Computing (2022) 104:1839–1865
https://doi.org/10.1007/s00607-022-01069-2

REGULAR PAPER

Formal model for inter-component communication and its security in android

Mohamed A. El-Zawawy[1] · Parvez Faruki[2] · Mauro Conti[3]

Received: 20 May 2021 / Accepted: 22 February 2022 / Published online: 27 March 2022

Abstract

The Android application framework has a pervasive presence. In early 2021, Android has over 70% share of the operating system mobile market (according to *Global-Stats*). Components are the main building blocks of Android Applications. These blocks communicate via a rich Inter-Component Communication (ICC) model rather than the traditional inter-process communication model. `Intents`, `Intent`-filters, and their `Intents` resolution (matching) algorithm are main elements of the ICC. However, the `Intent` resolution algorithm is not robust enough and has flaws that can lead to security breaches. In this paper, we present DLAIR, as an enrichment of the `Intent` resolution algorithm to overcome its security issues. To this end, we start by presenting a formal model to express and validate the ICC semantics. This includes defining key properties guaranteeing consistent and realistic semantic states. We then demonstrate how the semantics can be used to formally validate ICC aspects and to express and check ICC system updates. We verified our proposed model and all its lemmas and theorems in the Coq Proof Assistant, a machine-assisted verification tool. We extend our semantics to develop DLAIR which is assisted by a heuristic, and lightweight tool, LekInt. This tool identifies suspicious execution paths responsible for `intent` based sensitive user-information leakage. On a dataset of 2000 real-world apps, we evaluated LekInt against Flowdroid, a state-of-the-art information leakage analysis tool. Experiments show that LekInt is more effective and efficient than Flowdroid which has a higher false-negative rate and lower false-positive rate than LekInt.

✉ Mohamed A. El-Zawawy
maelzawawy@cu.edu.eg

Parvez Faruki
parvez.faruki@avptirajkot.gujgov.edu.in

Mauro Conti
conti@math.unipd.it

[1] Department of Mathematics, Faculty of Science, Cairo University, Giza 12613, Egypt

[2] Department of Information Technology, AVPTI Rajkot, Rajkot, India

[3] Department of Mathematics, University of Padua, Padua, Italy

Considering the dynamic context in which LekInt is designed to work, the advantage of efficiency overcomes the disadvantage of higher false-negative.

Keywords Android security · ICC · Intents · Intent filters · Semantics · Formal verification · Coq proof assistant

Mathematics Subject Classification 68N30 · 68M25

1 Introduction

The legacy Cell-phone and feature phone were primarily used for messaging, phone calls, short-text, and multimedia communication. However, the advent of smartphone operating systems, the Apple iOS, and Google Android, has ushered a paradigm shift in computing. The current generation smartphone is sophisticated hand-held computers capable of running third-party developer applications (apps). Users prefer Android OS-supported smartphones for social networking, banking, email, and shopping with apps taking an increasing role in our daily life [8, 17, 21]. Android framework components such as *Activities, Broadcast receivers, Content providers*, and *Services* are the basic building blocks of any user app. Android can be realized as an asynchronous framework whose components are independent with multiple entry points [12, 15].

Android framework has a rich Inter-component Communication (ICC) model. It enables the development and deployment of highly usable apps with a facility to extend services offered by other apps. Hence, ICC facilitates the communication between components of apps. `Intent` is a complicated messaging technique for implementing ICC [13]. Although `Intent` is equipped with a security mechanism to prevent attackers from directly accessing data and services of third-party apps, it does have critical security issues [15]. Each app has a file (manifest file) that includes definitions of `Intent filters`. These filters determine `Intents` that can be treated by the app components [13].

`Intent` resolution [13] is an Android framework mechanism for matching a component via its `Intent` filter with the corresponding `Intent` specified in source-code of another component. The `Intent` filters are defined in the `Anroidmanifest.xml`, a binary file for app developers used for explicit declarations. Currently, Android does the matching process even if `Intent` is leaking sensitive data. While an Android app is executed in Sandbox, an app `Intent` is resolved dynamically. Therefore, it is not possible to afford the delay caused by state-of-the-art techniques, dynamically testing the `Intent` leakage. Further, Android ICC framework is limited due to non-availability of formal `Intent` representation model that can match the manifest defined `Intent` filters with corresponding `Intents` at run-time. Such a model is invaluable for theoretical representation and potential framework improvements that can be used for proofs of desirable properties of system updates. More specifically, such formal representation would provide a robust basis for a modified `Intent` resolution algorithm that considers sensitive data leakage.

