) analyzes the code of applications and constructs behavior graphs to denote the program logic. Taking the behavior graphs as signatures, DroidSIFT builds a classification system to distinguish benign applications from malware. Apposcopy (Feng et al. 2014) takes into account the inter-component communication on Android, and constructs an inter-component call graph to link up all components of the application to detect malware of privacy leakage with crafted signatures. Different from these two approaches, our approach proposes to detect attacks based on semantic model of attacks and use dynamic analysis to confirm their maliciousness.

Our approach combines two approaches, static and dynamic analysis, and achieves both advantages of two aspects. We first employ static analysis based on semantic models of attacks to quickly find out the potential malicious applications, with the trigger and the predicates which cause the occurrence of attacks. Then we leverage dynamic analysis to confirm the attacks to reduce false positives. As a result, our approach is effective on large-scale tests and reduces the false positive rate via dynamic attack confirmation.

Conclusion

In this paper, we introduce a novel attack model to depict the essential characteristics and features. In addition, we build a transformation from an Android application to a directed graph, called the inter-component communication graph. ICCG captures all structure information of application, including call relationships and communication between different methods, and it contains all control flow information for each method. Then we propose an effective algorithm to search attacks in ICCG. The approach is proved to be feasible and effective in the experiments. In future, we expect to extend our detect algorithm to handle more complicated obfuscation or encryption techniques, and will continue enriching the attack model in order to handle more variants or new attacks.

Acknowledgments

Kai Chen was supported in part by National Key R&D Program of China (No. 2016QY04W0805), NSFC U1536106, 61728209, National Top-notch Youth Talents Program of China, Youth Innovation Promotion Association CAS, Beijing Nova Program and a research grant from Ant Financial. This work is also partly supported by International Cooperation Program on CyberSecurity, administered by SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China (No. SNSBBH-2017111036).

Authors' contributions

All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Author details

[1] SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. [2] Nanyang Technological University, Singapore, Singapore. [3] Singapore Institute of Technology, Singapore, Singapore. [4] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China.

Received: 4 January 2018 Accepted: 17 April 2018
Published online: 05 June 2018

References

Arzt S, Bodden E (2016) StubDroid: Automatic Inference of Precise Data-flow Summaries for the Android Framework. In: Proceedings of the 38th International Conference on Software Engineering. pp 725–735

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P (2014) FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh. pp 259–269

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, McDaniel P (2014) Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14. ACM, New York. pp 259–269

Au KWY, Zhou Y, Huang Z, Lie D (2012) PScout: Analyzing the Android Permission Specification. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12. ACM, New York. pp 217–228

Bosu A, Liu F, Yao DD, Wang G (2017) Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, Abu Dhabi. pp 71–85

Chen KZ, Johnson NM, D'Silva V, Dai S, MacNamara K, Magrino TR, Wu EX, Rinard M, Song DX (2013) Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In: 20th Annual Network and Distributed System Security Symposium, NDSS '13, San Diego. http://internetsociety.org/doc/contextual-policy-enforcement-android-applications-permission-event-graphs

Chen QA, Qian Z, Mao ZM (2014) Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In: Proceedings of the 23rd USENIX Conference on Security Symposium, SEC'14. USENIX Association, Berkeley. pp 1037–1052

Enck W, Gilbert P, Chun B-G, Cox LP, Jung J, McDaniel P, Sheth AN (2010) TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10. USENIX Association, Berkeley. pp 393–407

Enck W, Octeau D, McDaniel P, Chaudhuri S (2011) A Study of Android Application Security. In: Proceedings of the 20th USENIX Conference on Security, SEC'11. USENIX Association, Berkeley. pp 21–21

Enck W, Ongtang M, McDaniel PD (2009) Understanding Android Security. IEEE Secur Priv 7(1):50–57

F-Secure Lab (2013) Mobile Threat Report, January - March 2013. Technical report

Feng Y, Anand S, Dillig I, Aiken A (2014) Apposcopy: Semantics-Based Detection of Android Malware Through Static Analysis. ACM, New Year. https://doi.org/10.1145/2635868.2635869

Grace MC, Zhou Y, Wang Z, Jiang X (2012) Systematic Detection of Capability Leaks in Stock Android Smartphones. In: 19th Annual Network & Distributed System Security Symposium. http://dblp.uni-trier.de/rec/bib/conf/ndss/GraceZWJ12

Gunadi H, Tiu A (2013) Efficient runtime monitoring with metric temporal logic: A case study in the android operating system. CoRR abs/1311.2362. http://arxiv.org/abs/1311.2362

Hao S, Li D, Halfond WGJ, Govindan R (2013) Estimating Mobile Application Energy Consumption Using Program Analysis. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13. IEEE Press, Piscataway. pp 92–101

Hilgers C, Macht H, Müller T, Spreitzenbarth M (2014) Post-Mortem Memory Analysis of Cold-Booted Android Devices. In: Proceedings of the 2014 Eighth International Conference on IT Security Incident Management & IT Forensics, IMF '14. IEEE Computer Society, Washington. pp 62–75

Lhoták O, Hendren L (2003) Scaling Java Points-to Analysis Using SPARK. In: Proceedings of the 12th International Conference on Compiler Construction, CC'03. Springer-Verlag, Berlin. pp 153–169

Li L, Bartel A, Bissyandé TF, Klein J, Traon YL, Arzt S, Rasthofer S, Bodden E, Octeau D, McDaniel PD (2015) IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1. pp 280–291

