Computers & Security 126 (2023) 103090

Contents lists available at ScienceDirect

Computers & Security

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

A permission-carrying security policy and static enforcement for information flows in Android programs

Xiaojian Liu*, Kehong Liu

College of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an, Shaanxi, 710054, China

A R T I C L E I N F O

Article history:
Received 28 January 2022
Revised 6 October 2022
Accepted 30 December 2022
Available online 1 January 2023

Keywords:
Information flows
Security policy
Taint analysis
Android programs
Malware

A B S T R A C T

To detect information leaks in Android programs, existing taint analysis approaches usually specify and enforce (statically or dynamically) the *two-level* information flow policy, represented by a lattice $(\{\top, \bot\}, \sqsubseteq)$ with $\bot \sqsubseteq \top$. However, this policy leaves permissions (an access control mechanism built in the Android system) out of consideration, causing a too *coarse-grained* analysis result in some scenarios. In fact, the existing information flow controls should integrate permissions to develop a more refined flow policy. Following this intuition, in this paper, we propose a permission-carrying secure information flow policy and accomplish a static enforcement mechanism for this policy. We first devise a small language to capture typical features of Android programs. On this base, we define the permission-carrying security policy using a subset lattice of permissions, and offer a group of rules to certify the security of information flows. Secondly, we implement a static enforcement mechanism for this policy, which allows us to detect potential insecure information flows by analysing programs in component-level. To illustrate the usefulness of this policy, we further examine two typical security threats, *confused deputy* and *collusive data leaks*, as the running cases to show how to detect security attacks in our theoretical framework. The final experiment shows that our approach is effective and scalable for real-world apps. Compare with several leading tools, our approach is applicable to checking both intra-app and inter-app vulnerabilities, and achieves a more precise detection result due to the superiority of our fine-grained security policy.

1. Introduction

With an explosive spread of mobile applications (Android or iOS) in our daily lives, a large amount of privacy-sensitive information deposits in smartphones; it is very imperative to protect privacy from illegal leaks and intentional abuses. Taint analysis is a promising approach to detect and track potential information dissemination, especially for Web and mobile applications (Arzt et al., 2014; Backes et al., 2017; Enck et al., 2014; Gordon et al., 2015; Pan et al., 2018; Sun et al., 2016; Tripp et al., 2013; Wei et al., 2018; Xue et al., 2018). Privacy-sensitive objects (data or some security-sensitive APIs), called *sources*, are labelled as *taints*; risky ports (usually in form of APIs) publicly accessed by external observers are called *sinks*. If an information flow

$$source \overset{\alpha_1;\cdots;\alpha_n}{\rightsquigarrow} sink$$

from a source to a sink via executing commands α_i is detected (statically or dynamically), then a potential information leak will be revealed.

Performing a valid taint analysis requires specifying and enforcing a certain *information flow policy* for a given program. An information flow policy, in essence, is a lattice $(SC, \sqsubseteq)$ (Denning, 1982), where SC is a finite set of *security classes*, and $\sqsubseteq$ is a binary relation partially ordered the classes of SC. Information is only permitted to flow within a security class or upward, but not downward or to unrelated classes. The existing taint analysis approaches are essentially based on the *two-level* information flow policy, namely, SC is limited to $\{\bot, \top\}$, and $\sqsubseteq$ simply orders $\bot \sqsubseteq \top$. The source object has the $\top$ (*High*) security class, and the sink object has the $\bot$ (*Low*) security class. A propagation $source \overset{\alpha_1;\cdots;\alpha_n}{\rightsquigarrow} sink$ means an information flow from *High* level to *Low* level — apparently not permitted by the policy.

1.1. Challenges

However, the two-level flow policy might be too *coarse-grained* in some scenarios. Consider the following two typical security threats: *confused deputy* (also called *privilege escalation at application-level*) (Bugiel et al., 2012) and *inter-app collusion* attacks (note: their threat models and formal definitions will be given in Section 2.2 and 5 respectively), illustrated in Fig. 1.

* Corresponding author.
E-mail addresses: 780209965@qq.com (X. Liu), 1292692089@qq.com (K. Liu).

https://doi.org/10.1016/j.cose.2022.103090
0167-4048/© 2022 Elsevier Ltd. All rights reserved.

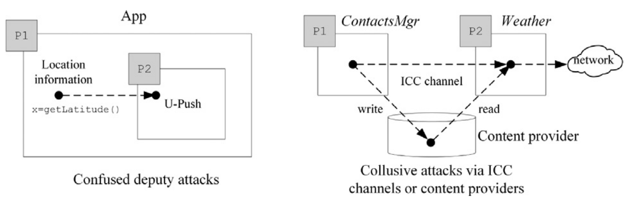


Fig. 1. Confused deputy and inter-app collusion attacks.

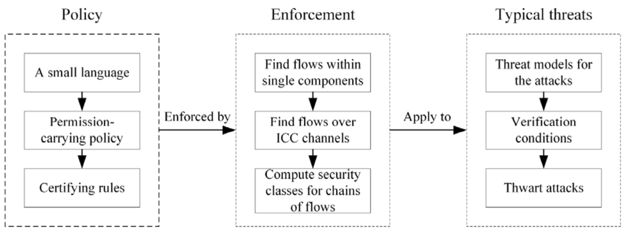

Policy
A small language
Permission-carrying policy
Certifying rules
Enforced by
Enforcement
Find flows within single components
Find flows over ICC channels
Compute security classes for chains of flows
Apply to
Typical threats
Threat models for the attacks
Verification conditions
Thwart attacks

Fig. 2. The workflow of this paper.

Confused deputy attacks

Assume an app integrates *U-Push*[1], a third-party SDK (Software Development Kit) of Push category, to implement a precise message pushing service. U-Push uses a set of permissions P2 to collect private information such as network states, phone states, and locations. But from version 6.2.0, U-Push SDK ceases to be equipped with `ACCESS_FINE_LOCATION` permission, no longer allowed to collect the user's location information. However, the app may need location information for other purposes and include this permission in its used permissions P1. In this case, U-Push is *overprivileged*—still able to share this permission and collect location information, posing a threat of *confused deputy*.

To prevent private information flows into U-Push, we could use taint analysis to detect potential information flows and issue an alarm once finding a flow to U-Push. However, this practice is imprecise and prone to incur high risks of both *false positive* and *false negative*. As all private information (such as network states, phone states, and locations) is labelled with *High* class, the taint analyser cannot further distinguish the "*sources*" of the tainted information; this may result in a dilemma—if all such tainted information flows to U-Push are alarmed, then the result may be too *coarse*, leading to a high false positive, because some kinds of information, such as network states and phone states, are still allowed to flow into U-Push; otherwise, if all information flows to U-Push are *not* alarmed, the result may suffer from a high false negative, because location information is not allowed to flow into U-Push from version 6.2.0.

Inter-app collusion attacks

Two apps *ContactsMgr* and *Weather* in a smartphone may collude with one another to disclose privacy (Marforio et al., 2012). The app *ContactsMgr* with `READ_CONTACTS` as one of its used permissions (P1) has access to the users contacts, but has no access to the network; the app *Weather* with `INTERNET` as one of its used permissions (P2) can access the network, but has no access to the users contacts. Each colluding malware only requests a minimal set of privileges, which may make it appear benign under conventional detection. However, the both may interact with each other by exploiting storage channels (e.g., log files or Content Providers) or ICC (Inter-Component Communication) channels to perform malicious attacks, such as leaking contacts information to a specific Web server. We call this scenario the attacks of *inter-app collusions*.

Detecting collusion attacks with taint analysis approach also encounters the same problem with confused deputy, due to coarse-grained security classes cannot distinguish information with different *sources*. For example, *ContactsMgr* may get some privacy (e.g., contacts or location) and write to a log file, then the log becomes tainted; afterwards, an information flow from the log to *Weather* is detected. If this flow is alarmed, then a false positive may be yielded, because *Weather* may be privileged to get some private information, such as locations; otherwise, if this flow is not alarmed, then apparently a false negative will be incurred.

As a more precise solution to these problems, we improve the traditional taint analysis with a more refined information flow policy. We refine the two-level security classes $(\perp, \top)$ with a hierarchy of permissions, and label each security-related data with a subset of permissions, which allows us to distinguish different "*sources*" of information and produce a more precise detection result.

For example, for the threat of confused deputy, if the app get some location information by calling `x=getLatitude()`, then `x` gets tainted and labelled with a set of permissions.
{`ACCESS_FINE_LOCATION`, `ACCESS_COARSE_LOCATION`}.

Once a flow from `x` to U-Push is detected and this set is not a subset of P2, meaning that U-Push has no privilege to access location information, then a threat of confused deputy is to be alarmed. Otherwise, if the set is a subset of P2, then the flow is permitted without any alarm. Obviously, this solution achieves a more precise detection result than the traditional taint analysis.

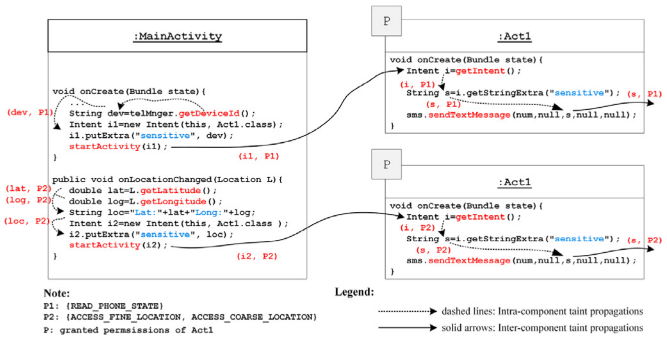
1.2. Main work and contributions

The main work of this paper is outlined in Fig. 2. We first devise a small language to capture typical features of Android programs. On this foundation, we define the permission-carrying information flow policy, and offer a group of rules to certify the security of information flows. The second part of our work is dedicated to a static enforcement of this policy, which allows us to verify the security (or detect the vulnerability) of information flows by simply analysing program texts without any executions. Furthermore, to investigate the usefulness of this policy, we use *confused deputy* and *collusive data leaks* as two typical threats throughout this paper to demonstrate our solution to detecting security attacks under our framework.

Several formal languages (Calzavara et al., 2017; Chaudhuri, 2009) have been devised to capture typical features and operational semantics for Android programs. Although close to the language in Chaudhuri (2009), ours additionally captures the heap abstraction for objects (including both component instances and local user-defined class objects), as well as retains a more sophisticated permission-feature to exactly model the real permission mechanism in Android system. HornDroid (Calzavara et al., 2017; 2016) exhibits a formal language μ-Dalvik$_A$ for Android programs, but it only covers limited typical Android-specific features. More importantly, the permission mechanism, being indispensable for investigating security of Android programs, however, was not involved yet in the language. Besides, to regulate secure information flows, some typical works, such as Jif (Myers, 1999; Myers and Liskov, 2000) and SME (Chakraborty et al., 2019), proposed multi-level lattice-based security policies. Jif provided a more general model for the control of information flows, but its application is confined in pure Java programs—some Android-specific structure and security features have not yet been addressed. The policy lattice of Chakraborty et al. (2019) derives from the *protection levels* of Androids permissions, however, this lattice, in essence, is still a binary coarse-grained policy, though more levels could be accommodated for different application scenarios.

In summary, the main contributions of this work are as follows:

[1] https://www.umeng.com/push.



```
:MainActivity
void onCreate(Bundle state){
    ...
    String dev=telMnger.getDeviceId();
    Intent i1=new Intent(this, Act1.class);
    i1.putExtra("sensitive", dev);
    startActivity(i1);
}
public void onLocationChanged(Location L){
    double lat=L.getLatitude();
    double log=L.getLongitude();
    String loc="Lat:"+lat+"Long:"+log;
    Intent i2=new Intent(this, Act1.class );
    i2.putExtra("sensitive", loc);
    startActivity(i2);
}
```

```
:Act1
void onCreate(Bundle state){
    Intent i=getIntent();
    String s=i.getStringExtra("sensitive");
    sms.sendTextMessage(num,null,s,null,null);
}
```

```
:Act1
void onCreate(Bundle state){
    Intent i=getIntent();
    String s=i.getStringExtra("sensitive");
    sms.sendTextMessage(num,null,s,null,null);
}
```

Note:
P1: {READ_PHONE_STATE}
P2: {ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION}
P: granted permsissions of Act1

Legend:
dashed lines: Intra-component taint propagations
solid arrows: Inter-component taint propagations

Fig. 3. The running example.

1. *A language-based framework*. We design a small language and present its operational semantics for Android programs. On this foundation, we are able to formally define information flows, present security rules, specify threat models, and develop enforcement algorithms.
2. *A permission-based multilevel flow policy*. We improve the traditional two-level security policy in taint analysis with a hierarchical security policy that labels each privacy-sensitive object with a set of permissions and propagates them along program executions. As truly signalling access rights to security-related resources (such as device IDs, phone states, and locations), permissions can clearly act as "*labels*" distinguishing and tracking different "*sources*" of information, whereby leading to a fine-tuning result than the traditional taint analysis.
3. *A static enforcement of the policy*. We *over-approximate the multiple objects of a component by their class*, generate ICCGs (Inter-Component Communication Graphs), and enforce the policy for a given Android program in component-level instead of object-level. This treatment not only ensures the *safety* of the enforcement, but also leads to a simplified solution to the problem.

The remainder of this paper is organized as follows. Section 2 gives a motivating example and offers our solution. Section 3 introduces a small language for Android programs and presents its operational semantics. Section 4 specifies the permission-carrying flow policy and provides a set of rules for secure information flows, and Section 5 gives formal models for confused deputy and collusion threats. Section 6 devotes to the enforcement of this policy. We implement our approach and report the experimental results in Section 7, and finally present discussion and related work in Sections 8 and 9 respectively.

2. Motivation and solution

2.1. Motivating example

In this section, we use a running example (Fig. 3) to show the basic idea of our approach. The example demonstrates such a scenario of information leaks. The main Activity calls the APIs `getDeviceId()`, `getLatitude()`, and `getLongitude()`, to get some security-sensitive information about the user's device and locations, and then encapsulates these information to Intent objects. Afterwards, the main Activity launches two instances of `Act1` (assume the launch mode of `Act1` is `standard`) by calling `startActivity()`. Each instance receives Intents by calling `getIntent()`, unmarshalls the information from the Intents, and then sends the private information to the outside via the SMS manager.

We call the APIs that obtain security-sensitive information the *source*-APIs, such as `getDeviceId()`, `getLatitude()`, and `getLongitude()`; the APIs that disseminate information to the outside world the *sink*-APIs, such as `sendTextMessage()` in this example. A source-API is generally guarded by one or more permissions, which serve as a prerequisite for a successful calling. In this example, permission P1, i.e., `READ_PHONE_STATE` guards `getDeviceId()`; permissions P2, i.e., $\{\texttt{ACCESS\_FINE\_LOCATIOIN}, \texttt{ACCESS\_COARSE\_LOCATIOIN}\}$ guard `getLatitude()` and `getLongitude()`.

In most cases, rather than confined within a single component, an information leak may cross over multiple components through ICC (Inter-Component Communication) channels. We call the APIs that send Intents the *exit*-APIs, such as `startActivity()`; the APIs that receive Intents the *entry*-APIs, such as `getIntent()` in this example. Connecting an *exit*-API and an *entry*-API, an ICC channel is established, and via this channel Intents are transferred from the source component to the target.

2.2. Threats and our solution

To illustrate our solution, we examine two typical security attacks, *confused deputy* and *collusion*, as the target to fight against.

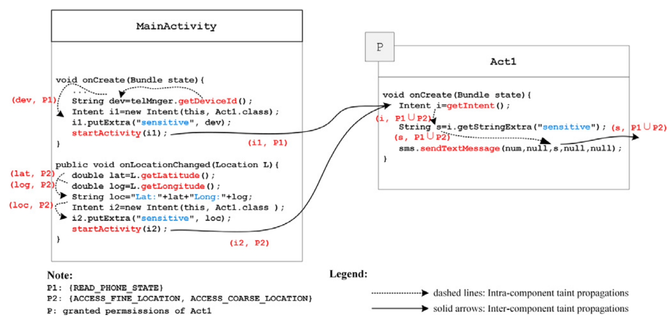
Threat model. In general, *confused deputy* attacks concern scenarios where a malicious application exploits the vulnerable interface of another privileged (but confused) application. On the other hand, *collusion* attacks concern malicious applications that *collude* to combine their permissions, allowing them to perform actions beyond their individual privileges (Bugiel et al., 2012).

Note that, in some studies, confused deputy attacks are referred to as *privilege escalation at application-level* as well, because a most common trick of an attacker is to escalate its own privileges by exploiting another privileged application. Privilege escalation attacks can be launched at two levels: *kernel and middleware-level* and *application-level*. The former exploits kernel-controlled channels and completely bypasses the middleware reference monitor; some of these vulnerabilities have been discovered and reported so far (such as CVE-2022-20114, CVE-2021-0307, CVE-2020-0418, CVE-2021-0306, CVE-2021-0317 (Li et al., 2021)), but not concerned in this paper. Our main concentration in this work is on the latter, i.e., confused deputy—an adversarial application (or third party SDK) exploits another privileged application (or component). To emphasis our main concern, we would prefer the term *confused deputy* (rather than *privilege escalation*) to designate the threat to fight against.

Solution. To track information flows and thwart potential security attacks, we shall label each variable that carries security-related information as *tainted*. Rather than labelled simply with $\top$ and $\bot$, we attach a set of permissions to each tainted variable. For example, after calling the source-API `getDeviceId()`, the variable "`dev`" gets tainted and labelled with permission P1, denoted by the pair "`(dev, P1)`" in Fig. 3, meaning that any valid access to "`dev`" should be guarded by the permission P1.

Once the information flow "`(dev, P1)` $\rightsquigarrow$ `(i1, P1)`" within `MainActivity` enters an instance of `Act1` via an ICC channel, a detection against confused deputy should be conducted: if $\texttt{P1} \subseteq \texttt{P}$, meaning that <u>`:Act1`</u> has a privilege to access device ID, then there is no risk of confused deputy in the information flow; otherwise, an attack of confused deputy will be reported.