In this paper, we investigate `Intent` vulnerabilities responsible for data leakage and propose a formal model which defines Android ICC semantics. The semantics

captures Intents, Intent filters, Intent-related APIs, and finally the Intent-resolution algorithm matching Intents and Intent filters. We define a set of key properties that guarantee a consistent and realistic Android state in our proposed semantics. We label the states satisfying such properties as "well-defined". Further, we prove that the transitions of the proposed semantic preserve the "well-defined" propriety of states. Then, we prove the effectiveness of the semantics in two ways: i) We express different aspects of ICC such as changing Intent types via its attributes; ii) We express and check a specific potential update for improving ICC security. The update includes inserting a caching element to semantic states so that processes that led to security breaches can be reported and avoided later on. We expressed and the model and verified all its lemmas and theorems with *Coq proof assistant*, a formal modeling tool [10]. We present the proof of one lemma in the paper and make proofs of all other lemmas and theorems available at our GitHub repository.[1]

Based on our ICC formalization, we propose DLAIR, a security-aware algorithm for Intent resolution. To assist DLAIR, we propose a lightweight and heuristic tool ($\mathcal{L}ek\mathcal{I}nt$) that identifies suspicious execution paths. These are the paths that use Intents in a way that is suspicious of leaking sensitive data. On a set of 2, 000 real-world apps, we evaluated $\mathcal{L}ek\mathcal{I}nt$ and compared it against *Flowdroid* [3], the state-of-the-art information leakage detection tool. The evaluation proves that $\mathcal{L}ek\mathcal{I}nt$ is more effective and efficient than *Flowdroid*. $\mathcal{L}ek\mathcal{I}nt$ and *Flowdroid* reported issues in 824 and 425 applications, respectively. The average running times per application for $\mathcal{L}ek\mathcal{I}nt$ and *Flowdroid* are 47.8 and 74.6 seconds, respectively. Therefore $\mathcal{L}ek\mathcal{I}nt$ has a better time complexity than *Flowdroid*. Efficiency is an important factor as $\mathcal{L}ek\mathcal{I}nt$ is to run dynamically as part of Android. On a set of 100 apps, *Flowdroid* has higher false-negative rate (21 against 8) and lower false-positive rate (4 against 13) than $\mathcal{L}ek\mathcal{I}nt$. Bearing in mind the dynamic environment in which $\mathcal{L}ek\mathcal{I}nt$ is designed to work, the advantages of efficiency and high true positives rate overcome the disadvantage of high false-negative rate (due to the approximation adopted in $\mathcal{L}ek\mathcal{I}nt$ to boost its lightweight feature). Some of the issues reported by $\mathcal{L}ek\mathcal{I}nt$, which were not identified by the *Flowdroid*, are referred to in this paper. The metadata of the dataset and results of the experimental evaluation are available in our GitHub repository.[2]

Contributions This paper has the following contributions:

1. We propose a formal model to theoretically represent the Android framework for Inter-Component Communication (ICC).
2. We formally represent general and security-related ICC properties in our model.
3. We implement and verify our formal model and all the lemmas and theorems of the paper in the well-known proof-assistant Coq [10].
4. We propose DLAIR, a novel security-aware Intent resolution framework.
5. Based on the security guarantee, we propose $\mathcal{L}ek\mathcal{I}nt$, a lightweight and heuristic tool that detects Intent generated sensitive data leakage.
6. We compare $\mathcal{L}ek\mathcal{I}nt$ against *Flowdroid* as one of the state-of-the-art techniques.

[1] https://github.com/maelzawawy/LekInt.

[2] https://github.com/maelzawawy/LekInt.

Paper Outline The paper is organized as follows. The most relevant state-of-the-art is critically evaluated and discussed in Sect. 2. Necessary background is presented in Sect. 3. The threat model that motivates the work in this paper is presented in Sect. 4. Section 5 presents our formal model for ICC framework. This model comprises the semantic states and key properties defining *well-defined* state. The main APIs of ICC have been discussed in this section as well. In Sect. 6, we present semantic rules for the framework APIs and prove that, semantic preserves the "well-defined" predicate of states. Section 7 utilizes the semantics presented in the paper to express basic and advanced ICC attributes. Section 8 presents DLAIR including the lightweight tool $\mathcal{L}$ek$\mathcal{I}$nt and its design, implementation, and evaluation.