Luo W, Xu S, Jiang X (2013) Real-time Detection and Prevention of Android SMS Permission Abuses. In: Proceedings of the First International Workshop on Security in Embedded Systems and Smartphones, SESP '13. ACM, New York

Mariconti E, Onwuzurike L, Andriotis P, Cristofaro ED, Ross GJ, Stringhini G (2016) Mamadroid: Detecting android malware by building markov chains of behavioral models. CoRR abs/1612:04433

Octeau D, McDaniel P, Jha S, Bartel A, Bodden E, Klein J, Traon YL (2013) Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis. In: Proceedings of the 22Nd USENIX Conference on Security, SEC'13. USENIX Association, Berkeley. pp 543–558

Oliner AJ, Iyer A, Lagerspetz E, Tarkoma S (2012) Collaborative Energy Debugging for Mobile Devices. In: the 8th Workshop on Hot Topics in System Dependability. USENIX, Berkeley

Orthacker C, Teufl P, Kraxberger S, Lackner G, Gissing M, Marsalek A, Leibetseder J, Prevenhueber O (2011) Android Security Permissions - Can We Trust Them? In: Security and Privacy in Mobile Information and Communication Systems. Springer Berlin Heidelberg, Berlin. pp 40–51

Pathak A, Hu YC, Zhang M Bootstrapping Energy Debugging on Smartphones: A First Look at Energy Bugs in Mobile Devices. In: Proceedings of the 10th ACM Workshop on Hot Topics in Networks, HotNets-X. ACM, New York. pp 5:1–5:6. https://doi.org/10.1145/2070562.2070567

Pathak A, Hu YC, Zhang M (2012) Where is the energy spent inside my app? Fine-grained Energy Accounting on Smartphones with Eprof. In: Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12. ACM, New York. pp 29–42. https://doi.org/10.1145/2168836.2168841

Prince B New Android Malware Targets Banking Apps, Phone Information: Fireeye. http://www.securityweek.com/new-android-malware-targets-banking-apps-phone-information-fireeye. Accessed 05 Oct 2017

ProGuard (2017). http://developer.android.com/tools/help/proguard.html. Accessed 03 Dec 2017

Qu Z, Rastogi V, Zhang X, Chen Y, Zhu T, Chen Z (2014) AutoCog: Measuring the Description-to-permission Fidelity in Android Applications. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp 1354–1365

Rastogi V, Chen Y, Jiang X (2013) DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, ASIA CCS '13. ACM, New York. pp 329–334

Reps TW, Horwitz S, Sagiv S (1995) Precise Interprocedural Dataflow Analysis via Graph Reachability. In: Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco. https://doi.org/10.1145/199448.199462

Schlegel R, Zhang K, Zhou X, Intwala M, Kapadia A, Wang X (2011) Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In: 18th Annual Network and Distributed System Security Symposium

Shabtai A, Fledel Y, Kanonov U, Elovici Y, Dolev S, Glezer C (2010) Google Android: A Comprehensive Security Assessment. IEEE Secur Priv 8(2):35–44

Symantec Inc. (2017) Internet Security Threat Report. Technical report

Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V (1999) Soot - a Java Bytecode Optimization Framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '99. IBM Press. p 13. http://dl.acm.org/citation.cfm?id=781995.782008

Vekris P, Jhala R, Lerner S, Agarwal Y (2012) Towards Verifying Android Apps for the Absence of No-Sleep Energy Bugs. In: Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems, HotPower'12. USENIX Association, Berkeley. pp 3–3

Wei F, Roy S, Ou X, Robby (2014) Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp 1329–1341

Wong MY, Lie D (2016) IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In: 23rd Annual Network & Distributed System Security Symposium

Xing L, Pan X, Wang R, Yuan K, Wang X (2014) Upgrading Your Android, Elevating My Malware: Privilege Escalation Through Mobile OS Updating. In: IEEE Security & Privacy

Xu K, Li Y, Deng RH (2016) ICCDetector: ICC-Based Malware Detection on Android. IEEE Trans Inf Forensics Secur 11(6):1252–1264

Xuxian J, Yajin Z (2013) Android Malware. SpringerBriefs in Computer Science
Yan LK, Yin H (2012) DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In: USENIX Security. USENIX Association, Berkeley. pp 29–29
Yang C, Xu Z, Gu G, Yegneswaran V, Porras PA (2014) DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications. In: 19th European Symposium on Research in Computer Security. Springer International Publishing. pp 163–182
Yang W, Xiao X, Andow B, Li S, Xie T, Enck W (2015) AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. Proceedings of the 37th International Conference on Software Engineering. pp. 303–313
Yang Z, Yang M, Zhang Y, Gu G, Ning P, Wang XS (2013) AppIntent: Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In: Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security, CCS '13. ACM, New York. pp 1043–1054
Zhang M, Duan Y, Yin H, Zhao Z (2014) Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In: Proceedings of the 21th ACM Conference on Computer and Communications Security, CCS '14, Scottsdale
Zhang M, Yin H (2014) Efficient, Context-aware Privacy Leakage Confinement for Android Applications Without Firmware Modding. In: Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS'14), Kyoto
Zhang Y, Yang M, Xu B, Yang Z, Gu G, Ning P, Wang XS, Zang B (2013) Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS '13. ACM, New York. pp 611–622. https://doi.org/10.1145/2508859.2516689
Zhou Y, Jiang X (2011) An Analysis of the AnserverBot Trojan. Technical report. http://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBot_Analysis.pdf
Zhou Y, Jiang X (2012) Dissecting Android Malware: Characterization and Evolution. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP '12. IEEE Computer Society, Washington. pp 95–109

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com