Furthermore, we can enforce this policy to detect collusive data leaks among multiple components. For example, the single flows "`getDeviceId()` $\rightsquigarrow$ `(dev, P1)` $\rightsquigarrow$ `(i1, P1)` $\rightsquigarrow$ `startActivity()`" and "`getIntent()` $\rightsquigarrow$ $(i, \emptyset)$ $\rightsquigarrow$ $(s, \emptyset)$ $\rightsquigarrow$ `sendTextMessage()`" seem no harm within each corresponding component, but whenever they are linked together through a channel "`startActivity()`



```
MainActivity

void onCreate(Bundle state){
    ...
    String dev=telMnger.getDeviceId();
    Intent i1=new Intent(this, Act1.class);
    i1.putExtra("sensitive", dev);
    startActivity(i1);
}

public void onLocationChanged(Location L){
    double lat=L.getLatitude();
    double log=L.getLongitude();
    String loc="Lat:"+lat+"Long:"+log;
    Intent i2=new Intent(this, Act1.class );
    i2.putExtra("sensitive", loc);
    startActivity(i2);
}
```

```
Act1

void onCreate(Bundle state){
    Intent i=getIntent();
    String s=i.getStringExtra("sensitive");
    sms.sendTextMessage(num,null,s,null,null);
}
```

Note:
P1: {READ_PHONE_STATE}
P2: {ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION}
P: granted permsissions of Act1

P
(dev, P1)
(i1, P1)
(lat, P2)
(log, P2)
(loc, P2)
(i2, P2)
(i, P1∪P2)
(s, P1∪P2)
(s, P1∪P2)
Legend:
dashed lines: Intra-component taint propagations
solid arrows: Inter-component taint propagations

Fig. 4. Enforce the policy in component-level.

$\xrightarrow{ICC}$ getIntent()", an information dissemination from getDeviceId() to sendTextMessage() is to be revealed.

2.3. Enforce the policy statically in component-level

In the above running example, there are two exit-points in MainActivity—one is startActivity(i1) in the callback onCreate(); another is startActivity(i2) in the callback onLocationChanged(). However, the entry-point, i.e., the statement Intent i=getIntent() in onCreate(), is unique, but distributed over two instances of Act1. To alleviate this situation and provide a simplified enforcement, we borrow the idea from the static analysis—*over-approximates multiple objects by their class*—this allows us to cope with the issue in component-level rather than object-level, as illustrated in Fig. 4.

To this end, we need to construct a graph, referred to as *ICCGs* (*Inter-Component Communication Graphs*), to reflect ICC channels among multiple components. An ICC channel is a connection between an exit-point and a corresponding entry-point. If multiple channels converge to the same entry-point, then the permissions transferred through these channels should be *merged*, reflecting such a requirement that any one of these channels *may* contribute to the entry-point. In the running example, two channels meet at the same entry-point Intent i=getIntent(), causing the permissions P1 and P2 to be merged at this point, i.e., (i, P1∪P2). Consequently, to detect against confused deputy, we should check the condition $P1 \cup P2 \subseteq P$, which considers the contributions of the both information flows into account.

3. Language

To precisely specify information flows in Android programs and formally define threat models, we shall first define executions of Android programs. Android programs are usually encoded by Java language and execute on the Dalvik (or ART for Android 5.0 or above) virtual machines. So in principle, the complete executions of a program could be achieved by integrating the executions of the Java program and those of the underlying virtual machine. However, to simplify the treatment and focus on the main features of Android programs, we need to present a high level description for Android programs.

In what follows, we first give a concept model for Android programs, then propose a small language to describe the crucial commands of programs. Afterwards, the operational semantics of the language is provided.

3.1. Applications, components, and permissions

The basic entities in Android programs include: *applications, components*, and *permissions*. Each entity has a group of attributes (or features) configured in the *manifest.xml* of an app. Here we only

Definitions

$$\begin{aligned}
fil &::= \mathsf{N} \rightarrow \mathsf{CMP} \\
a \in \mathsf{APP} &::= \mathsf{app}(decl, enf, cmps) \\
cmps &::= \{c_1, \ldots, c_n\} \\
c \in \mathsf{CMP} &::= \mathsf{activity}(C, exp, fil, enf, \lambda x.t_1, \lambda x.t_2) \\
&\quad\ \mid \mathsf{service}(C, exp, fil, enf, \lambda x.t) \\
&\quad\ \mid \mathsf{receiver}(C, exp, fil, enf, \lambda x.t) \\
&\quad\ \mid \mathsf{provider}(C, exp, u, r, w, v) \\
ic \in \mathsf{iCMP} &::= \text{component instance of } c \\
p \in \mathsf{P}_{sys} &::= (level, gids, rsc, rights) \\
p \in \mathsf{P}_{uri} &::= (uris, rights)
\end{aligned}$$

Helper notations

$$\begin{aligned}
c.\mathsf{app} \in \mathsf{APP} &: \text{the app where component } c \text{ is defined.} \\
c.\mathsf{exp} \in \{\mathsf{T}, \mathsf{F}\} &: \text{the } exp \text{ attribute of } c. \\
(a|c|ic).\mathsf{decl} \subseteq \mathsf{P}_{sys} &: \text{the declared permissions of } a,\ c, \text{ or } ic. \\
(a|c|ic).\mathsf{enf} \subseteq \mathsf{P}_{sys} &: \text{the } enf \text{ attribute of } a,\ c, \text{ or } ic. \\
c.\mathsf{r} \subseteq \mathsf{P}_{uri} &: \text{the } r \text{ attribute of Provider } c. \\
c.\mathsf{w} \subseteq \mathsf{P}_{uri} &: \text{the } w \text{ attribute of Provider } c. \\
ic.\mathsf{type} \in \mathsf{CMP} &: \text{the component type of } ic \\
n.\mathsf{sp} \subseteq \mathsf{P}_{sys} &: \text{the sending permissions of } n \\
n.\mathsf{rp} \subseteq \mathsf{P}_{sys} &: \text{the receiving permissions of } n \\
sameApp(ic, ic') &: ic.\mathsf{type}.\mathsf{app} = ic'.\mathsf{type}.\mathsf{app}
\end{aligned}$$

Fig. 5. The main features of Android programs.

focus on the features of interest, and abstract away the others. The definitions of these entities are given in Fig. 5.

We first introduce some domain notations. We use APP, CMP, iCMP, $\mathsf{P} = \mathsf{P}_{sys} \cup \mathsf{P}_{uri}$, and INTENT to denote the finite sets of *applications, components, component instances, permissions*, and *intents respectively*. A permission is either a system built-in permission (P_{sys}) or a URI-permission (P_{uri}). Notations N, C, and URI refer to the finite sets of *names* (i.e., strings with Java-style naming convention), Java classes, and URIs. A name $n \in \mathsf{N}$ can refer to an app, a component, a URI, a permission, or an Intent action. For example, the name com.tencent.mobileqq refers to the QQ app, android.permission.READ_SMS refers to the permission of reading SMS messages.

An Intent filter fil is defined as a mapping from a name $n \in \mathsf{N}$ to a component. For an implicit Intent, n is the *action* name of the Intent. Each Android application $a \in \mathsf{APP}$ consists of a set of *declared* permissions $decl \subseteq \mathsf{P}_{sys}$, a set of *enforced* permissions $enf \subseteq \mathsf{P}_{sys}$, and a group of components. The declared permissions of an app are of the permissions declared in the *manifest.xml* of the app and granted when it was installed in the smartphone; the enforced permissions are of the permissions enforced by the app to other apps that intend to interact with it.

Components $c \in \mathsf{CMP}$ in an app fall into four kinds: *Activity, Service, BroadcastReceiver* (*Receiver* for short), and *ContentProvider* (*Provider* for short). The first attribute $C \in \mathsf{C}$ is the Java class that implements c. Each kind of component except for *Provider* is also equipped with declared permissions, however they are derived from the declared permissions of its app; in other words, the declared permissions of c, denoted $c.\mathsf{decl}$, equals to $c.\mathsf{app}.\mathsf{decl}$.

The enforced permissions can be declared in both applications and (or) components. If the enforced permissions of a component are explicitly declared, then they will override the counterpart of the component's app; otherwise, they will be simply inherited from the component's app, i.e., $c.\mathsf{enf} = c.\mathsf{app}.\mathsf{enf}$ in this case.

The attribute *exp* of a component c indicates whether or not c is to be *exported* to other apps. If $exp = \mathsf{T}$, other apps are able to interact with c; otherwise, if $exp = \mathsf{F}$, the access to c is confined to the same app of c, namely, only the components of c's app (or the app having the same UserID) are able to interact with c.

For a Provider component, $u \in \mathsf{URI}$ is a URI identifying the authority of Provider component; $r, w \subseteq \mathsf{P}$ are of the "read" and "write" permissions for particular URIs in the Provider component; v is the values stored in the component.

To faithfully model the control flow transfers between components, we need also specify the callbacks defined in components. λ-abstractions $\lambda x.t_1$ and $\lambda x.t_2$ in Activity components correspond to the callbacks `onCreate()` and `onActivityResult()` respectively. They model the behaviours of Activity components once receiving the launching Intents and the returned values respectively. $\lambda x.t$ in Service components simulates the behaviour of callbacks `onCreate()`, and $\lambda x.t$ in Receiver components mimics the behaviour of callbacks `onReceive()` when receiving broadcast events. *Provider* components play a role of data storages, being *inactive* and no callbacks.

We distinguish permissions into *system built-in* permissions and *URI*-permissions. A system built-in permission $p \in \mathsf{P}_{sys}$ is a 4-tuple, where $level \in \{\mathsf{normal, dangerous, signature, signatureOrSystem}\}$ is the protection level of the permission; different levels stand for various protection hierarchies, indicating the potential risks of the permission. *gids* is the set of *supplementary group IDs* associated with the permission. Note that, the permission mechanism in Android system is based on the access control of Linux OS (Smalley and Craig, 2013); each permission essentially corresponds to one or more group IDs. For example, the permission `INTERNET` is mapped to a single group "inet", whereas `WRITE_EXTERNAL_STORAGE` is mapped to the groups "sdcard_r" and "sdcard_rw". *rsc* is the resource to be protected by the permission. For example, the resource of the permission `READ_CONTACTS` or `WRITE_CONTACTS`, is the "contacts" in the mobile phone, which records a large amount of personal contact information for the phone user. *rights* is the set of operations permitted by the permission on the *rsc*. For example, `WRITE_CONTACTS` allows for "write" operations, and `READ_CONTACTS` allows for "read" operations on the "contacts".

A URI-permission $p \in \mathsf{P}_{uri}$ is a pair, where $uris \subseteq \mathsf{URI}$ is a set of URIs defined in the Provider component, *rights* is the "read" or "write" access right to the *uris*.

The **Helper notations** facilitate the subsequent descriptions. We use *dot*-notations to denote attributes of an entity. For example, $a.\mathsf{decl}$ denotes the declared permissions of an app a. Note that, the declared and enforced permissions of a component instance derive from its type, i.e., $ic.\mathsf{decl} = ic.\mathsf{type}.\mathsf{decl}$ and $ic.\mathsf{enf} = ic.\mathsf{type}.\mathsf{enf}$. For an Intent whose action part is a string $n \in \mathsf{N}$, we use $n.\mathsf{sp}$ and $n.\mathsf{rp}$ to denote the permissions associated with sending and receiving the Intent.

3.2. Syntax

Not attempting to cover all program features, the proposed language only focuses on the features essentially relevant to *security of information flows*. These features mainly involve permissions, launching component instances, ICCs, and the APIs that obtain and propagate security-sensitive information. The commands given in our language faithfully reflect such features, and present a high-level description of programs.

The syntax of the language is illustrated in Fig. 6. Let $n \in \mathsf{N}$ be a string, $x \in \mathsf{VAR}$ be a variable defined in Android programs, void for the emptiness, $p \in 2^{\mathsf{P}}$ denotes a set of permissions, $u \in \mathsf{URI}$ a uri, e an arithematic expression, and be a boolean expression.

$$
\begin{aligned}
v &::= n \mid x \mid \mathsf{void} \\
i \in \mathsf{INTENT} &::= (n, v) \\
t \in \mathsf{CMD} &::= basic \mid \mathsf{start}_m(i) \mid \mathsf{bind}(i, \lambda x.t) \mid \mathsf{return}(v) \\
&\quad \mid \mathsf{send}(i, p) \mid u?x \mid u!v \\
&\quad \mid x := \{p\}\mathsf{source}(v) \mid \{p\}\mathsf{sink}(v) \\
basic &::= \mathsf{skip} \mid x := e \mid t_1; t_2 \mid t_1 \triangleleft be \triangleright t_2 \mid be * t \\
&\quad \mid \mathsf{try}\ \{t_1\}\ \mathsf{catch}(exception)\ \{t_2\} \\
&\quad \mid \mathsf{new}\ C \mid proc(x_{in}, x_{out})
\end{aligned}
$$

Fig. 6. The syntax of the language.

The non-terminals in the syntax are explained as follows.

1. Each Intent i is a pair (n, v). For an *explicit* Intent, n is the name of the target component class in programs; whereas for an *implicit* one, n is the *action* part of i, to be matched with the target component by the Android system. v is the value carried with i. The form (n, void) represents the Intents without carrying any values.
2. Commands t include *basic* commands and a group of Android-specific commands.
3. Command $\mathsf{start}_m(i)$ starts an Activity instance with Intent i. This command mimics the Android API `startActivity()` or `startActivityForResult()`. The subscript m means the launching modes of Activity, which can be $\mathsf{standard}$, $\mathsf{singleTop}$, $\mathsf{singleTask}$, or $\mathsf{singleInstance}$ (Google, 2020).
4. Command $\mathsf{return}(v)$ simulates the behaviour of the Android APIs `setResult(v);finish()`—the sub-Activity first sets the returned value to its parent-Activity, then finishes itself. Apparently, $\mathsf{start}_m(i)$ followed by $\mathsf{return}(v)$ encodes such a data-fetching process that a parent-Activity starts a sub-Activity by calling `startActivityForResults()`, then the sub-Activity returns some results to the parent-Activity, which will subsequently trigger the callback `onActivityResult()` on the parent-Activity side.
5. Command $\mathsf{bind}(i, \lambda x.t)$ encodes the Android API $\mathsf{bindService}(i)$; $\lambda x.t$ corresponds to the callback `onServiceConnected()` defined in the caller component. Remember that when a Service component has been connected successfully, the Android framework will return the instance (or the handler) of the Service component to the caller by calling `onServiceConnected()`.
6. Command $\mathsf{send}(i, p)$, modelling the Android API `sendBroadcast()`, sends Intent i to a Receiver instance while enforcing permissions $p \subseteq \mathsf{P}_{sys}$ to this instance.
7. Commands $u?x$ and $u!v$ model the operations "read data to variable x from" and "write a value v to" a Provider component authorized by URI u respectively.
8. $x := \{p\}\mathsf{source}(v)$ is a general form of all source-APIs through which security-sensitive information v flows into variable x. These APIs are usually guarded or preconditioned by a set of permissions p.
9. Likewise, $\{p\}\mathsf{sink}(v)$ is a general form of all sink-APIs from which value v leaks to public observers.

3.3. Semantics

As information flows always propagate along with program execution steps, we shall first define executions, i.e., the operational semantics, for the commands of the language. Android programs are programmed using Java language, so it could be assumed to define the semantics following the approach for object-oriented programs, such as the rCOS (Jifeng et al., 2006) and the syntax-directed Hoare Logic (Pierik and Boer, 2003). However, the following differences between Android programs and general Java programs motivate us to revisit the semantics for Android programs.

First, executions of Android programs are initiated by external events instead of *main* methods; *Second*, the life-cycles of component instances are completely governed by the Android system; by no means can programmers create and destroy component instances. *Third*, Android programs involve richer program features, such as permissions and ICCs, which should be involved in the semantics.

The semantics of our language is defined as a state-transition system. Each command α transforms the system from a state s to the next state s', denoted by

$$\langle \alpha, s\rangle \rightarrow \langle \alpha', s'\rangle,$$

where s and s' are called the *pre* and *post* states of the command, meaning that if α starts properly in a state s, it will terminate in a state s' and start on the execution of α'. Executions of Android programs are specified by sequences of state transitions from initial states.

Observables

Identifying states relies on what program behaviours to be observed. In our setting, we denote a state s as a 4-tuple

$$(ic, h, \sigma, \mathcal{S}),$$

where each element is an observable:

1. $ic \in \mathsf{iCMP}$: the current active component instance.
2. h: the heap memory, which includes the existing component instances and the general objects that are created and used by these component instances.
3. σ: the local environment of objects. For a given object o (a component instance or a general object), its local environment, denoted $\sigma(o)$, is a function that maps the fields or local variables of o to values. The values can be primitive type values or object references.
4. $\mathcal{S}$: the stack of Activity instances. Note that, the *stacks* here refer to the data structures used to manage the navigation relationships among Activity instances, rather than the "stacks" of procedural calls in common programming languages. For example, in standard launching mode (Google, 2020), once an Activity instance A is launched by the current Activity instance B, then A is pushed to the current stack as the new top element; when A returns, it is popped, and B comes back to the top of the stack.

We use $s = (ic, o, \sigma, \mathcal{S})$ and $s' = (ic', o', \sigma', \mathcal{S}')$ to denote the pre- and post-states respectively for executions of commands, thus the semantics of a command is specified as a relation $\mathcal{R}(s, s')$ on s and s'.

Transition rules

As the semantics for the *basic* commands has been extensively studied in a large body of works (Calzavara et al., 2017; 2016; Jifeng et al., 2006; Nielson et al., 2004; Pierik and Boer, 2003), here we only concentrate on the semantics of the Android-specific commands. Moreover, as Chaudhuri (2009) shown, Service components can be encoded as Receiver components, and accordingly bind() commands can be thought of send() commands, so we only define the transition rules for the commands except for bind.

A transition rule can be represented as a form

$$\frac{Premise}{Conclusion},$$

where *Premise* is a predicate (i.e., a relation $\mathcal{R}(s, s')$ on s and s') that constitutes the condition for the state transition, *Conclusion* is the enabled state transition. The transition rules for the commands in our language are shown in Table 1.

Command $\mathsf{start}_{std}((n, v))$ starts an Activity instance by initiating an Intent (n, v). The subscript std means standard launching mode. Here we only consider the standard mode; the semantics for the other modes can be defined similarly, but their forms will appear much more complicated, as more elaborated operations on the stacks are required in these modes.

Starting an Activity instance with standard mode involves creating a *fresh* target instance (i.e., ic'), and pushing it to the current stack; the component of the target instance must match the component designated by the filter fil, i.e., $ic'.\mathsf{type} = fil(n)$. If the target instance and the current instance (i.e., ic) do not reside in the same app, then the target component must be *exposed* (i.e., $exp = \top$). Furthermore, some constraints on permissions should be satisfied: the declared permissions of ic' shall include the receiving permissions of the Intent, i.e., $n.\mathsf{rp} \subseteq ic'.\mathsf{decl}$; the declared permissions of ic should include both the permissions enforced by the target component and the permissions enforced by sending the Intent.