2 Related work

Many studies have been published on intents and component communication using intents. This section reviews recent studies focused on these issues and also on Android formalization [5, 28]. Potential future research is also discussed in this section.

Permissions are located in application manifest files. Permissions can be realized as requests to acquire access to delicate resources. Betarte et al. [5, 6] presented a model for formalizing permission system of Android 6. They also formalized and proved some properties and security mechanisms for the permission model of Android using Coq. While our model targets the interesting behavior of intents and intent filters simply, the work in [5, 6] is focused on the permission system in a complicated way. Chin et al. presented a tool, *ComDroid* [9], to check application interaction in Android. The tool is based on code analysis and detects security vulnerabilities related to data leakage in Android components. However, unlike our work, *ComDroid* is not supported by formal semantics to application interaction or its analysis steps.

Xu et al. presented a malware detection method, *ICCDetector* [29] which needs training with sets of benign applications and malware. This method focuses on the interactions among components of the same application. However, most other similar methods focus on resources claimed by malware such as permissions, system calls, and API calls. The work in our paper can be realized as a formal way to represent this interaction between components. Feizollah et al. [16] studied the effectiveness of including intent information in a characterizing feature for detecting malicious applications. They claimed that using intents reveals more precise malware behaviors than using only permission features. This emphasizes the need for a formal model to intents such as the one presented in our paper.

Building on the Android framework, Schmerl et al. presented a technique, *Raindroid* [25], that boosts security and preserves extendability. *Raindroid* relies on static analysis methods to recognize interaction among applications. The technique relies also on run-time methods that observe these interactions. Then the methods advise the system to reject interactions, require user permissions, or grant them. However, formal reasoning about the correctness of the advice is not presented in the paper. The model presented in our paper can be augmented with specific APIs to provide formal reasoning. Barros et al. presented static-analysis methods for Java reflection and intents in [4]. These methods are based on implicit control-flow. The methods also

reveal control-flow points and data passed at these points. While our work is based on axiomatic semantics to precisely capture the `Intent` concept, the work in [4] uses type systems to treat intents.

Android has a mechanism by which an application can have private components. Such components are not accessible by other applications. Using next-intent vulnerability (NIV), this privacy can be bypassed by malicious applications. Hence private components become invocable by other applications. In [26], Tang et al. proposed an intent-flow static analysis *NIVAnalyzer* to investigate smali code and reveal NIV. The static detection is then followed by a dynamic verification via crafted exploit applications. However, the method is not supported with a formal model proving the correctness of the analysis. Some of the APIs used in *NIVAnalyzer* are given semantics in our model. Also remaining APIs (such as *getIntent()*) used in *NIVAnalyzer* can be added to our model easily. This can lead to formal reasoning about NIV.

3 Background

Android Concepts The building blocks for the Android app development framework are activities (UI tasks); broadcast receiver (system-wide broadcasts); content provider (data handling); and service (background services). A service is a group of background processes that do not have a user interface since they execute in the background. Broadcast receivers are background event observers that are commenced by other components. Content providers enable reaching and treating memory contents such as files, folders, and databases [11, 20].

`Intent`[3] is a messaging object used for communication between the app components. `Intents` are either explicit or implicit. An explicit `Intent` determines the target component which is usually in the same app. Implicit `Intents` do not specify a certain component. Instead, they specify the needed action. This makes it possible for a component from another app to do the action. Listing 1 illustrates an example of an explicit `Intent` usage, `ImpViewIntent` also manifests implicit intent example. `ImpViewIntent` exhibits the action for viewing text. `ImpViewIntent` requests that another adequate app shows the text (announcement).

Listing 1 Examples of Explicit and Implicit Intents.

```
Intent ExpServiceIntent= new Intent(this, MyService.class);
startService(ExpServiceIntent);

Intent ImpViewIntent= new Intent();
ImpViewIntent.setAction(Intent.ACTION_VIEW);
ImpViewIntent.putExtra(Intent.EXTRA_TEXT, announcement);
ImpViewIntent.setType("text/plain");
if (ImpViewIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(ImpViewIntent);
}
```

An `Intent` filter is a specification in manifest file. The filter determines `Intents` that a component can handle. Declaring an `Intent` filter for an activity enables other

[3] https://developer.android.com/reference/android/content/Intent.

apps to invoke the activity using a convenient `Intent`. Listing 2 demonstrates an example of such `Intent` filter. The filter declares that its component receives an `ACTION_VIEW Intent` when `text` is the value of data type [7].