In addition to creating a target instance, execution of start_{std} also results in a control and data flow transfer from the current instance to the target instance—the control flow is transferred to the `onCreate()` callback of the target instance (modelled by $\lambda x.t$) and the value v encapsulated in the Intent is passed to the procedure $\lambda x.t$ by substituting the formal parameter x with v.

The [return] rule finishes the sub-Activity instance (i.e., ic), pops it from the current stack, and returns value v to the function $\lambda x.t'$ (modelling the callback `onActivityResult()`) of its parent-Activity instance (i.e., ic'). The sequence of commands $\mathsf{start}_{std}; \mathsf{return}(v)$ can simulate the data-fetching process implemented by the API `startActivityForResults()` and `onActivityResult()`.

Execution of $\mathsf{send}((n, v), p)$ requires that the receiver (i.e., ic') must satisfy the permissions p and $n.\mathsf{rp}$; while the sender (i.e., ic) must satisfy the permissions $n.\mathsf{sp}$ and the permissions enf enforced by the receiver. Likewise, execution of send will result in a control and data transfer—the callback `onReceive()` (modelled by $\lambda x.t$) of the receiver will be called and the value v is transferred to the formal parameter x.

The rules [update] and [query] are concerned with writing to and reading from a Provider instance with URI u. The current component instance (i.e., ic) must hold the "write" or "read" permissions (i.e., w and r) of the Provider instance. Update command $u!v$ will change the field value of the Provider instance, i.e., $\sigma'(ip) = \sigma(ip)[x \mapsto v]$. Dually, querying a value from a Provider instance will update the value of variable x in current active instance ic, i.e., $\sigma'(ic) = \sigma(ic)[x \mapsto v]$.

The rules [source] and [sink] state that the source or sink APIs are enabled if the calling instance (i.e., ic) satisfies the guarded permissions of these APIs.

4. The permission-carrying security information flow policy

Following the fundamental work Denning (1982), in this section, we first define the permission-carrying information flow policy, then extend the *state* (in Section 3.3) to *information state* to include security classes of variables in a state. Secondly, we present an information flow model for Android programs, and define *chains of information flows* that pass through multiple components. Finally, we present a set of certifying rules to authorize the information flows along with the Android-specific commands. These rules allows us to reason about the security of information flows and detect violations of the security policy.

Table 1
The transition rules for the commands of our language.

Rule	Premise i.e., $\mathcal{R}(s, s')$	Conclusion
[start_{std}]	ic' is fresh $\wedge h' = h \cup \{ic'\} \wedge \mathcal{S}' = ic'; \mathcal{S} \wedge$ $ic'.\mathsf{type} = \mathsf{activity}(\_, exp, fil, enf, \lambda x.t, \lambda x.t') \wedge ic'.\mathsf{type} = fil(n) \wedge$ $\neg sameApp(ic, ic') \implies exp = \mathsf{T} \wedge n.\mathsf{rp} \subseteq ic'.\mathsf{decl} \wedge (n.\mathsf{sp} \cup enf) \subseteq ic.\mathsf{decl}$	$\langle \mathsf{start}_{std}((n, v)), s\rangle \to \langle [v/x]t, s'\rangle$
[return]	$\mathcal{S} = ic; \mathcal{S}' \wedge ic' = \mathcal{S}'.\mathsf{top} \wedge$ $ic'.\mathsf{type} = \mathsf{activity}(\_, exp, fil, enf, \lambda x.t, \lambda x.t') \wedge$ $\neg sameApp(ic, ic') \Rightarrow enf \subseteq ic.\mathsf{decl}$	$\langle \mathsf{return}(v), s\rangle \to \langle [v/x]t', s'\rangle$
[send]	$ic'.\mathsf{type} = \mathsf{receiver}(\_, exp, fil, enf, \lambda x.t) \wedge ic'.\mathsf{type} = fil(n) \wedge p \subseteq ic'.\mathsf{decl} \wedge$ $\neg sameApp(ic, ic') \Rightarrow exp = \mathsf{T} \wedge n.\mathsf{rp} \subseteq ic'.\mathsf{decl} \wedge (n.\mathsf{sp} \cup enf) \subseteq ic.\mathsf{decl}$	$\langle \mathsf{send}((n, v), p), s\rangle \to \langle [v/x]t, s'\rangle$
[update]	$\exists ip \in h \bullet ip.\mathsf{type} = \mathsf{provider}(\_, exp, u, \_, w, x) \wedge$ $\neg sameApp(ic, ip) \implies exp = \mathsf{T} \wedge w \subseteq ic.\mathsf{decl} \wedge$ $\sigma'(ip) = \sigma(ip)[x \mapsto v]$	$\langle u!v, s\rangle \to s'$
[query]	$\exists ip \in h \bullet ip.\mathsf{type} = \mathsf{provider}(\_, exp, u, r, \_, v) \wedge$ $\neg sameApp(ic, ip) \implies exp = \mathsf{T} \wedge r \subseteq ic.\mathsf{decl} \wedge$ $\sigma'(ic) = \sigma(ic)[x \mapsto v]$	$\langle u?x, s\rangle \to s'$
[source]	$p \subseteq ic.\mathsf{decl} \wedge \sigma'(ic) = \sigma(ic)[x \mapsto v]$	$\langle \{p\}x := \mathsf{source}(v), s\rangle \to s'$
[sink]	$p \subseteq ic.\mathsf{decl}$	$\langle \{p\}\mathsf{sink}(v), s\rangle \to s$

4.1. Lattices of information flow policies

Definition 1. (Lattices) A lattice is a partially ordered set $(D, \sqsubseteq)$, where D is a set (finite or infinite), $\sqsubseteq$ is a *reflexive, transitive*, and *antisymmetric* partial order relation on D, and there exist both a *least upper bound* (*lub*) operator $\sqcup$ and a *greatest lower bound* (*glb*) operator $\sqcap$ for every pair of elements in D. Both $\sqcup$ and $\sqcap$ are *idempotent, commutative*, and *associative*, and there is an equivalent relationship between $\sqcap$, $\sqsubseteq$, and $\sqcup$:

$$\forall x, y \in D \bullet (x \sqcap y = x) \Leftrightarrow (x \sqsubseteq y) \Leftrightarrow (x \sqcup y = y).$$

A lattice $(D, \sqsubseteq)$ has a *top* element $\top$, and a *bottom* element $\perp$, such that for all $x \in D$, $\top \sqcup x = \top$, $\perp \sqcup x = x$.

In general, an information flow policy can be defined by a lattice $(SC, \sqsubseteq)$, where SC is a finite set of security classes, and $\sqsubseteq$ a binary relation partially ordering SC. For security classes $A, B \in SC$, the relation $A \sqsubseteq B$ means class A information is lower than or equal to class B information. Information is permitted to flow within a class or upward, but not downward or to unrelated classes; thus class A information is permitted to flow into class B if and only if $A \sqsubseteq B$. Accordingly, we shape the proposed flow policy as such a lattice.

Definition 2. (Permission-carrying information flow policy)
The permission-carrying information flow policy is a lattice $(SC, \sqsubseteq)$, where SC is 2^{P}, the powerset of permissions P; $\sqsubseteq$ is the subset relation $\subseteq$ on 2^{P}; $\sqcup$ and $\sqcap$ are of the set union $\cup$ and the set intersection $\cap$ on 2^{P} respectively; $\top = \mathsf{P}$, and $\perp = \emptyset$.

4.2. Information states

To investigate the security of information flows, we shall consider the security classes of variables in system states. For a variable x (a field or local variable) and a state s, we use x_s and $\underline{x}_s$ to denote the value and security class of x in state s respectively, if x exists in state s. We call $\underline{x}_s$ the *information state* of x under s. We shall write simply x and $\underline{x}$ when state s is clear from context or unimportant to the discussion.

The security class of a variable may be either *constant* or *varying*. With constant security classes, the security class of x is constant over the lifespan of x; that is, $\underline{x}_s = \underline{x}_{s'}$ for all states s and s' that include x. With varying security classes, the security class of x varies with states.

Assume command α yields a state transition $\langle \alpha, s\rangle \to s'$, and a variable x in s and a variable y in s'. Let $H(x|y')$ be the equivocation (conditional entropy) of x_s given $y_{s'}$, and let $H(x|y)$ be the equivocation of x_s given the value y_s in state s; if y does not exist in state s, then $H(x|y) = H(x)$, where $H(x)$ is the entropy (uncertainty) of x.

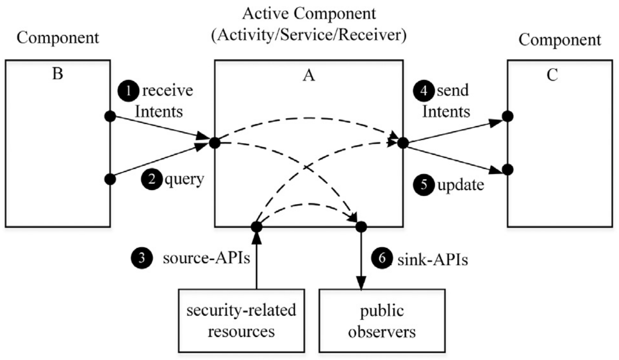


Fig. 7. A model for information flows in Android programs.

Definition 3. (Information flows) Execution of α in state s causes an *information flow* from x to y, denoted

$$x_s \overset{\alpha}{\rightsquigarrow} y_{s'},$$

if new information about x_s can be determined from $y_{s'}$, that is, if $H(x|y') < H(x|y)$. We shall write $x \overset{\alpha}{\rightsquigarrow} y$, or simply $x \rightsquigarrow y$, if there exist states s and s' such that execution of command α causes a flow $x_s \overset{\alpha}{\rightsquigarrow} y_{s'}$.

Definition 4. (Secure information flows) An information flow

$$x_s \overset{\alpha}{\rightsquigarrow} y_{s'}$$

is *secure* if and only if $\underline{x}_s \sqsubseteq \underline{y}_{s'}$; that is, the final security class of y is *at least as great as* the initial security class of x.

4.3. Information flows in Android programs

Fig. 7 gives a model to show all kinds of information flows in Android programs. Information flows in an instance of component A can be categorized according to their *entry*-points and *exit*-points. The entry-points of information flows can be these commands: ❶ receive communicating Intents; ❷ query something from Provider components; and ❸ source-APIs to obtain some information from security-related resources. Likewise, the exit-points of information flows can be of: ❹ send Intents to other components; ❺ update something to Provider components; and ❺ disseminate some information to the outside world with sink-APIs.

The entry-points and exit-points can be arbitrarily combined to form a total of 9 types of information flows within a component.

For example, an information flow "❶→❹" represents such a flow that takes "❶ receive Intents" as entry-point and "❹ send Intents" as exit-point. Apparently, information flows in different component instances can be concatenated to constitute *chains* of information flows, which pass through multiple component instances.

Definition 5. (Chains of information flows) Let IF_i denote a set of information flows inside an instance iC_i of component C_i. A set of chains of information flows over $\mathsf{IF}_i (1 \le i \le n)$, denoted by CIF, is defined as

$$\mathsf{CIF} = \mathsf{IF}_1 \hookrightarrow \mathsf{IF}_2 \hookrightarrow \cdots \hookrightarrow \mathsf{IF}_n,$$

where $\hookrightarrow \subseteq \mathsf{IF}_i \times \mathsf{IF}_{i+1}$ denotes a *concatenation relation* between information flows, which is defined as

$$\hookrightarrow = \begin{array}{c}\{(if_i, if_{i+1}) \mid if_i \in \mathsf{IF}_i \wedge if_{i+1} \in \mathsf{IF}_{i+1} \wedge \\ \text{the entry point of } if_{i+1} \text{ matches the exit point of } if_i\}.\end{array}$$

The predicate "*matches*" in the definition means: if the *exit*-point of if_i is a command sending an Intent, then the *entry*-point of if_{i+1} is that receiving this Intent; if the *exit*-point of if_i is an update command $u!v$, then the *entry*-point of if_{i+1} is the query command $u?x$.

In general, by abusing the symbol $\hookrightarrow$, we also denote a chain of information flows in CIF as $if_1 \hookrightarrow if_2 \hookrightarrow \cdots \hookrightarrow if_n$, where $if_i \in \mathsf{IF}_i$ $(1 \le i \le n)$.

4.4. Certifying rules for secure information flows

The security of chains of information flows relies on that of both information flows IF_i and concatenations $\mathsf{IF}_i \hookrightarrow \mathsf{IF}_{i+1}$. As IF_i are caused by basic commands (because they are confined inside single components), and the security rules for them had been achieved in Denning (1982), Huang et al. (2015), Sabelfeld and Myers (2003), Volpano et al. (1996), Nanevski et al. (2013), Calzavara et al. (2017), and Calzavara et al. (2016), here we only define the security rules for the Android-specific commands, which cause concatenations of information flows.

The rules are presented in Fig. 8. We use "[if-α]" to name a rule for the information flows caused by command α.

Let's explain the meaning of rule [if-start_{std}] as an example. If command $\mathsf{start}_{std}((n, v))$ successfully executes and triggers a state transition from s to s', and $\underline{v}_s \sqsubseteq ic'.\mathsf{decl}$ holds, then there is an authorized information flow from v_s to $x_{s'}$, and the information state of x in s' becomes $\underline{v}_s \sqcup n.\mathsf{rp}$. The condition $\underline{v}_s \sqsubseteq ic'.\mathsf{decl}$ prevents attacks of *confused deputy*. As variable x is a parameter of the term $\lambda x.t$, with a varying security class, we conservatively assign the least upper bound $\underline{v}_s \sqcup n.\mathsf{rp}$ to $\underline{x}_{s'}$; this assignment leads to $\underline{v}_s \sqsubseteq \underline{x}_{s'}$, ensuring the information always flows into equal or higher security classes. Note that, as v is transmitted via the Intent (n, v), permissions $n.\mathsf{rp}$ shall be involved in $\underline{x}_{s'}$ to reflect the restriction that the subsequent access to x requires these permissions as well. Similarly, the meanings of [if-return] and [if-send] can be explained likewise.

For the rule [if-update], updating the storage x with the value v shall forward the information state of v to x, thus we have $\underline{x}_{s'} = \underline{v}_s$ in the conclusion part of this rule; dually, querying a value from data storage v to variable x shall cause a transfer of information state from v to x, thus we have $\underline{x}_{s'} = \underline{v}_s \sqcup r$ in rule [if-query].

For the information flows associated with the source-APIs, we treat the information state of x as the precondition p of the source-APIs, meaning that the information in x is protected by p and the subsequent access to x should satisfy at least as great as permissions p. The rule [if-sink] states that an information flow from variable x to the public observer is authorized only if the information state of x is $\bot$, i.e., no security-related information involved

$$\frac{\begin{array}{l}\alpha = \mathsf{start}_{std}((n, v)) \\ \langle \alpha, s\rangle \to \langle [v/x]t, s'\rangle \quad \underline{v}_s \sqsubseteq ic'.\mathsf{decl}\end{array}}{v_s \overset{\alpha}{\leadsto} x_{s'} \text{ is authorized}, \quad \underline{x}_{s'} = (\underline{v}_s \sqcup n.\mathsf{rp})}\,[\text{if-start}_{std}]$$

$$\frac{\begin{array}{l}\alpha = \mathsf{return}(v) \\ \langle \alpha, s\rangle \to \langle [v/x]t', s'\rangle \quad \underline{v}_s \sqsubseteq ic'.\mathsf{decl}\end{array}}{v_s \overset{\alpha}{\leadsto} x_{s'} \text{ is authorized}, \quad \underline{x}_{s'} = \underline{v}_s}\,[\text{if-return}]$$

$$\frac{\begin{array}{l}\alpha = \mathsf{send}((n, v), p) \\ \langle \alpha, s\rangle \to \langle [v/x]t, s'\rangle \quad \underline{v}_s \sqsubseteq ic'.\mathsf{decl}\end{array}}{v_s \overset{\alpha}{\leadsto} x_{s'} \text{ is authorized}, \quad \underline{x}_{s'} = (\underline{v}_s \sqcup n.\mathsf{rp} \sqcup p)}\,[\text{if-send}]$$

$$\frac{\begin{array}{l}ip.\mathsf{type} = \mathsf{provider}(\_, exp, u, r, \_, v) \\ \langle u?x, s\rangle \to s' \quad \underline{v}_s \sqsubseteq ic.\mathsf{decl}\end{array}}{v_s \overset{u?x}{\leadsto} x_{s'} \text{ is authorized}, \quad \underline{x}_{s'} = \underline{v}_s \sqcup r}\,[\text{if-query}]$$

$$\frac{\begin{array}{l}ip.\mathsf{type} = \mathsf{provider}(\_, exp, u, \_, w, x) \\ \langle u!v, s\rangle \to s' \quad \underline{v}_s \sqsubseteq ip.\mathsf{decl}\end{array}}{v_s \overset{u!v}{\leadsto} x_{s'} \text{ is authorized}, \quad \underline{x}_{s'} = \underline{v}_s}\,[\text{if-update}]$$

$$\frac{\alpha = \{p\}x := \mathsf{source}(v) \quad \langle \alpha, s\rangle \to s'}{v_s \overset{\alpha}{\leadsto} x_{s'} \text{ is authorized}, \quad \underline{x}_{s'} = p}\,[\text{if-source}]$$

$$\frac{\alpha = \{p\}\mathsf{sink}(v) \quad \langle \alpha, s\rangle \to s' \quad \underline{v}_s \sqsubseteq \bot}{v_s \overset{\alpha}{\leadsto} outside \text{ is authorized}}\,[\text{if-sink}]$$

$$\frac{\langle x := const, s\rangle \to s'}{const_s \overset{x:=const}{\leadsto} x_{s'} \text{ is authorized}, \quad \underline{x}_{s'} = \bot}\,[\text{if-sanitize}]$$

Fig. 8. The certifying rules for secure information flows.

in x. This rule assumes that the outside observer is always public, thus having the lowest security class. Obviously, applying this rule guarantees no security-related information leak to the outside world via sink-APIs.

In addition, we present a rule [if-sanitize], which states that if a constant value is assigned to a variable x, then the information state of x is cleaned down to $\bot$, because a constant value has the lowest security class.

5. Formal definition of the threat model

In the rest of this paper, we attempt to apply the proposed policy to detect potential security attacks for Android programs. Two typical threats in Android programs, *confused deputy* and *collusive data leaks* (Tam et al., 2017) will be concerned in this section. We first present a formal model for the both threats with the notions introduced above, then address the soundness of the certifying rules (Section 4.4) for the freedom of such threats.

5.1. Threat model

As shown in Section 2.2, the general definitions for confused deputy and collusion have been given. But to accommodate the inter-component setting, we shall refine the previous definitions in terms of information flows.

In the light of inter-component information flows, a *confused deputy* is a threat associated with a sensitive inter-component ICC channel between a sender component A and a receiver component B, where A has an ICC exit leak and B does not have the permission to access the data from A; a *collusive data leak* is a threat associated with a sensitive ICC channel between a sender component A and a receiver component B, where A has an ICC exit leak and B leaks the received data from A via an ICC entry leak (Bosu et al., 2017).

Using the notions developed in the previous sections, we define the both threats formally as follows.

Definition 6. (Confused deputy) For a chain of information flows $if_1 \hookrightarrow if_2 \hookrightarrow \cdots \hookrightarrow if_n \in \mathsf{CIF}$, if there exists a concatenation $if_i \hookrightarrow if_{i+1}$ between component instances iC_i and iC_{i+1}, such that the *exit*-point of if_i transmits a value v to the *entry*-point of if_{i+1} via a state transition s to s' by executing one of the commands $\mathsf{start}_{std}((n, v))$, $\mathsf{return}(v)$, $\mathsf{send}((n, v))$, $u?x$, or $u!v$, and $\underline{v}_s \not\sqsubseteq iC_{i+1}.\mathsf{decl}$, then there is a threat of confused deputy over the chain.

Definition 7. (Collusive data leaks) A collusive data leak is a chain of information flows $if_1 \hookrightarrow if_2 \hookrightarrow \cdots \hookrightarrow if_n \in \mathsf{CIF}$, where if_1 begins with one of the source-APIs as its entry-point, and if_n ends with one of the sink-APIs as its exit-point.

These attacks come into force due to an inherent *vulnerability* in the Android system—lack of a mechanism to track and verify *information states* of variables in executions of programs. The built-in permission mechanism of the Android system cannot prevent these attacks, because it only protects security-related objects from illegal access, but does not regulate the information contained in variables and specify valid channels along which information flows.

5.2. Soundness of the certifying rules

The certifying rules (Section 4.4) can be used to prevent threats of confused deputy and collusive data leaks. We shall first address the soundness of these rules for the freedom of such threats. Soundness here means that applying these rules to information flows can definitely ensure the flows threat-free. This conclusion is expressed by the following theorem.

Theorem 1. *(Soundness of the certifying rules) For a chain of information flows* $if_1 \hookrightarrow if_2 \hookrightarrow \cdots \hookrightarrow if_n \in \mathsf{CIF}$, *if* if_i $(1 \le i \le n)$ *and* $if_i \hookrightarrow if_{i+1}$ $(1 \le i \le n-1)$ *are authorized by the certifying rules, then there is no threats of confused deputy and collusive data leaks posed in this chain.*

Proof. See proof of Theorem 1 of APPENDIX A. □

6. Static enforcement of the security policy

The permission-carrying information flow policy can be enforced dynamically at execution time, or statically at compile time. In this section, we propose a *static* enforcement of the policy by simply analysing program texts without any executions yet. The static enforcement is performed in the component-level rather than instance-level; we use a component class to *over-approximate* its instances. In this sense, this enforcement is *safe* but not *precise*; it may suffer from a degree of false positives.

To enforce the security policy to information flows, we shall first find the possible information flows in apps, then compute the security classes of variables in the flows and determine the security according to Definition 4. We separate this process into two steps, each of which will be elaborated in the subsections:

1. Find the information flows *inside individual components*, and compute the security classes of variables in the flows.
2. Find the information flows over ICC channels, and compute the security classes of variables in the chains of information flows.

Table 2
The matched *exit*-APIs and *entry*-APIs.

exit-APIs (Hotspots-APIs)	*entry*-APIs
`startActivity(i)`	`getIntent(i)`
`startActivities(i)`	
`startActivityForResult(i)`	`onNewIntent(i)`
`startService(i)`	`onHandleIntent(i)`
	`onStartCommand(i)`
`bindService(i)`	`onBind(i)`
`setResult(i)`	`onActivityResult(i)`
`sendBroadcast(i)`	`onReceive(i)`
`update(u,v)`	`query(u,x)`

6.1. Find information flows within individual components

Finding information flows and computing security classes of variables, in essence, can be formalized as a data-flow analysis problem (Nielson et al., 2004), which can be efficiently solved using the traditional *IFDS/IDE* framework (Reps et al., 1995; Sagiv et al., 1996). However, as some sophisticated and effective tools, such as FlowDroid (Arzt et al., 2014), have been implemented the IFDS/IDE algorithm especially for Android programs, we leverage one of the tools, FlowDroid, to find the information flows within a component.

FlowDroid is a state-of-the-art taint analysis tool for Android programs; it can discover all possible information flows from program points called the "sources" to program points called the "sinks" within a component. Note that the IccTA (Li et al., 2015) version of FlowDroid can additionally find the information flows over ICC channels, but here we only use the basic feature of FlowDroid to detect the flows within single components.

However, FlowDroid is based on the *two-level* security policy, only the labels $\top$ propagate along with information flows. Therefore, to compute the security classes of our policy, we shall investigate the relationship between the two-level security policy and ours, which is stated as the following theorem.

Theorem 2. *Let* $x_i \rightsquigarrow y$ $(1 \le i \le n)$ *be tainted information flows discovered by FlowDroid, then the security class of* y *is the least upper bound of the security classes of* x_i, *i.e.,*

$$\underline{y} = \bigsqcup_{1 \le i \le n} \underline{x_i}.$$

Proof. See proof of Theorem 2 of APPENDIX A. □

In particular, if every source point has the form $x_i := \{p_i\}\mathsf{source}()$, then $\underline{y} = \bigsqcup_{1 \le i \le n} p_i$ holds.

6.2. Find information flows over ICC channels

An ICC channel is a communicating relationship among components, which can be established whenever a component sends an Intent to another one, or a component updates a data storage which is later queried by another component.

For convenience, we call the APIs that send Intents or update data storages the *exit*-APIs, and the APIs that receive Intents or query data storages the *entry*-APIs. The exit-APIs and entry-APIs should be matched. Table 2 shows the exit-APIs, entry-APIs, and their matching relation.

We use *ICCGs* (Inter-Component Channel Graphs) to specify the channels among components. Before giving its definition, we introduce some auxiliary notions. We use a pair

$$\langle api, pos \rangle$$

to denote an API-call site, where *api* is the name of the API, *pos* is the program position where the API is called. We call the pair *exit*-

point if *api* is an exit-API; likewise, *entry*-point if *api* is an entry-API. For a component C, we use EXIT_C and ENTRY_C to denote the sets of exit- and entry-points of C; EXIT and ENTRY to denote the sets of overall exit- and entry-points of components in apps.

Definition 8. (ICCGs) An ICCG is a directed labelled multigraph

$ICCG = (N, E, L)$,

where

1. $N \subseteq \mathsf{CMP}$ is the set of nodes;
2. $E \subseteq N \times N$ is the set of edges;
3. $L : E \to \mathsf{EXIT} \times (\mathsf{INTENT} \cup \mathsf{URI}) \times \mathsf{ENTRY}$ is a function that maps each edge to a label;
4. For each edge $e = (C_1, C_2) \in E$, its label

   $L(e) = (\langle api_1, pos_1\rangle, iu, \langle api_2, pos_2\rangle)$

   should satisfy the constraints: $\langle api_1, pos_1\rangle \in \mathsf{EXIT}_{C_1}$, $\langle api_2, pos_2\rangle \in \mathsf{ENTRY}_{C_2}$, api_1 matches api_2, and iu is an Intent or URI object that serves as the argument of api_1.

For example, for an edge (C_1, C_2), its associated label may be of

$(\langle \mathtt{startActivity}, pos_1\rangle, i, \langle \mathtt{getIntent}, pos_2\rangle)$.

This edge together with its label constructs such a channel from C_1 to C_2: C_1 calls for `startActivity(i)` at pos_1 to send the Intent i, and C_2 receives this Intent by calling `getIntent()` at pos_2, where `startActivity()` is an exit-API, and `getIntent()` is an entry-API; the both are matched according to Table 2.

It is a challenge to find the possible channels (i.e., the edges) among components due to the extensive usage of the *implicit* Intent communications in Android programs. The target components are designated using values of particular string variables (such as *action* attribute of implicit Intents) instead of hard coded component names. Therefore, we need to resolve the values of these string variables with *string analysis* techniques (Bultan et al., 2017) to determine the target components. However, resolving the exact value of a string variable with static analysis techniques is *undecidable* (Bultan et al., 2017) in nature, so the result of the analysis for a given string variable at a particular program point is usually a set of values which *safely approximates* the exact value of the variable; in other words, the channels determined in this manner is *safe* but may be *imprecise*.

In this paper, we employ IC3 (Bosu et al., 2017; Octeau et al., 2016), a leading string analysis solver, to analyse the field values of Intent and URI objects in Android programs. The strength of IC3 is the power of determining the values of attributes (with string type) of an object that may have multiple fields, taking the correlations between these fields into account.

To determine the possible target components of a channel, we shall first determine the exit-points (also called the *hotspots*) of a component, then take the Intent or URI objects that serve as the arguments of the exit-APIs as the objects to be analysed. For a hotspot $\langle api, pos\rangle$, we use $V(\langle api, pos\rangle)$ to denote the set of all resolved values for the Intent or URI objects at point $\langle api, pos\rangle$. For a component C, we use $V(C)$ to denote the set of $V(\langle api, pos\rangle)$ at all hotspots of C, i.e.,

$V(C) = \{V(\langle api, pos\rangle) \mid \langle api, pos\rangle \text{ is a hotspot of } C\}$,

and $fil(C)$ the set of Intent filters of C. $V(C)$ can be achieved from the output of IC3 solver, and $fil(C)$ is obtained precisely by parsing the manifest file of an app.

The following Algorithm 1 illustrates the processes of finding all possible channels and further constructing an ICCG for a given app. (note: the algorithm is applicable to a group of apps as well.)

The algorithm proceeds in two major steps. First, as shown at Line 1, IC3 solver is used to extract the possible values of Intent or

Algorithm 1: Construct an ICCG for an App.

```
Input:
app: The app to be analysed
Output:
$ICCG = (N, E, L)$ of app
1  $V(C)$:=IC3-solver($app$) for each component $C \in app$;
2  foreach $C \in app$ do
3      foreach $\langle api, pos\rangle \in \mathsf{EXIT}_C$ do
4          if $api$ is not "update" then
5              EstablishIntentChannels($C$, $\langle api, pos\rangle$);
6          else
7              EstablishUriChannels($C$, $\langle \mathtt{update}, pos\rangle$);
8          end
9      end
10 end
11 procedure EstablishIntentChannels($C$, $\langle api, pos\rangle$)foreach
   $C' \in app \wedge C' \neq C$ do
12     foreach $i \in V(\langle api, pos\rangle)$ do
13         if $imatch(i, fil(C'))$ then
14             foreach $\langle api', pos'\rangle \in \mathsf{ENTRY}_{C'} \wedge$ $api'$ matches $api$ do
15                 $E := E \cup \{(C, C')\}$;
16                 $L((C, C')) = (\langle api, pos\rangle, i, \langle api', pos'\rangle)$;
17             end
18         end
19     end
20 end
21 procedure EstablishUriChannels($C$, $\langle \mathtt{update}, pos\rangle$)foreach
   $C' \in app \wedge C' \neq C$ do
22     foreach $\langle query, pos\rangle \in \mathsf{ENTRY}_{C'}$ do
23         foreach $u \in V(\langle update, pos\rangle), u' \in V(\langle query, pos'\rangle)$ do
24             if $umatch(u, u')$ then
25                 $E := E \cup \{(C, C')\}$;
26                 $L((C, C')) = (\langle \mathtt{update}, pos\rangle, u, \langle \mathtt{query}, pos'\rangle)$;
27             end
28         end
29     end
30 end
```

URI objects at all hotspots of each component C in *app*. The second step establishes two kinds of channels: Lines 11–20 establish the channels for Intent communications, and lines 21–30 establish the channels caused by `update` and `query` operations on data storages represented by URIs. If an Intent value i at an exit-point of C matches a filter of a component C' (at Line 13), then an edge shall be established (at Lines 14–17). There may be multiple edges between C and C', because multiple entry-points in C' may receive i. For the channels caused by updating and querying data storages, we make such a *conservative* assumption that if a data storage is updated by a component and queried by another one, then there may exist a channel between the both components. Notice that, this treatment is a *safe* approximation to the exact channels, because we disregard the orders of calling `update()` and `query()`.

There are three predicates *imatch* (at Line 13), *umatch* (at Line 24), and API-*match* (at Line 14) in the algorithm. *imatch* means the fields of an Intent i match the counterparts of the filters of a component; *umatch* means two URI objects (strings) match one another. The both matches support regular expression matching. The API-match means an exit-API matches an entry-API, illustrated in Table 2.

As an example, we use Fig. 9 to show the ICCG for an app. The ICCG involves four components $C1$, $C2$, $C3$, and $C4$. The dots in each component represent program points. The solid edges repre-

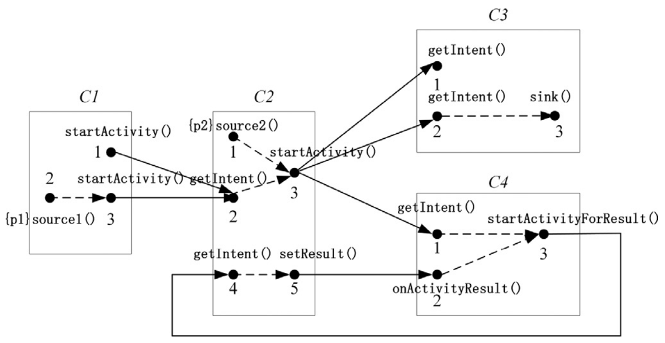


Fig. 9. An example of ICCGs and chains of information flows.

sent the channels among components, and the tail- and head-dots of these edges denote the exit- and entry-points of components respectively. Note that, an exit-point may issue multiple edges, because multiple entry-points may match the same exit-point; likewise, an entry-point may be reached by multiple edges, because one entry-point may receive Intents sent by multiple exit-points.

6.3. Solve the security classes of program points in chains of information flows

For simplicity, we use the form

$$\textit{exit}\text{-point} \hookrightarrow \textit{entry}\text{-point}$$

to denote edges, i.e., channels, in an ICCG, which connect the information flows inside individual components to form chains of information flows. To verify security of chains, we need to resolve the security classes of variables in chains.

Alternatively, we represent information flows within a component as a relation between program points

$$\langle \textit{source}\text{-api}, \textit{pos}\rangle \rightsquigarrow \langle \textit{sink}\text{-api}, \textit{pos}'\rangle.$$

For example, in the running example Fig. 4, the information flow

$$\langle \texttt{getDeviceId}, 2\rangle \rightsquigarrow \langle \texttt{startActivity}, 5\rangle,$$

means that the information obtained by calling `getDeviceId()` at Line 2, flows to the argument `i1` of `startActivity()` at Line 5, which exactly equals to the aforementioned form $\texttt{dev} \rightsquigarrow \texttt{i1}$. Therefore, we can equivalently consider security classes to be associated with program points, and say *security classes of program points* in what follows.

To form chains of information flows, we need to find the information flows involving exit- and entry-APIs. For this purpose, we think exit-APIs as a special kind of sink-APIs, entry-APIs as a special kind of source-APIs, and append them to the list of the original sink-APIs and source-APIs of FlowDroid (Rasthofer et al., 2014), resulting in the *extended* sources and sinks:

$$\mathsf{Source} := \textit{source}\text{-APIs} \cup \textit{entry}\text{-APIs}$$
$$\mathsf{Sink} := \textit{sink}\text{-APIs} \cup \textit{exit}\text{-APIs}.$$

With the aid of FlowDroid configured with Source and Sink, we can find the information flows, denoted IF_C, in each component C. All the flows in the components of an *app* are denoted as the set

$$IF = \bigcup_{C \in app} IF_C.$$

To compute the security classes of program points in chains of information flows, we need to establish the *dependence equations* of security classes by applying Theorem 2 to both the flows in components and the flows among components. The solution of the equations is exactly the security classes of program points.

We use Fig. 9 as an example to show how to establish and solve the dependence equations for chains of information flows. In this figure, the *dots* in each component denote program points, including all *exit*-, *entry*-, *source*-, and *sink*-points. For simplicity, we use the symbol C_{ij} to note the jth program point of component Ci, and use x_{ij} to denote the security class associated with C_{ij}.

Applying Theorem 2 and the certifying rules in Section 4.4 to the flows, we get such a set of dependence equations for the security classes:

$$\begin{cases} x_{11} = \bot & x_{12} = \{p1\} & x_{13} = x_{12} \\ x_{21} = \{p2\} & x_{22} = x_{11} \sqcup x_{13} & x_{23} = x_{21} \sqcup x_{22} \\ x_{24} = x_{33} & x_{25} = x_{24} & \\ x_{31} = x_{23} & x_{32} = x_{23} & x_{33} = x_{32} \\ x_{41} = x_{23} & x_{42} = x_{25} & x_{43} = x_{41} \sqcup x_{42} \end{cases}$$

The solution of the equations is:

$$x_{11} = \bot, x_{12} = x_{13} = \{p1\},$$
$$x_{21} = \{p2\}, x_{22} = \{p1\}, x_{23} = \{p1\} \sqcup \{p2\},$$
$$x_{31} = x_{32} = x_{33} = \{p1\} \sqcup \{p2\},$$
$$x_{41} = \{p1\} \sqcup \{p2\}, x_{42} = x_{43} = x_{24} = x_{25} = \{p1\} \sqcup \{p2\}.$$

Some points about the solution deserve to be noted:

1. The security class of C_{11} is $\bot$ (i.e., $\emptyset$), because no security-related information flows to C_{11}.
2. The information at C_{32} is leaked out through the flow $C_{32} \rightsquigarrow C_{33}$ in C3. As C_{33} is associated with a *sink*-API and $x_{33} \neq \bot$, according to rule [if-sink], this flow violates the security requirements of information flows.
3. In the cases of loops in chains, the equations for the loops are *recursive*, and the solution is the *least fixpoint* (*lfp*) of the equations. It is easy to show the existence of an *lfp* for the recursive equations due to the *monotonicity* of the operator $\sqcup$.