Listing 2 `Intent` Filter Example.

```
<activity android:name="ViewActivity">
    <intent-filter>
        <category android:name ="android.intent.category. DEFAULT"/>
        <action android:name ="android.intent.action.VIEW"/>
        <data android:mimeType ="text/plain"/>
    </intent-filter>
</activity>
```

`Intent` Resolution `Intent` resolution [13] is the process of assigning a convenient component to a given implicit `Intent`. The assigned component must contain an `Intent` filter whose attributes match corresponding ones in the `Intent`. These attributes are action, category, and data. Therefore, `Intent` resolution includes three tests: action, category, and data tests. The tests are formalized n Table 2.

4 Threat model

This section presents the threat model which motivates us to formalize the intent resolution mechanism. The threat model visualized in Fig. 1 depicts main entities of the Android system; *App1*, *App2*, and *Intent Resolution Algorithm (IRA)*. *App1* has a component *Comp1*, that defines an implicit `Intent`, I, to start an activity. We assumes that I stores sensitive data in its *Extra* field. This is fair assumption as it common for `Intents` to store sensitive data [1]. Then I is passed to the API *startActivity* that asks Android to start an activity with characteristics determined in I. The Android system passes I to IRA to find a matching component. *App2* has a component, *Comp2*. The manifest file of *App2* has an Intent filter, F, corresponding to *Comp2*. *IRA* uses F to check if *Comp2* is a good match for I. If I and F match each other, I (with the wrapped sensitive data) is passed to *App2* that initiates an instance of *Comp2*. The flow of I from *App1* to *App2* is an example of leaking sensitive data in the ICC framework.

The example above shows that there is a need for a lightweight and heuristic algorithm that identifies `Intent`-related sensitive data leakage. This need motivated our paper. The lightweight nature of the algorithm is justified by the fact that it is to be used dynamically while the app is running. Therefore the algorithm should not affect the performance significantly. The heuristic aspect of the algorithm is due to the fact that it should recognize cases where there is no need for security checks. Examples of these cases are when the checked `Intent` and the `Intent` filter are protected by permissions. In cases like these, the sensitive data passing is intended for some logical reasons related to the functions of the involved apps. The other motivation for this paper is the need to build the algorithm on formal semantics for the process of Inter-Component communication (ICC). It would be an advantage if the semantics is general enough to reason formally about ICC functional and security aspects [29].

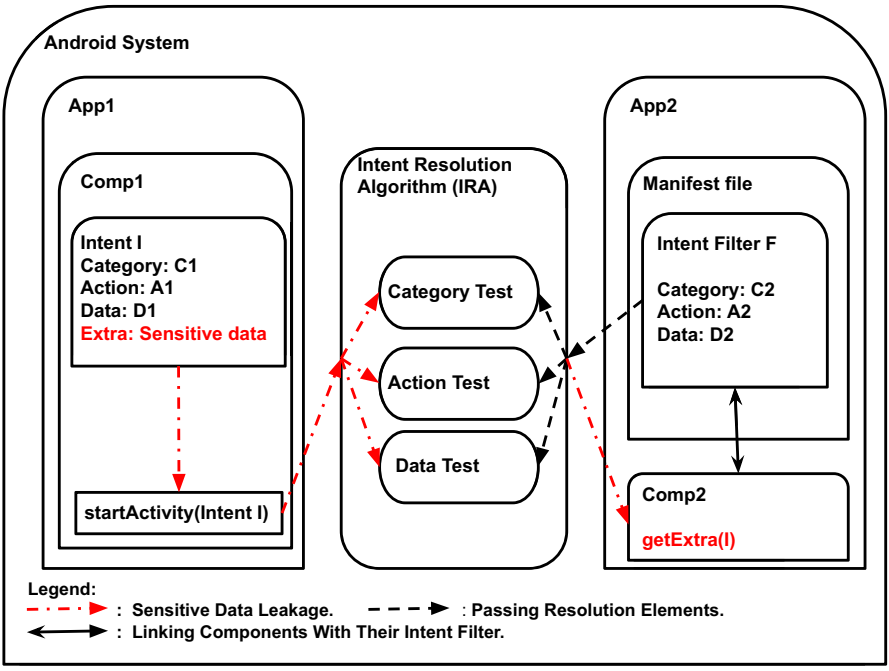

Fig. 1 Threat model

5 Semantic states

In the following, we present the basis of our formal model for Android ICC. This section introduces the states of the model and axiomatic semantics [18, 23] for `Intent` APIs. The semantics are formalized as state-transition maps that preserve a `well-defined` feature of states. The theoretical model is verified with the proof assistant Coq [10].