In summary, we use the following Algorithm 2 to conclude the procedure of computing the security classes of program points in chains of information flows.

Algorithm 2: Solve the security classes of program points.

```
Input:
app—The app to be analysed
Output:
The security classes of program points in chains of
information flows
1 Source := source-APIs ∪ entry-APIs;
2 Sink := sink-APIs ∪ exit-APIs;
3 $IF_C$ := FlowDroid($app$, Source, Sink) for each $C \in app$;
4 Construct ICCG $g = (N, E, L)$ for $app$ by Algorithm 1;
5 For each $C \in app$, establish dependence equations for the
  security classes by applying Theorem 2 and the certifying
  rules (Section 4.4) to the flows $IF_C$ and edges $E$;
6 The solution of the equations is the security classes
  associated with the program points;
```

6.4. Detect confused deputy and collusive data leaks

According to the threat model (Definitions 7 and 6), we need to verify the following two conditions to ensure the security of information flows in an ICCG:

1. *No confused deputy*. As we use a component type to safely approximate all instances of the component, the verification condition for the absence of confused deputy is

   $\forall exit\text{-point} \hookrightarrow entry\text{-point} \in E \bullet \underline{exit\text{-point}} \sqsubseteq \overline{C'}.\mathsf{decl}$,

   where E is the edges of the ICCG, $\underline{exit\text{-point}}$ is the security class associated with the *exit*-point, $\overline{C'}$ is the component of *entry*-point.
2. *No information leaks*, i.e.,

   $\forall C \forall src \rightsquigarrow sink \in IF_C \bullet \underline{src} \sqsubseteq \underline{sink}$,

   where *src* is a program point associated with one of the APIs in Source, *sink* is a program point associated with a *sink*-API, $\underline{src}$ and $\underline{sink}$ are of their security classes. As *sink* has a fixed security class, i.e., $\underline{sink} = \bot$ (See the rule [if-sink] in Section 4.4), this condition actually indicates $\underline{src} \sqsubseteq \bot$.

If the preceding conditions fail to hold, then threats of confused deputy or data leaks will be alarmed. If so, we can further search for the edges or paths causing these threats by using the following Algorithm 3.

Algorithm 3: Detect threats of confused deputy and data collusive leaks in chains of information flows.

```
Input:
app—The app to be detected
Output:
1. Report confused deputy or collusive leaks if the chains fail to
satisfy the verification conditions.
2. Leaks—the set of chains of information leaks.
Construct ICCG $g = (N, E, L)$ for app by Algorithm 1;
Use Algorithm 2 to solve the security classes of all program
points in the chains of information flows;
foreach $C \in app$ do
    $C$.decl:=parse(app.manifest);
end
foreach $exit \hookrightarrow entry \in E$ do
    if $\underline{exit} \not\sqsubseteq \overline{C'}.decl$ then
        report a "confused deputy" for edge $E$;
    end
end
$Leaks := \emptyset$;
foreach $src \rightsquigarrow sink \in IF_C$ do
    if sink is associated with a sink-API $\wedge \underline{sink} \neq \bot$ then
        report an "information leak";
        $Path := FindReachablePaths(sink)$;
        $Leaks := \{head \to sink \in Path \mid head$ is associated with a source-API$\}$;
    end
end
procedure FindReachablePaths(sink):
$WL := \{sink \to sink\}$;
$Path := \{sink \to sink\}$;
while $WL \neq \emptyset$ do
    select and remove a path $head \to sink$ from $WL$;
    foreach $head' \to head \in E \cup IF_C$ do
        if $head' \to sink \notin Path$ then
            $Path := Path \cup \{head' \to sink\}$;
            $WL := WL \cup \{head' \to sink\}$;
        end
    end
end
return Path;
```

At Lines 3–4, the algorithm gets the declared permissions of each component; this process is easy to implement because all declared permissions are explicitly configured in the *manifest.xml* file of *app*. The codes at Lines 6–10 and Lines 11–18 dedicate to detecting threats of confused deputy and information leaks respectively. If an information leak is detected, the procedure *FindReachablePaths*, at Lines 15–16, is called to further search for all the reachable paths to the sink (recorded in the set *Path*). The chains of information leaks to the *sink* is recorded in the set *Leaks*. Considering that loops may exist in the searching paths, we employ the backward worklist algorithm (Nielson et al., 2004), at Lines 19–31, to implement the searching process. The symbol $\to$ generalizes two kinds of information flows: the flows over channels (i.e., $\hookrightarrow$), and the flows (i.e., $\rightsquigarrow$) in a component.

As an example, the set of leaking paths in Fig. 9 is $\{C_{12} \to C_{33}, C_{21} \to C_{33}\}$. As a counterexample, the path $C_{12} \to C_{31}$ is not a leaking path, as the end point C_{31} is not a sink-API; likewise, nor is the path $C_{11} \to C_{32}$, as the start point C_{11} has the security class of $\bot$.

7. Experiments

7.1. Experimental environments and datasets

Hardware and software environments

The hardware environment of our experiments is as follows: the operating system is the 64-bit version of Windows 10, the processor is Intel(R) Core(TM) i7CPU@1.8GHz, the memory (RAM) is 16GB, and the hard disk is 1TB. The software environment is configured as follows: the development environment is Android 10 (SDK Platform 29) plus Android Studio 3.5.2 and java-1.8.0; the FlowDroid is the latest release 2.9, and the IC3's version is IC3-DIALDroid[2], an updated version of IC3[3].

Datasets

We evaluate both benchmark suites and real-world apps. The benchmark suites are collected from the datasets listed in Table 3. These datasets provide a ground truth result, on which various tools and approaches can be compared and evaluated.

TaintBench (Luo et al., 2022) is a new criteria for constructing real-world benchmark suites for static taint analysis. It provides 39 comparatively real-world samples (sum of 203 expected flows, average 20,000 lines of code for each) with 249 documented benchmark cases in total. The ground truth documented in TaintBench exhibits 23 intra-app ICC leaks, but no inter-app ICC leaks are documented.

DroidBench 3.0 is the most comprehensive benchmark suite to evaluate taint analysis tools for Android programs. Among the 174 test cases, DroidBench for IccTA branch provides 18 test cases for intra-app ICC leaks, and 11 test cases for inter-app ICC leaks.

DIALDroidBench is a dataset accompanied by DIALDroid (Bosu et al., 2017) toolset. Here, we use the suite in the TaintBench for DIALDroid branch as the benchmark, which comprises 30 samples. Although DIALDroid claims to aim for analysing large-scale inter-app communications, surprisingly no ground truth result for inter-app ICCs is presented yet in the DIALDroid branch.

ICC-Bench, introduced by Amandroid (Wei et al., 2018), provides 24 test cases, all of which dedicate to evaluating leaks by exploiting intra-app ICCs.

The real-world apps are selected from the *Tencent App Store*[4] and the *Google Play Market*. Every sample from these stores can be assumed *benign*, because it has undergone some single-app screenings imposed by some kind of vetting mechanisms (e.g., Google

[2] https://www.github.com/dialdroid-android/ic3-dialdroid.

[3] http://www.siis.cse.psu.edu/ic3/.

[4] https://www.app.qq.com.

Table 3
The benchmark suites in our experiments.

Benchmark	# Apks	# Intra-app ICCs	# Inter-app ICCs
TaintBench (Luo et al., 2022)	39	23	0
DroidBench3.0+ IccTA (Arzt et al., 2014; Li et al., 2015)	174	18	11
DIALDroidBench (Bosu et al., 2017)	30	26	0
ICC-Bench (Wei et al., 2018)	24	24	0
Sum	**267**	**91**	**11**

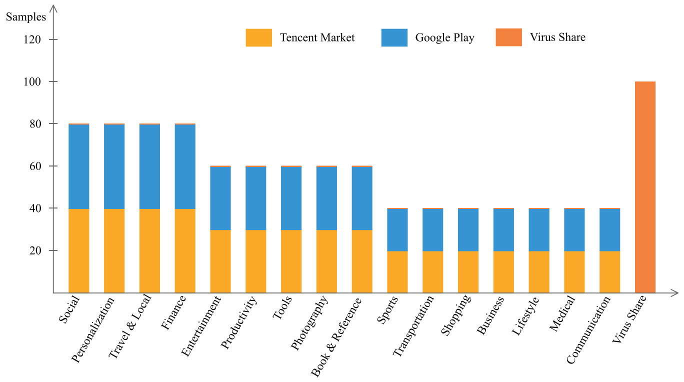


Fig. 10. The distribution of 1000 samples over categories.

Table 4
Some typical real-world samples from Tencent & Google App Store.

ID	Name	**Size**(MB)	Category	**Download** (Million)
1	tmall.apk	72.51	Shopping	250
2	taote.apk	110.0	Shopping	1800
3	jingdong.apk	118.0	Shopping	900
4	iqiyi.apk	63.66	Entertmt	120
5	douyin.apk	80.47	Entertmt	103
6	kuaishou	130	Entertmt	215
7	TED.apk	40.80	Tools	303
8	beidanci.apk	96.83	Tools	2910
9	JustDating.apk	32.6	Social	1315
10	qqHD.apk	35.58	Social	3868
11	momo.apk	96.05	Social	620
12	BaiduMap.apk	118	Travel	400
13	dida.apk	163	Travel	6677
14	paypal.apk	114	Finance	780
15	jdfinance.apk	107	Finance	110
16	gongshangbank.apk	163	Finance	140

Bouncer). To achieve a complete and convincing dataset, we additionally collect a proportion of *malicious* samples from Virus Share[5].

A total of 1000 real-world samples are collected, which cover 16 main categories. In each category, a equal number of samples are captured from the Tencent Market and Google Play. The distribution of the samples over categories is shown in Fig. 10, in which a little more samples fall within the categories *Social, Personalization, Travel & Local*, and *Financial*, as being intimately concerned with the customer privacy, and more likely pose a threat of information leaks. Table 4 shows a portion of typical samples and their main features.

7.2. Experimental evaluation

The evaluation intends to investigate the following research questions:

RQ1 How to configure FlowDroid to find information flows in components?

RQ2 How to know the permissions associated with source-APIs, security-related Intents, and security-related Providers?

RQ3 How to evaluate the power of IC3 to precisely resolve field values of Intent and URI objects?

RQ4 How to evaluate the performance of constructing ICCGs and certifying security of information flows?

RQ5 How to evaluate the power of our approach to detect typical security threats, compared with state-of-the-art tools?

RQ6 How to show the detection mechanism in a real-world case study?

In the sequel, we address each research question in detail.

RQ1 How to configure FlowDroid to find information flows in components of real-world apps?

We configure FlowDroid with various options to gain a tradeoff between performance and precision. In each option, we prefer to select the items favourable for gaining more accurate and complete analysis results, even though they may cost more time or memory consumptions. The main configuration options and selected items are listed in Table B.11 of APPENDIX B, where the items labelled with checked boxes ☑are of the selected ones.

RQ2 How to obtain the permissions associated with source-APIs, security-related Intents, and security-related Providers?

To implement our approach, we shall first determine the permissions associated with security-related APIs, Intents, and URIs. This issue had been addressed by Pscout[6] project, which yielded a set of mappings from security-sensitive APIs, Intents, and URIs to their required permissions. A typical part of these mappings are illustrated in the Tables B.12, B.13, and B.14 of APPENDIX B.

The guarded permissions associated with a security-sensitive API mostly consist of only a single permission, but some APIs may require multiple permissions. For example, the API "getLookupUri" in Table B.12, is guarded by four permissions. In Table B.13, the sending/receiving permissions of an Intent action are headed by symbols "S" and "R" respectively. The presences of "NONE" represent no permission is required. Similarly, in Table B.14, we put "R" and "W" in front of reading and writing permissions of a provider URI respectively.

RQ3 How to evaluate the power of IC3 to precisely resolve field values of the Intent and URI objects at program hotspots?

In construction of ICCGs, we highly require a powerful string analyser to accurately resolve field values of the interesting Intent or URI objects at hotspots. Though it is undecidable to statically resolve the exact value of a string variable, we always expect the resolved values as precise as possible, because the more precise the field values are resolved, the smaller number of channels (i.e., edges) need to be constructed, and then the less false alarms would be arisen.

To thoroughly evaluate IC3's ability to resolve string values, we design a set of test cases according to two orthogonal factors:

1. the patterns of assigning values to `Intent`'s or `URI`'s fields;
2. the string operations commonly used to manipulate string values, such as +, `concat`, `append`, and `subString`.

[5] http://www.virusshare.com.

[6] https://www.github.com/dlgroupuoft/PScout.

Table 5
Evaluation results for various patterns and string operations.

Pattern	①	②	③	④	⑤	⑥	⑦	⑧	⑨
new-Intent-pattern	✓	☑	✓	☑	☑	☑	✗	✓	☑
setAction-pattern	✓	☑	✓	☑	☑	☑	✗	✓	☑
setComponent-pattern	✓	☑	✓	☑	☑	☑	✗	✓	☑
setClassName-pattern	✓	☑	✓	☑	☑	☑	✗	✓	☑
.class-pattern					✓				
getClass-pattern					✓				

①: *const* ②: *var* ③: *const* + *const* ④: *var* + *const* ⑤: *var*.concat(*const*) ⑥:*var*.concat(*var*) ⑦: *const*.subString() ⑧:StringBuilder.append(*const*) ⑨: StringBuilder.append(*var*) ✓: resolved completely ☑: resolved partially ✗: unresolved.

The evaluation is performed on the cross product of the both factors, and the result is illustrated in Table 5. The first column represents six patterns of assigning values to `Intent` or `URI` object fields; the first row represents nine cases, from ① to ⑨, which manipulate strings to be passed as arguments to the corresponding class methods or constructors.

We take `action` field of `Intent` object as an example to explain the assignment patterns (Table B.15 of APPENDIX B). For an implicit Intent, `action` can be set by calling one of the constructors of `Intent`, or `setAction`() class method. While for an explicit one, its target component can be set by calling `setComponent`(), `setClassName`(), or with class name explicitly. All these patterns require string type arguments passed to the corresponding class methods, we therefore shall test whether IC3 solver is able to successfully resolve these argument values or target class names for all these patterns.

The typical string manipulations are presented in Table B.16 of APPENDIX B. Here we only take `setAction`(*s*) as an example. The argument *s* can be a string constant, variable, or constructed using various string operations, therefore we shall test whether IC3 solve is able to successfully resolve the arguments for all these manipulations.

The test result for the samples of DroidBench 3.0 and ICC-Bench is shown in Table 5, from which we see that:

1. For the string manipulations that only involve string constants, IC3 solver is able to completely resolve all argument values (marked by ✓).
2. But for some commonly used string operations, such as `subString`(), IC3 solver fails to resolve in all cases even only with string constants (marked by ✗).
3. For the string manipulations involving a string variable, IC3 is able to track dataflows of the variable and resolve its value successfully; but for a variable whose value depends on some inputs from the user, IC3 fails to work. For example, the `action` field of Intent *i* is set by the statement
i.`setAction`("android.intent.action."+ *x*);
where the value of *x*, say, is entered by the user via a text field, IC3 resolves `action` value as a regular expression "android.intent.action.*", namely only the prefix is retrieved. We call this situation *resolved partially*, and marked by the symbol ☑.
4. For an explicit Intent object, its target component name can be resolved completely, as illustrated by the rows of "`.class-pattern`" and "`getClass-pattern`" in Table 5.

The test results indicate that IC3 is able to accurately resolve the field values of `Intent` and `URI` objects in most cases. It thus can be leveraged to find the target components of the Intent objects.

RQ4 How to evaluate the performance of constructing ICCGs and certifying security of information flows?

Sources and sinks. As shown in Algorithm 2, FlowDroid shall be configured with a list of Source and Sink to find information flows in a component. The list is generated by appending the *source*- and *sink*-APIs with the *entry*- and *exit*-APIs. The former had been collected in SuSi project (Rasthofer et al., 2014); some typical APIs in different categories are listed in Tables B.17 and B.18 of APPENDIX B. The latter have been given in Table 2.

Construct ICCGs and certify chains of information flows. Following the Algorithms 1 and 3, we perform the Algorithms for the overall 1267 samples (a sum of 267 from the benchmark suites, and 1000 from real-world), then find the chains of information flows and detect potential security threats. The test result (average values) is illustrated in Table 6.

We measure an app from seven aspects: (1) The number of used-permissions (**# Perms**); (2) The number of components (**# Comps**) including all four kinds of components (Activity, Service, Receiver, and Provider) defined in an app; (3) The number of exposed components (**# Exposed comps**) in each app; (4) The number of components containing information flows (**# Comps containing IFs**); it is achieved by applying FlowDroid to each component; (5) The number of exit-points (**# Exit-points**) and entry-points (**# Entry-points**) of the components involving information flows; (6) The performance in terms of runtime and memory consumptions, which count for both constructing ICCGs and detecting security threats in information flows.

In all 1267 apps, 61 apps (all come from real-world samples) overflow in time or memory threshold (our time threshold: 3600s, memory threshold: 4GB), so only 1206 apps are available for evaluation. The effectiveness of the approach is measured in terms of:

1. **# Intra-app confused deputy**: the total number of confused deputy detected in all 1206 apps;
2. **# Inter-app confused deputy**: the total number of confused deputy detected in all 1206 × 1206 pairs of apps;
3. **# Intra-app leaks**: the total number of collusive leaks detected in all 1026 apps;
4. **# Inter-app leaks**: the total number of collusive leaks detected in all 1206 × 1206 pairs of apps.

#Intra-app confused deputy/leaks aim to count the number of threats discovered within individual apps, whereas the **#Inter-app** counterpart counts for the numbers of threats in pairs of apps. Note that we only consider pairwise combinations when detecting inter-app threats; combinations of three or more apps are omitted.

In general, there is no need to detect confused deputy within an app, as the intra-app interactions do not require permission checking. However, consider a fair number of third-party SDKs have been involved in real-world apps (an average of 8.9 SDKs in an app according to a report on Chinese app mar-

Table 6
The average values for 1267 samples (Time threshold: 3600s, Memory threshold: 4GB).

# Sample	# Perms	# Comps	# Exposed comps	# Comps containing IFs	# Exit-points	# Entry-points	**Runtime**(s)	**Memory**(MB)
1267	45	130	19	23	32	37	1120	1375
# Intra-app confused deputy: 461 in 1206 apps **# Inter-app confused deputy**: 539 in 1206 × 1206 pairs of apps								
# Intra-app leaks: 392 in 1206 apps **# Inter-app leaks**: 178 in 1206 × 1206 pairs of apps								

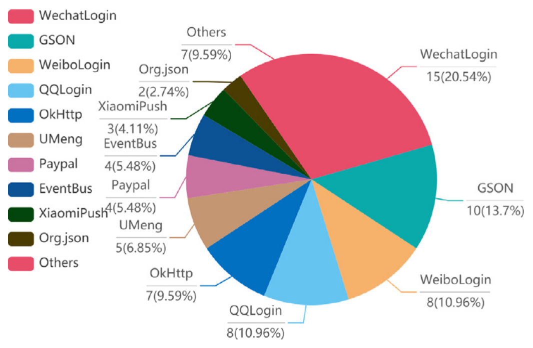


Fig. 11. The distribution of the SDKs in 1206 samples.

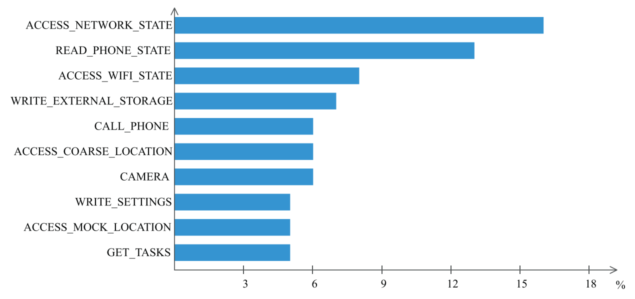


Fig. 12. The top 10 common escalated permissions within apps.

ket (OPPO&Deloitte, 2021)), we thus mainly investigate the over-privileged SDKs in the samples. We find a total of 563 intra-app confused deputy in 1206 apps. The distribution of the SDKs involved in these samples is illustrated in Fig. 11. The top 10 common escalated permissions and their percentages on the total number of the confused deputy are shown in Fig. 12.

Not all confused deputy and collusive leaks really indicate true security threats. The reasons are twofold: first, the static approach unavoidably brings forth a certain degree of false positives, which may overestimate the security threats; second, some information leaks may be permitted by the granted permissions. For example, in apps of LBS (Location-Based Service) category, such as *dida.apk*, location information is allowed to flow out, as the permissions about locations are granted to the apps and obtaining such information is a normal action for these apps.

In addition, only a small number of collusive data leaks were discovered in all 1206 × 1206 pairs of apps. We speculate the main reason for this phenomenon is that the samples crawled from Tencent App Store and Google Play are benign in general; it is unlikely to yield malicious information flows by pairing two benign apps. A larger number of malicious information flows might be found under these circumstances that one app in a pair is malicious and the other is benign.

RQ5 How to evaluate the power of our approach compared with other leading works?

We compare our approach with the state-of-the-art works to evaluate its strength to discover typical security threats. The selected leading works include FlowDroid (Arzt et al., 2014), IccTA (Li et al., 2015), AmanDroid (Wei et al., 2018), DIALDroid (Bosu et al., 2017), and TaintBench (Luo et al., 2022). We characterize their main detection features, as shown in Table 7, from two orthogonal dimensions: the typical threats to be detected (**confused deputy** or **information leaks**) and the scope to work on (**intra-app** or **inter-app**).

Table 7
The detection features of the state-of-the-art works.

Work	Confused deputy		Information leak	
	intra-app	inter-app	intra-app	inter-app
FlowDroid	–	–	✓	–
IccTA	–	–	✓	✓
AmanDroid	–	–	✓	✓
TaintBench	–	–	✓	✓
DIALDroid	–	✓	✓	✓
Ours	✓	✓	✓	✓

✓: support –: unsupport.

The traditional version of FlowDroid aims to detect the tainted information flows within a single app. IccTA and AmanDroid are designed to find inter-component information flows (can be intra-app or inter-app), not used to detect threats of confused deputy. TaintBench acts as not only a comprehensive benchmark for taint flow analysis, but also a framework to query taint flows for apps by synthesising the analysis results yielded by multiple tools. DIALDroid, more alike ours, can examine the threats of both information leaks and confused deputy for single apps or pairs of apps, but fails to detect intra-app confused deputy yet.

According to the detection features of these works, we design the following three sets of experiments:

1. **Exp1**: Evaluate the performance of detecting *information leaks*, on the benchmark suites (see Table 3), for FlowDroid+ IccTA, AmanDroid, DIALDroid, and ours. The motivation behind this design is that: (i) the ground truth documents are available for the information flows of the benchmark suites (note: the ground truth for the confused deputy is absent completely); (ii) all of the works support the feature of detecting information flows, thus they are comparable to each other.
2. **Exp2**: Compare ours with DIALDroid, on the suite of 1000 real-world apps (see Fig. 10), to evaluate their detection power for *inter-app confused deputy* and *information leaks*. The philosophy is that: DIALDroid supports almost all features with ours, so the both are comparable to each other. However, the main challenge for this experiment is the absence of ground truth for the real-world samples. To this end, we offer two solutions to alleviate this issue (i.e., confirm whether a detected result is really *true* or *false*): (i) manually inspect the result to verify its truth; or (ii) drive the target app to run automatically and confirm the truth at runtime. The fulfilment of this solution needs the aids of the *Xposed* framework (Xposed, 2021) and the *FastBot* (Cai et al.,

Table 8
Confusion matrix.

Actual class	Predicated class	
	yes	no
yes	TP (leaks were correctly detected)	FN (leaks were missed)
no	FP (legitimate flows were incorrectly recognized as leaks)	TN (legitimate flows were let pass by the detector)

Table 9
The experimental result for **Exp1** on the benchmark suites.

Measure		FD	AD	DD	Ours
Intra-app leak	TP	87	80	87	87
	FP	11	13	11	5
	FN	4	7	2	2
	p	89%	86%	89%	94%
	r	95%	92%	97%	97%
	$F1$	92%	89%	93%	95%
Inter-app leak	TP	65	53	65	68
	FP	15	17	11	5
	FN	5	7	5	4
	p	81%	76%	85%	93%
	r	92%	88%	92%	94%
	$F1$	86%	82%	88%	93%

FD: FlowDroid+ICCTA AD:AmanDroid DD:DIALDroid TP: True Positive FP: False Positive FN: False Negative p: Precision =TP/(TP+FP) r: Recall =TP/(TP+FN) $F1$: F-measure $= 2pr/(p+r)$.

Table 10
The experimental results for **Exp2** and **Exp3** on the real-world samples.

Work	Measure	Confused deputy		Information leaks	
		intra-app	inter-app	intra-app	inter-app
DD	TP	–	483	353	153
	FP	–	93	61	32
	p	–	84%	85%	83%
Ours	TP	415	485	358	155
	FP	46	50	32	18
	p	90%	91%	92%	90%

2020). The former is a hook framework for the Android system, allowing us to instrument pieces of codes into interesting program points without any change or recompiling the original Apk. By means of this framework, we insert codes into the predefined source-, sink-, exit-, and entry-APIs, to track and verify the truth of information leaks. The latter is an automated testing tool for apps, able to drive an app to run automatically while gaining a high code coverage in less runtime. Integrating Xposed and FastBot together allows us to achieve a high degree of automation to confirm the reported alarms at runtime.

3. **Exp3**: For the threats of *intra-app confused deputy*, as aforementioned, in general, there is no need to detect them within an app, because the intra-app interactions do not require permission checking. So we mainly investigate the confused deputy over the third-party SDKs embedded in the samples. Since DIALDroid cannot support the detection for intra-app confused deputy so far (see Table 7), it is not involved in this experiment.

The experimental result for **Exp1** is presented in Table 9. The evaluation is based on the standard metrics, including *precision*, *recall*, and *F1-score*, and compares these metrics with the related approaches to demonstrate the detection effectiveness of the approach. The comparison is performed in terms of the elements (TP, FP, and FN) in the *confusion matrix* (Han et al., 2022) (see Table 8), each of which is further refined as *intra-app* and *inter-app* versions.

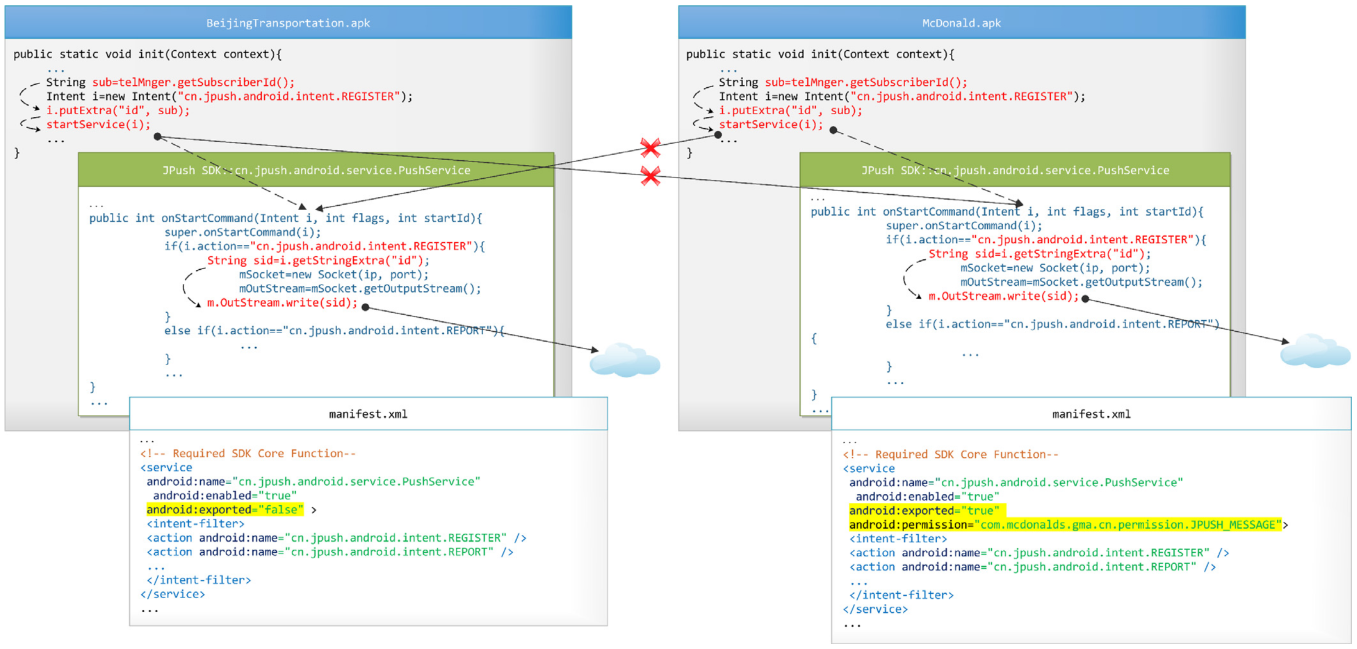
From the result, we conclude that: (i) Our approach outperforms, in *precision, recall*, and *F1* score, the other three works for both intra- and inter-app information leaks; (ii) For each work, its recall rate is superior than its precision rate, reflecting that the false negatives is smaller than false positives. This phenomenon may be caused by the *static analysis* nature of these works—all these works employ static program analysis techniques to find the potential information leaks in apps, which would unavoidably *overestimate* the actual leaks in some extent.

The experimental results for **Exp2** and **Exp3** are exhibited in Table 10. The comparison is measured in terms of the number of detected confused deputy and the number of detected information leaks, each of which is further refined as intra-app and inter-app versions. Note that, only the metric *precision* is involved in this experiment; *recall* is neglected, because FN (False Negative) is unknown due to the absence of the ground truth documents for the real-world samples.

From the results we see that: the value of TP reported by our approach is *comparable* to (in fact, slightly higher than) that of DIALDroid, whereas the value of FP of our approach is greatly less than that of DIALDroid. This fact explains why our precision is much better than that of DIALDroid. We investigate the reason and find that the following three factors account for the high precision of our approach:

1. The component's attribute "*exported*" (i.e., *exp* in the definition of components in Fig. 5) is considered in discovering potential information flows among inter-app components. The configuration "`exported=false`" for a component (Activity, Service, Receiver, or Provider) of an app means that the component is confined in the app, any interaction initiating from other apps is prohibited. In contrast, DIALDroid leaves this attribute out of consideration, thinking all the interactions with matched Intents were feasible. This practice will definitely bring in a large number of false positives in finding true information flows.
2. The attribute *enforced permission* (i.e., *enf* in the definitions of apps and components in Fig. 5) is involved in the discovery of information flows. In fact, many apps use permissions (especially user-defined permissions) to control the communication between apps, or protect their inner services and shared data from external access (Li et al., 2021). For example, only the apps with the `BROADCAST_RECEIVER`[7] permission can send a broadcast to the broadcast receiver of `com.tencent.portfolio`. This concern allows us to exclude the information flows that cannot meet the enforced permissions from the detected flows, thus achieving a more accurate result.
3. The permissions associated with Intents are considered. These permissions consists of two kinds: the permissions associated with sending and receiving Intents (i.e., `sp` and `rp` in Fig. 5) and the permissions attached with Intents in $\mathsf{send}(i, p)$ operations (see Fig. 6). A valid Intent communication should satisfy both of the permissions. Full consideration of these permissions enables us to find the information flows more precisely.

[7] Full name is `com.tencent.portfolio.permission.BROADCAST_RECEIVER`.


BeijingTransportation.apk
JPush SDK::cn.jpush.android.service.PushService
manifest.xml
McDonald.apk
JPush SDK::cn.jpush.android.service.PushService
manifest.xml

```
public static void init(Context context){
      ...
      String sub=telMnger.getSubscriberId();
      Intent i=new Intent("cn.jpush.android.intent.REGISTER");
      i.putExtra("id", sub);
      startService(i);
      ...
}
```

```
...
public int onStartCommand(Intent i, int flags, int startId){
            super.onStartCommand(i);
            if(i.action=="cn.jpush.android.intent.REGISTER"){
                  String sid=i.getStringExtra("id");
                        mSocket=new Socket(ip, port);
                        mOutStream=mSocket.getOutputStream();
                  m.OutStream.write(sid);
            }
            else if(i.action=="cn.jpush.android.intent.REPORT"){
                        ...
            }
            ...
}
...
```

```
...
<!-- Required SDK Core Function--
<service
 android:name="cn.jpush.android.service.PushService"
  android:enabled="true"
 android:exported="false" >
 <intent-filter>
 <action android:name="cn.jpush.android.intent.REGISTER" />
 <action android:name="cn.jpush.android.intent.REPORT" />
 ...
 </intent-filter>
</service>
...
```

```
public static void init(Context context){
      ...
      String sub=telMnger.getSubscriberId();
      Intent i=new Intent("cn.jpush.android.intent.REGISTER");
      i.putExtra("id", sub);
      startService(i);
      ...
}
```

```
...
public int onStartCommand(Intent i, int flags, int startId){
            super.onStartCommand(i);
            if(i.action=="cn.jpush.android.intent.REGISTER"){
                  String sid=i.getStringExtra("id");
                        mSocket=new Socket(ip, port);
                        mOutStream=mSocket.getOutputStream();
                  m.OutStream.write(sid);
            }
            else if(i.action=="cn.jpush.android.intent.REPORT")
{
                        ...
            }
            ...
}
...
```

```
...
<!-- Required SDK Core Function--
<service
 android:name="cn.jpush.android.service.PushService"
  android:enabled="true"
 android:exported="true"
 android:permission="com.mcdonalds.gma.cn.permission.JPUSH_MESSAGE">
 <intent-filter>
 <action android:name="cn.jpush.android.intent.REGISTER" />
 <action android:name="cn.jpush.android.intent.REPORT" />
 ...
 </intent-filter>
</service>
...
```

Fig. 13. A case study.

Note that, the concerns for above three factors have been encoded in the transition rules of our language (see Table 1) and the certifying rules (see Fig. 8) for secure information flows. We claim that it is very necessary to fit program behaviour models in the detection of security attacks, because only by executing programs are attacks able to come into force. Different from general Java programs, Android programs have permissions and varied configurations as the most significant features, which should be involved in the semantic model of programs and applied to the detection of attacks. In contrast, IccTA, AmanDroid, and DIALDroid only considered Intent-matching between components, but ignored permission and configuration constraints in finding channels among components. This point can explain why our approach outperforms the others.

RQ6 How to show the detection mechanism in a real-world case study?

We use the following case study to show the detection mechanism and its superiority in real-world applications. Two samples in our dataset, named *BeijingTransportation* and *McDonald*, embed a third-party SDK called *JPush*[8] to implement the pushing feature respectively, see Fig. 13.

Before initiating the pushing functionality, an app should first register an account in the JPush Web Portal, which needs private information about the user's subscriberId. Either app calls for `startService()` to send an Intent to `PushService` with action `cn.jpush.android.intent.REGISTER`. The both `PushServices` are configured differently, see the shaded lines in the manifest files of the apps. Once receiving the Intent with `REGISTER` action, `PushService` will unmarshal the subscriberId from the Intent, and send it to the JPush Portal by calling a sink method `OutStream.write()`.

In this case, there is one intra-app information leak in either of the apps (see the dashed lines within an app). These leaks can be detected successfully using both our approach and DIALDroid. However, the inter-app information leaks, reported by DIALDroid, are actually false positives. DIALDroid only concerns Intent-matching in finding potential communicating channels, thus thinks the flow from McDonald's `startService()` to BeijingTransportation's `onStartCommand()` a valid one. Nevertheless, considering the configurations of `PushService`, the both inter-app flows are invalid (labelled with the cross symbols). Our approach can correctly exclude these false positives, thus achieving a more precise detection result.

8. Discussion

8.1. Two-level security policy vs. multiple-level policy

Traditional information flow policy is two-level; every variable is labelled tainted ($\top$) or cleaned ($\bot$). This policy is a high-level abstraction for a wide range of practical problems, such as detecting information leaks (Arzt et al., 2014) and discovering resource usage vulnerabilities (Wang et al., 2020) in programs. However, this policy might be too *coarse-grained* in some application scenarios, as the two cases shown in Section 1. In these cases, we want to know not only which variables are tainted, but also tainted by what information; that is, we want to distinguish the *sources* of the tainted information flows.

To this end, we need a *refined* policy with multiple security clearances. In Android apps, permissions can be regarded as a qualified candidate for this purpose, because (1) different security-related sources are usually guarded by different permissions; (2) permissions provide a high-level abstraction for the security property of sources; (3) the subset lattices of permissions enjoy properties of *monotonicity* and *distributivity* (Aho et al., 2007), which allow us to precisely compute security classes using the traditional data-flow analysis framework.

8.2. Static vs. dynamic enforcement

Generally, an information flow policy can be enforced either *statically* (Arzt et al., 2014; Gordon et al., 2015; Pan et al., 2018; Tripp et al., 2013; Wei et al., 2018) at compile time, or *dynamically* (Enck et al., 2014; Sun et al., 2016) at execution time. The proposed approach in this paper falls into the *static* category. The static enforcement mechanism proceeds without executing programs; it is very useful in some scenarios. For example, in some Cloud vetting

[8] https://www.jiguang.cn/push.

platforms for mobile apps, such as Baidu Anquan (Baidu, 2021) and 360 Brain of Security (360, 2021), a large number of apps (about 400,000∼1800,000 apps per year) need to be examined before put on shelves. Apparently, it is nearly impossible to run each app and find potential vulnerabilities manually, even automatically, in a given short time.

Our proposed enforcement works in the component-level, i.e., use a component type to abstract all its possible instances. This treatment greatly reduces dimensions of the problem. On the other hand, our approach suffers from a degree of *imprecision* due to adoption of the approximation techniques—we use a component type over-approximates its instances, *meet* the security classes of a program point reached from different execution paths, and utilise the static string solver to find the potential channels among components. These over-approximation techniques are *safe* but may introduce somewhat *imprecision* in the analysis results. However, this shortcoming can be remedied by employing highly precise static techniques or integrating with dynamic techniques to achieve a more accurate result.

8.3. Superiority vs. limitation

Our approach is applicable to both *intra-app* and *inter-app* threat detections. The threat of inter-app confused deputy must be checked, but, in general, the intra-app confused deputy do not need to be detected, because all components defined in an app share the same declared permissions of the app, consequently the verification condition $\underline{\textit{exit}\text{-point}} \sqsubseteq C'.\mathsf{decl}$ (Section 6.4) holds trivially. However, in the cases where one or more third-party SDKs are integrated in an app, the detection for confused deputy becomes mandatory. As shown in Section 1, some SDKs may gain more privileges than what they really need, bringing in potential risks to the app.

The language proposed in Section 3.2 nearly captures all major features of Android programs; but one feature, granting (revoking) a permission to (from) an app, has not been touched so far. In fact, an app can be granted a read or write URI permission by calling the API `grantUriPermission()` or attaching a flag `FLAG_GRANT_READ(WRITE)_URI_PERMISSION` to an Intent object. For simplicity, here we omit this feature, and assume the declared permissions of an app can be determined statically from the manifest file and keep invariable in its lifecycle. However, it should be noted that the absence of this feature does not impair the safety of our approach, because our policy satisfies *monotonicity*, namely, less permissions of a component only increase the false positives of the detection, but never increase false negatives.

8.4. Performance bottlenecks

A real-world app usually consists of a large number of components (134 components in average), thus the performance is one of the chief concerns of our approach. In fact, we only need to concern the components involving information flows in the analysis. The test result shows that only a small number of components (about 18%) contain information flows; this fact allows us to greatly reduce the number of components in the construction of ICCGs, thus makes our approach scalable.

Most of the apps can be processed successfully within a given time and memory threshold, but for some samples, the processing was overflowed. Our observation shows that, much time and memory consumptions were spent on finding information flows in components with FlowDroid and resolving string fields with IC3 solver. Both FlowDroid and IC3 solver use IFDS/IDE (Reps et al., 1995; Sagiv et al., 1996) as the underlying data-flow algorithm to compute data facts at every program point. This algorithm will cause a large consumption of time and memory if configured with the features that promote analysis precision, especially for large apps (greater than 10MB in general). This fact has been confirmed by many experiments (Arzt, 2017) as well.

9. Related work

9.1. Language-based flow security

In the book (Denning, 1982), D. E. Denning first gave the definition of *information flow policy*, and proposed two kinds of flow control mechanisms: *execution-based* (dynamic) and *compiler-based* (static) mechanisms for the traditional sequential programming languages. This work serves as the theoretical foundation for our approach. To reason about the security of information flows, many works investigated the *sound flow-sensitive type systems* (Chaudhuri, 2009; Huang et al., 2015; Myers, 1999; Myers and Liskov, 2000; Nanevski et al., 2013; Sabelfeld and Myers, 2003) to track information flows in programs written by simple languages or Java, in which Chaudhuri (2009) and Huang et al. (2015) were dedicated to the information flows in Android programs.

This paper is largely inspired by Chaudhuri (2009), in which a small language was given and a security type system was proposed. This work brought in the idea of using permissions as security classes, but the security policy was not shaped formally. More importantly, this work did not involve the enforcement of the policy. Xu et al. (2021) developed a permission-dependent type system for secure information flows in Android programs. However, the proposed language failed to cover some primary Android-specific features, such as components and ICC-related commands, which limits its application to apps in reality. Huang et al. (2015) proposed a type-based taint analysis approach for Android programs, but the security policy was still a binary lattice, regardless of permissions yet.

Jif (Java information flow). Jif (Liu et al., 2017; Myers, 1999) uses a lattice model, referred to as *Decentralized Label Model* (*DLM*), to expresses security policies, which can be embedded into Java programs to label information manipulated by programs. The lattice model is a partially ordered set of *labels*; a label is a set of security policies (including both *confidentiality* policies and *integrity* policies), each of which designates an owner and a set of readers (or writers). Labeled information is released (or modified) only by the consensus of all of the confidentiality (or integrity) policies. As a part of type checking, checking security policies is performed mostly at compile time by *Jif compiler* to ensure that programs respect these policies contained in the labels of information.

Jif provides more powerful policies than ours for control of information flows, but its application is confined in pure Java programs; some primary Android-specific features have not been covered yet. The power of Jif manifests in the following two aspects: (1) Jif allows us to express *declassification* (or *downward*) policies for information flows. In some real-world cases, information flows may be required to break down the security property of information flows—allowed to flow downward or to unrelated classes. This requirement can be readily expressed by the confidentiality policies in labels on objects or variables, but not permitted by our policies because ours is unable to offer the flexibility to specify declassification; (2) Jif is able to additionally specify and enforce *integrity* policies for information flows. Our security policy only concentrates on the *confidentiality* property of information flows (i.e., assuring no secret information is disclosed) without considering the *integrity* property (i.e., assuring no information is tampered with). Nonetheless, this emphasis on the confidentiality is reasonable because preventing private information from dissemination is the first-class concern on the security of smartphone applications. However, it is possible to apply DLM to the Android settings, as exhibited by Nadkarni and Enck (2013).

HornDroid. HornDroid (Calzavara et al., 2017; 2016) presents a formal language μ-Dalvik$_A$ for Android programs, which captures the low level operations of Dalvik virtual machine. HornDroid defines the semantics of Android applications as a set of Horn clauses, which can be yielded by the reduction relation for state transitions. The security properties (i.e., the absence of undesired information flows of sensitive data into selected sinks) are formulated as a set of proof obligations, which can be automatically discharged by off-the-shelf SMT solvers (e.g., Z3). In essence, the static analysis accomplished in HornDroid agrees with the IFDS/IDE algorithm underlying FlowDroid—the set of Horn clauses at a reachable configuration just corresponds to the *meet-over-all-valid-path-functions* at a reachable program point; discharging proof obligations for security properties just equates to solving a graph-reachability problem.

Comparable to our work, the most notable strength of HornDroid is the ability to model *exceptions* and *threads* for the semantics of Android programs. Modelling exceptions means HornDroid is able to address the typical statements for exception handling, such as `throw e` and `try{...} catch(e)`. Moreover, HornDroid formalism covers the core methods of the Java Thread API: they enable thread spawning and thread communication by means of interruptions and synchronizations. Although neither of the language features are addressed in our semantics model, finding information flows associated with exceptions and threads is nonetheless involved in our approach, because the both features have been fully achieved in FlowDroid (Arzt, 2017) which is borrowed by our approach to discover the potential information flows within components. Furthermore, one of the contributions claimed by HornDroid, called *strong updates* (Lhoták and Chung, 2011) for static points-to analysis, however, has been already covered in FlowDroid to increase the precision of static analysis.

The main weakness of HornDroid and its enhanced version (Calzavara et al., 2016) is lacking some typical Android-specific features, which limits its expression power for real-world Android applications. For example, μ-Dalvik$_A$ only consists of Activity component; only models intra-application communication based on explicit Intents (implicit Intents are absent). More importantly, the permission mechanism, being indispensable for the semantics of Android programs, was not touched yet, resulting in actual behaviours of Android apps not captured faithfully.

9.2. Taint analysis: Static vs. dynamic

Taint analysis approaches for Android programs roughly fall into two categories: *static* and *dynamic*. FlowDroid (Arzt et al., 2014) is the most cited work in the static category. It integrates various data-flow features (context-, flow-, field-, object-sensitive) and Android-specific features (e.g., lifecycle-aware) into the traditional IFDS/IDE (Reps et al., 1995; Sagiv et al., 1996) algorithm to improve the accuracy of data-flow analysis for Android apps. The main limitation of FlowDroid is the disability to deal with the cross-component information flows; however, this shortcoming can be remedied by integrating the ICC analysis into it, the typical works of which includes Wei et al. (2018), Gordon et al. (2015), Li et al. (2015), and Feng et al. (2014). The key point of ICC-involved taint analysis is to find communicating relations among components, where an effective string solver is mandatory. IC3 (Octeau et al., 2016) and its improved version (Bosu et al., 2017) are of the leading string solvers dedicated to analysing Intent and URI objects in Android programs.

The typical works on dynamic taint analysis include Enck et al. (2014), Sun et al. (2016), Xue et al. (2018), Backes et al. (2017), Xu et al. (2012), Yan and Yin (2012), and Chakraborty et al. (2019). TaintDroid is an extension to the Android platform. It automatically labels data from privacy-sensitive sources and transitively applies labels as sensitive data propagates through program variables, files, and interprocess messages. When tainted data are transmitted over the network, or leave the system, Taintdroid logs the relevant information and issues an alarm. From Android 5.0, the Android system upgrades with ART (Android RunTime) environment, many data-flow analysis systems designed for the legacy Dalvik environment became no longer applicable. TaintART was designed for the ART environment and employed a multi-level taint analysis technique to minimize the taint tag storage. NDroid (Xue et al., 2018) was designed to identify the information leakages across Java context and native context, which existing dynamic analysis systems fail to capture. DroidScope (Yan and Yin, 2012) adopts instrumentation techniques to monitor an app from three tiers: hardware, OS and DVM, and track information leaks through both the Java and native components using taint analysis. Aurasium (Xu et al., 2012) also provides a tool to enable dynamic and fine-grained policy enforcement of Android applications. To intercept relevant events, Aurasium instruments single applications, rather than adopting system-level hooks.

Chakraborty et al. (2019) ensures the security of information flows by achieving *non-interference* of information flows between Android apps. Two information flows are called *non-interference* provided that the attacker is unable to distinguish their outputs given the same confidential input. Therefore, comparing the outputs produced by two information flows that only differ in their confidential input can assist us to detect information leaks. To do this, Chakraborty et al. (2019) introduced a *secure scheduler*, embedded inside Androids framework, to launch and run multiple instances for an application, each of which runs for a different security level, and control the information flow between these instances. The policy lattice of Chakraborty et al. (2019) derived from the *protection levels* of Androids permissions—regarding *dangerous* and *normal* permissions as *high* level, and no permission as *low* level. This lattice, in essence, is still a binary coarse-grained policy, although more levels could be accommodated for different application scenarios. In addition, the policy is to be enforced at runtime; an I/O scheduler initiates multiple application instances to achieve control over data flows and prevent data leaks. As equal to the number of security levels, that of the running instances may be growing large in case of multiple security levels adopted, ultimately bringing in a higher performance overhead.

In summary, all these works, both *static* and *dynamic*, utilize the *two-level* security policy logic to label and track information flows, which is a special case of our permission-carrying flow policy.

9.3. Complex information flows & implicit information flows

Shen et al. (2018) examines the structure, patterns, and relations of information flows, and uses these features to recognize malware and benign applications. However, this work did not explore how to discover flow structures, especially for those over ICC channels. In fact, our work also investigates the structure of information flows, as shown in Section 4.3. Furthermore, we concentrate on chains of flows across components (even apps) to detect vulnerabilities in them. He et al. (2019) introduced a sparse alternative to the traditional IFDS/IDE algorithm. It propagates every data-flow fact directly to its next possible use point along its own sparse control flow graph constructed on the fly, reducing significantly both the time and memory requirements.

DIALDroid (Bosu et al., 2017) reports some findings of large-scale detection of collusive and vulnerable apps, based on inter-app ICC channels among 110,150 real-world apps. It employs FlowDroid as well to detect information flows inside single components, and extracts ICC Entry/Exit points with the aid of IC3-DIALDroid. However, this work lacks a theoretical framework to guide the detection, and fails to fully examine the propagations of permissions along multiple flows to the same program points. Our approach approximates the security class of a program point by merging the permissions propagated from different flows to the point, thus achieving a low rate of false negatives.

Note that, our work does not deal with *implicit* or *covert* channels so far. In fact, FlowDroid has considered implicit information flows ("ImplicitFlowMode" option in Table B.11) along control dependences, such as conditional branches and while-loops. You et al. (2015) thoroughly investigates the implicit information flows in Android programs, but only limited to language features and program control dependences. Additionally, Lanet and Moussaileb (2017) explores privacy leaks through side channel attacks, including timing attack, cache memory attack, meta data attack, and graphic processing unit attack. It provides an in-depth insight into the information leaks via diverse covert channels.

10. Conclusion

In this paper, we propose a permission-carrying secure information flow policy and fulfil a static enforcement mechanism for it. This policy refines the traditional two-level security policy into a multi-level policy, which allows us to label tainted variables and track information flows in a more precise way. Our approach fully exploits permissions in Android programs, so it can be regarded as a close integration of the access control and information flow control. Considering that the third party SDKs have been widely used in actual apps, they play an essential role in the security of apps, so we next intend to carry out a thorough investigation into the vulnerabilities of SDKs by using our policy and enforcement mechanism.

In fact, finding vulnerable information flows is only an initial step towards inspecting and auditing the security of apps—risky information leaks does not truly indicate a malicious attack, we thus should consider multiple features, including information flows, and combine multiple techniques, such as the static/dynamic analysis and machine learning approaches, to give a more precise and credible evaluation for the security of apps.

Declaration of Competing Interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Xiaojian Liu reports financial support was provided by Seclover Corporation and HUAWEI Corporation.

CRediT authorship contribution statement

Xiaojian Liu: Conceptualization, Methodology, Formal analysis, Writing – original draft. **Kehong Liu:** Conceptualization, Formal analysis.

Data availability

Data will be made available on request.

Acknowledgment

This document is the results of the research projects funded by the Science and Technology Plan Project of Shaanxi under Grant 2017JM6105, the Ministry of Education Collaborative Education Project with Seclover Corporation, and the Ministry of Education Collaborative Education Project with HUAWEI Corporation.

Appendix A. Proof

Proof of Theorem 1

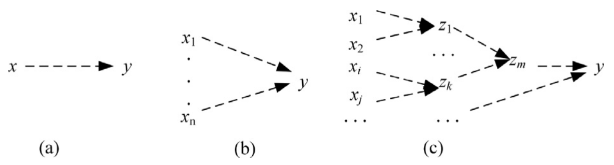


Fig. A1. Three cases of information flows.

Proof. Given a chain of information flows $if = if_1 \hookrightarrow if_2 \hookrightarrow \cdots \hookrightarrow if_n$. Assume each information flow if_i and $if_i \hookrightarrow if_{i+1}$ is authorized, we first prove no threats of confused deputy posed in any flows $if_i \hookrightarrow if_{i+1} (1 \leq i \leq n-1)$.

As mentioned previously, confused deputy only occurs whenever information flows over multiple components; these flows are caused only by executing commands start_{std}, return, send, $u?x$, and $u!v$. In the certifying rules of these commands, the preconditions $\underline{v}_s \sqsubseteq ic'.\mathsf{decl}$ (in [if-start_{std}], [if-return], and [if-send]), $\underline{v}_s \sqsubseteq ic.\mathsf{decl}$ (in [if-query]), and $\underline{v}_s \sqsubseteq ip.\mathsf{decl}$ (in [if-update]), ensure that $\underline{v}_s \sqsubseteq iC_{i+1}.\mathsf{decl}$ holds, i.e., no threats of confused deputy occur in $if_i \hookrightarrow if_{i+1}$ according to Definition 6.

Secondly, we prove no threats of collusive data leaks in if if every if_i and $if_i \hookrightarrow if_{i+1}$ is authorized. Before giving the proof, we can easily observe such a property, called *monotonically increasing information states*, in information flows: for any if_i or $if_i \hookrightarrow if_{i+1}$, say $v_s \rightsquigarrow x_{s'}$, we have $\underline{v}_s \sqsubseteq \underline{x}_{s'}$. This property can be derived from the conclusion parts of the certifying rules. From the rules [if-start_{std}], [if-return], [if-send], [if-query], and [if-update], we conclude $\underline{v}_s \sqsubseteq \underline{x}_{s'}$; for the rule [if-source], the information state of v in state s can be considered as the guarded permissions of the source-API, i.e., $\underline{v}_s = p$, thus we also have $\underline{v}_s \sqsubseteq \underline{x}_{s'}$. In the rule [if-sink], the *outside* world has a constant information state $\perp$, thus $\underline{v}_s \sqsubseteq \perp$ also holds.

Assume if_1 begins with a source-API $\{p\}x := \mathsf{source}(v)$ with $p \neq \emptyset$. According to the above property, we derive that the information state of the exit-point of if_n is higher than p; that is, it is infeasible that if_n ends with a sink-API, because if_n is authorized and $p \not\sqsubseteq \perp$. Therefore, there are no threats of collusive data leaks caused in if according to Definition 7. □

Proof of Theorem 2

Proof. We divide the tainted information flows into three cases, as illustrated in Fig. A1.

1. There is only one tainted information flow $x \rightsquigarrow y$ from x to y (Fig. A.14 (a)). Intuitively, we conclude that, in the propagation from x to y, the security class of x is neither *upgraded* (as no other information sources contribute to y) nor *degraded* (as x cannot be sanitized). In other words, the security class of x keeps invariable in the information flow, i.e., $\underline{y} = \underline{x}$.
2. There are multiple tainted information flows $x_i \rightsquigarrow y$ $(1 \leq i \leq n)$ reaching to y, and these flows are independent, i.e., no variables (except y) are shared between any pair of flows (Fig. A.14 (b)). In this case, all n information flows meet at y, thus all $\underline{x_i}$ should be *merged*, i.e., $\underline{y} = \bigsqcup_{1 \leq i \leq n} \underline{x_i}$ holds.
3. There are multiple tainted information flows $x_i \rightsquigarrow y$ $(1 \leq i \leq n)$ reaching to y, but some variables (except y) may be shared among flows (Fig. A.14 (c)). According to the *distributivity* of the lattice of our policy (Definition 2), the result of this case is the same as that of the second, thus $\underline{y} = \bigsqcup_{1 \leq i \leq n} \underline{x_i}$ also holds.

Summarize all above cases, the conclusion of this theorem is achieved. □

Appendix B. Experiment data

Table B.11
The options and selected items for the configuration of FlowDroid.

Options	Items	Description
ImplicitFlowMode	☐NoImplicitFlows ☐ArrayAccesses ☑AllImplicitFlows	Implicit flows shall be tracked
FlowSensitive-Aliasing	☑True ☐False	A flow-sensitive alias analysis is enabled
EnableCallbacks	☑True ☐False	Taints shall be tracked through callbacks
SequentialPath-Processing	☑True ☐False	Perform sequential path reconstruction
LayoutMatching-Mode	☐NoMatch ☑MatchAll ☐MatchSenOnly	Use all layout components as sources
PathAgnosticResults	☑True ☐False	Results that only differ in their paths is merged
PathBuilding-Algorithm	☐Recursive ☑CtxSensitive ☐CtxInsensitive ☐CxtInsensitive-SourceFinder	Select highly precise context-sensitive path reconstruction approach
StaticFieldTracking-Mode	☑CtxFlowSen ☐CtxFlowInsen ☐None	Track taints on static fields with context- and flow-sensitive approach
EnableReflection	☑True ☐False	Reflective method calls shall be supported

Table B.12
Permissions associated with typical APIs.

API	Guarded permissions
getSubscriberId getDeviceId getLine1Number	READ_PHONE_STATE
sendTextMessage sendDataMessage	SEND_SMS
getLastKnownLocation getCellLocation	ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION
connect getInputStream getContent	INTERNET
getUriForDownloadedFile	WRITE_EXTERNAL_STORAGE
getNetworkInfo getNetworkPreference	ACCESS_NETWORK_STATE
startScan disableNetwork enableNetwork	CHANGE_WIFI_STATE
getContactLookupUri getLookupUri markAsContacted	READ_CONTACTS WRITE_CONTACTS READ_SOCIAL_STREAM WRITE_SOCIAL_STREAM

Table B.13
Permissions associated with security-related Intents.

Intent action	sp / rp (Send / Receive permissions)
CALL	S CALL_PHONE
	R NONE
CALL_EMERGENCY	S CALL_PRIVILEGED
	R NONE
REBOOT	S SHUTDOWN
	R NONE
REQUEST_ENABLE	S BLUETOOTH
	R NONE
CONNECTION_ACCESS _REQUEST	S BLUETOOTH_ADMIN
	R BLUETOOTH_ADMIN
SMS_RECEIVED	S BROADCAST_SMS
	R RECEIVE_SMS
VIEW_NOTIFICATION	S WRITE_CONTACTS
	R NONE
PHONE_STATE	S NONE
	R READ_PHONE_STATE
BOOT_COMPLETED	S NONE
	R RECEIVE_BOOT_COMPLETED

Table B.14
Permissions associated with security-sensitive Providers.

Provider	Permissions
content://browser	R READ_HISTORY_BOOKMARKS
	W WRITE_HISTORY_BOOKMARKS
content://call_log	R READ_CALL_LOG
	W WRITE_CALL_LOG
content://com.android. contacts	R READ_CONTACTS
	W WRITE_CONTACTS
content://com.android. email.provider	R ACCESS_PROVIDER
	W ACCESS_PROVIDER
content://mms-sms	R READ_SMS
	W WRITE_SMS
content://user_dictionary	R READ_USER_DICTIONARY
	W WRITE_USER_DICTIONARY

Table B.15
Patterns of assigning a value to *action* field of an `Intent` object.

`new-Intent`-pattern	`Intent` i = `new` `Intent`(*s*);
`setAction`-pattern	`Intent` i = `new` `Intent`(); i.`setAction`(*s*);
`setComponent`-pattern	`ComponentName` c = `new` `ComponentName`(*s*); `Intent` i = `new` `Intent`().`setComponent`(c);
`setClassName`-pattern	`Intent` i = `new` `Intent().setClassName`(*s*);
`.class`-pattern	`Intent` i = `new` `Intent`(xxx.`class`);
`getClass`-pattern	`Intent` i = `new` `Intent`(xxx.`getClass()`);

Table B.16
Typical manipulations of string arguments.

Pattern	String Manipulation	Test Case
①	*const*	i.`setAction`("android.intent.action.VIEW");
②	*var*	`String` s1="android.intent.action.VIEW"; i.`setAction`(s1);
③	*const* + *const*	i.`setAction`("android.intent.action"+".VIEW");
④	*var* + *const*	`String` s1="android.intent.action"; i.`setAction`(s1+".VIEW");
⑤	*var*.concat(*const*)	`String` s1="android.intent.action."; i.`setAction`(s1.concat("VIEW"));
⑥	*var*.concat(*var*)	`String` s1="android.intent.action"; `String` s2=".VIEW"; i.`setAction`(s1.concat(s2));
⑦	*const*.subString()	i.`setAction`("12android.intent.action".`subString`(2));
⑧	StringBuilder. append(*const*)	`StringBuilder` s=`new` `StringBuilder`(); s.append("android.intent.action.VIEW"); i.`setAction`(s.`toString`());
⑨	StringBuilder. append(*var*)	`String` s="android.intent.action.VIEW"; `StringBuilder` sb=`new` `StringBuilder`(); sb.append(s); `setAction`(sb.`toString`());

Table B.17
Typical *source*-APIs in categories.

Category	*source*-APIs
Location	getLatitude() getLongitude()
Device and user information	getDeviceId() getSubscriberId() getSimSerialNumber() getLine1Number() loadUserByUsername()
Web request	getParameter() getLoginPage() getValue() getDefaultCredentials() getConfig()
Database	createQueryString() getString() getAllBookmarks() getAllVisitedUrls()
File	loadFiles() getCanonicalFile() read()
View	findViewById() getLabelFor()

Table B.18
Typical *sink*-APIs in categories.

Category	*sink*-APIs
Web request	sendRedirect() sendRequestHeader()
Database	setParameters() execute() query() executeQuery(java.lang.String)
File	log() int i() write()
String	append() replace() addSubstring() sendMessage()
I/O	printStackTrace() getOutputStream()
Android	putBinder() sendBroadcast() registerReceiver()

References

360, 2021. 360 brain of security. Available at https://www.360.cn/brain_of_security/.

Aho, A., Sethi, R., Ullman, J., 2007. Compilers: Principles, Techniques, and Tools (Second Edition). Addison-Wesley.

Arzt, S., 2017. Static Data Flow Analysis for Android Applications. Darmstadt, Technische Universität.

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. Acm Sigplan Notices 49 (6), 259–269. doi:10.1145/2594291.2594299.

Backes, M., Bugiel, S., Schranz, O., von Styp-Rekowsky, P., Weisgerber, S., 2017. Artist: The android runtime instrumentation and security toolkit. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, pp. 481–495. doi:10.1109/EuroSP.2017.43.

Baidu, 2021. Baidu anquan. Available at https://www.anquan.baidu.com/product/appprivacy.

Bosu, A., Liu, F., Yao, D., 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, pp. 71–85. doi:10.1145/3052973.3053004.

Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., Shastry, B., 2012. Towards taming privilege-escalation attacks on android. In: NDSS, Vol. 17. Citeseer, p. 19.

Bultan, T., Yu, F., Alkhalaf, M., Aydin, A., 2017. String Analysis for Software Verification and Security, Vol. 10. Springer doi:10.1007/978-3-319-68670-7.

Cai, T., Zhang, Z., Yang, P., 2020. Fastbot: A multi-agent model-based test generation system. In: Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test, pp. 93–96. doi:10.1145/3387903.3389308.

Calzavara, S., Grishchenko, I., Koutsos, A., Maffei, M., 2017. A sound flow-sensitive heap abstraction for the static analysis of android applications. In: 2017 IEEE 30th Computer Security Foundations Symposium (CSF). IEEE, pp. 22–36. doi:10.1109/CSF.2017.19.

Calzavara, S., Grishchenko, I., Maffei, M., 2016. Horndroid: Practical and sound static analysis of android applications by smt solving. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, pp. 47–62. doi:10.1109/EuroSP.2016.16.

Chakraborty, D., Hammer, C., Bugiel, S., 2019. Secure multi-execution in android. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, pp. 1934–1943. doi:10.1145/3297280.3297469.

Chaudhuri, A., 2009. Language-based security on android. In: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, pp. 1–7. doi:10.1145/1667209.1667211.

Denning, D.E.R., 1982. Cryptography and Data Security, Vol. 112. Addison-Wesley Reading doi:10.5555/539308.

Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N., 2014. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS) 32 (2), 1–29. doi:10.1145/2619091.

Feng, Y., Anand, S., Dillig, I., Aiken, A., 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 576–587. doi:10.1145/2635868.2635869.

Google, 2020. Android official document. Avavaible at https://www.developer.android.com/docs.

Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C., 2015. Information-flow analysis of android applications in droidsafe. In: Network and Distributed System Security Symposium. Internet Society, pp. 110–125. doi:10.14722/ndss.2015.23089.

Han, J., Pei, J., Tong, H., 2022. Data Mining: Concepts and Techniques. Morgan kaufmann.

He, D., Li, H., Wang, L., Meng, H., Zheng, H., Liu, J., Hu, S., Li, L., Xue, J., 2019. Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 267–279. doi:10.1109/ASE.2019.00034.

Huang, W., Dong, Y., Milanova, A., Dolby, J., 2015. Scalable and precise taint analysis for android. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis, pp. 106–117. doi:10.1145/2771783.2771803.

Jifeng, H., Li, X., Liu, Z., 2006. Rcos: a refinement calculus of object systems. Theor Comput Sci 365 (1–2), 109–142. doi:10.1016/j.tcs.2006.07.034.

Lanet, J.-L., Moussaileb, R., 2017. Detection of side channel attacks based on data tainting in android systems. In: ICT Systems Security and Privacy Protection: 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29–31, 2017, Proceedings, Vol. 502. Springer, pp. 205–218. doi:10.1007/978-3-319-58469-0_14.

Lhoták, O., Chung, K.-C.A., 2011. Points-to analysis with efficient strong updates. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 3–16. doi:10.1145/1926385.1926389.

Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P., 2015. Iccta: Detecting inter-component privacy leaks in android apps. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1. IEEE, pp. 280–291. doi:10.1109/ICSE.2015.48.

Li, R., Diao, W., Li, Z., Du, J., Guo, S., 2021. Android custom permissions demystified: From privilege escalation to design shortcomings. In: 2021 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 70–86. doi:10.1109/SP40001.2021.00070.

Liu, J., Arden, O., George, M.D., Myers, A.C., 2017. Fabric: building open distributed systems securely by construction. J. Comput. Secur. 25 (4–5), 367–426. doi:10.3233/JCS-15805.

Luo, L., Pauck, F., Piskachev, G., Benz, M., Pashchenko, I., Mory, M., Bodden, E., Hermann, B., Massacci, F., 2022. Taintbench: automatic real-world malware benchmarking of android taint analyses. Empirical Software Engineering 27 (1), 1–41. doi:10.1007/s10664-021-10013-5.

Marforio, C., Ritzdorf, H., Francillon, A., Capkun, S., 2012. Analysis of the communication between colluding applications on modern smartphones. In: Proceedings of the 28th Annual Computer Security Applications Conference, pp. 51–60. doi:10.1145/2420950.2420958.

Myers, A.C., 1999. Jflow: Practical mostly-static information flow control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 228–241. doi:10.1145/292540.292561.

Myers, A.C., Liskov, B., 2000. Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology (TOSEM) 9 (4), 410–442. doi:10.1145/363516.363526.

Nadkarni, A., Enck, W., 2013. Preventing accidental data disclosure in modern operating systems. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 1029–1042. doi:10.1145/2508859.2516677.

Nanevski, A., Banerjee, A., Garg, D., 2013. Dependent type theory for verification of information flow and access control policies. ACM Transactions on Programming Languages and Systems (TOPLAS) 35 (2), 1–41. doi:10.1145/2491522.2491523.

Nielson, F., Nielson, H.R., Hankin, C., 2004. Principles of Program Analysis. Springer Science & Business Media doi:10.1007/978-3-662-03811-6.

Octeau, D., Luchaup, D., Jha, S., McDaniel, P., 2016. Composite constant propagation and its application to android program analysis. IEEE Trans. Software Eng. 42 (11), 999–1014. doi:10.1109/TSE.2016.2550446.

OPPO&Deloitte, 2021. A whitepaper for mobile app personal information protection in chinese app market. Avavaible at https://www2.deloitte.com/content/dam/Deloitte/cn/Documents/risk/deloitte-cn-ra-mobile-app-personal-information-protection-white-paper-211028.pdf.

Pan, X., Cao, Y., Du, X., He, B., Fang, G., Shao, R., Chen, Y., 2018. Flowcog: Context-aware semantics extraction and analysis of information flow leaks in android apps. In: 27th USENIX Security Symposium (USENIX Security 18), pp. 1669–1685.

Pierik, C., Boer, F.S.d., 2003. A syntax-directed hoare logic for object-oriented programming concepts. In: International Conference on Formal Methods for Open Object-Based Distributed Systems. Springer, pp. 64–78. doi:10.1007/978-3-540-39958-2_5.

Rasthofer, S., Arzt, S., Bodden, E., 2014. A machine-learning approach for classifying and categorizing android sources and sinks. In: NDSS, Vol. 14, pp. 1125–1225. doi:10.14722/ndss.2014.23039.

Reps, T., Horwitz, S., Sagiv, M., 1995. Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 49–61. doi:10.1145/199448.199462.

Sabelfeld, A., Myers, A.C., 2003. Language-based information-flow security. IEEE J. Sel. Areas Commun. 21 (1), 5–19. doi:10.1109/JSAC.2002.806121.

Sagiv, M., Reps, T., Horwitz, S., 1996. Precise interprocedural dataflow analysis with applications to constant propagation. Theor Comput Sci 167 (1–2), 131–170. doi:10.1016/0304-3975(96)00072-2.

Shen, F., Del Vecchio, J., Mohaisen, A., Ko, S.Y., Ziarek, L., 2018. Android malware detection using complex-flows. IEEE Trans. Mob. Comput. 18 (6), 1231–1245. doi:10.1109/TMC.2018.2861405.

Smalley, S., Craig, R., 2013. Security enhanced (SE) android: bringing flexible mac to android. In: Ndss, Vol. 310, pp. 20–38. 10.1.1.391.2279

Sun, M., Wei, T., Lui, J.C., 2016. Taintart: A practical multi-level information-flow tracking system for android runtime. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 331–342. doi:10.1145/2976749.2978343.

Tam, K., Feizollah, A., Anuar, N.B., Salleh, R., Cavallaro, L., 2017. The evolution of android malware and android analysis techniques. ACM Computing Surveys (CSUR) 49 (4), 1–41. doi:10.1145/3017427.

Tripp, O., Pistoia, M., Cousot, P., Cousot, R., Guarnieri, S., 2013. Andromeda: Accurate and scalable security analysis of web applications. In: International Conference on Fundamental Approaches to Software Engineering. Springer, pp. 210–225. doi:10.1007/978-3-642-37057-1_15.

Volpano, D., Irvine, C., Smith, G., 1996. A sound type system for secure flow analysis. J. Comput. Secur. 4 (2–3), 167–187. doi:10.3233/JCS-1996-42-304.

Wang, H., Xie, X., Li, Y., Wen, C., Li, Y., Liu, Y., Qin, S., Chen, H., Sui, Y., 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE). IEEE, pp. 999–1010. doi:10.1145/3377811.3380386.

Wei, F., Roy, S., Ou, X., 2018. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. ACM Transactions on Privacy and Security (TOPS) 21 (3), 1–32. doi:10.1145/3183575.

Xposed, 2021. Xposed framework. Avavaible at https://www.repo.xposed.info/.

Xu, R., Saïdi, H., Anderson, R., 2012. Aurasium: Practical policy enforcement for android applications. In: 21st USENIX Security Symposium (USENIX Security 12), pp. 539–552. doi:10.5555/2362793.2362820.

Xu, Z., Chen, H., Tiu, A., Liu, Y., Sareen, K., 2021. A permission-dependent type system for secure information flow analysis. J. Comput. Secur. 29 (2), 161–228. doi:10.3233/JCS-200036.

Xue, L., Qian, C., Zhou, H., Luo, X., Zhou, Y., Shao, Y., Chan, A.T., 2018. Ndroid: toward tracking information flows across multiple android contexts. IEEE Trans. Inf. Forensics Secur. 14 (3), 814–828. doi:10.1109/TIFS.2018.2866347.

Yan, L.K., Yin, H., 2012. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In: 21st USENIX Security Symposium (USENIX Security 12), pp. 569–584. doi:10.5555/2362793.2362822.

You, W., Liang, B., Li, J., Shi, W., Zhang, X., 2015. Android implicit information flow demystified. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, pp. 585–590. doi:10.1145/2714576.2714604.

Xiaojian Liu received the B.S. and M.S. degrees in Mathematics from Nanjing University, Nanjing, China, in 1990 and 1997; and Ph.D degree from Xidian University, Xi'an, China, in 2004. From 2004 to 2006, he was a Post Doctoral Fellow with the Department of Computer Science, Northwestern Polytechnic University. In 2005, he was a visiting Fellow at UNU-IIST (The United Nations University International Institute for Software Technology) in Macau. He is currently an Associate Professor in the School of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an. His research focuses on software security and program analysis.

Kehong Liu was born in Xi'an, Shannxi Province, China, in 1999. Now, she is an undergraduate student in Xi'an University of Science and Technology, and will obtain Bachelor degree in 2023. During the undergraduate studies, she has been engaged in the research project of Android malware detection, and published several journal and conference papers on this topic.