An Ù application, as described in Fig. 2, is a pair of an application id, `AppID`, and a manifest, `Manifest`. The `Manifest` component is a formalization of the Android manifest file. The `Manifest` is formalized as a pair: `Comp`* representing the application components and `Compf` representing `Intent` filters together with their hosting components. A component (`Comp`) is an element of the set {`Activity`, `Service`, `BroadReceiver`} which are uniquely specified using identifiers [11]. Moreover, the `Service` has a Boolean (`ClientServer`) that specifies whether the service is a client or server one. The sets `AId, SId, BId` denote sets of activity IDs, service IDs, and broadcast receiver IDs, respectively. The union of these sets forms the set of component IDs, `CmpId`. Since the Components are implemented as classes, the Android framework creates instances of the component classes at run-time. In the following, the instances will be referred to as component instances. As long as the instance exists, we call it an active or a running instance. We describe a component as *installed (downloaded)* means that the component and its app are included in the system under study [11].

$$
\begin{aligned}
\texttt{ef} \in \texttt{ExtraField} &::= \texttt{extra-email} \mid \texttt{extra-subject} \mid \texttt{extra-text} \mid \texttt{extra-cc}.\\
\texttt{act} \in \texttt{Action} &::= \texttt{action-main} \mid \texttt{action-view} \mid \texttt{action-send} \mid \texttt{action-edit}.\\
\texttt{cat} \in \texttt{Category} &::= \texttt{catgeory-launcher} \mid \texttt{catgeory-default} \mid\\
&\quad\ \texttt{catgeory-alternative} \mid \texttt{catgeory-browsable}.\\
\texttt{MimeType} &::= \texttt{text} \mid \texttt{image} \mid \texttt{video}.\\
\texttt{cmpId} \in \texttt{CmpId} &::= \texttt{AId} \mid \texttt{SId} \mid \texttt{BId}\\
\texttt{ClientServer} &::= \texttt{True} \mid \texttt{False}\\
\texttt{Activity} &::= \texttt{AId}\\
\texttt{Service} &::= \texttt{SId} \times \texttt{ClientServer}\\
\texttt{BroadReceiver} &::= \texttt{BId}\\
c \in \texttt{Comp} &::= \texttt{Activity} \mid \texttt{Service} \mid \texttt{BroadReceiver}.\\
\texttt{dat} \in \texttt{Data} &::= \texttt{MimeType} \times \texttt{Scheme} \times \texttt{Host} \times \texttt{Port} \times \texttt{Path}.\\
\texttt{Extra} &::= \texttt{ExtraField} \times \texttt{ExtraValue}\\
i \in \texttt{Intent} &::= \texttt{IntId} \times \texttt{CmpId} \times \texttt{Action} \times \texttt{Data} \times (\texttt{Category})^* \times \texttt{Extra}.\\
f \in \texttt{IntentFilter} &::= \texttt{IntFilId} \times (\texttt{Action})^* \times (\texttt{Category})^* \times (\texttt{Data})^*.\\
\texttt{Compf} &::= (\texttt{Comp} \times \texttt{IntentFilter}) \longrightarrow \{\texttt{True}, \texttt{False}\}.\\
\texttt{Manifest} &::= \texttt{Comp}^* \times \texttt{Compf}.\\
\texttt{App} &::= \texttt{AppId} \times \texttt{Manifest}.
\end{aligned}
$$

Fig. 2 `Intents`, `Intent`-filters, and Applications Formalization

$$
\begin{aligned}
\texttt{IComp} &::= \texttt{InstId}\\
\texttt{Apps} &::= \texttt{App} \longrightarrow \{\texttt{True}, \texttt{False}\}.\\
\texttt{Active} &::= \texttt{IComp} \times \texttt{Comp} \longrightarrow \{\texttt{True}, \texttt{False}\}.\\
\texttt{IntFilter} &::= \texttt{App} \times \texttt{IntentFilter} \longrightarrow \{\texttt{True}, \texttt{False}\}.\\
\texttt{WaitImpInt} &::= \texttt{App} \times \texttt{Intents} \longrightarrow \{\texttt{True}, \texttt{False}\}.\\
\texttt{IntentResol} &::= \texttt{Intents} \times \texttt{IntentFilter} \longrightarrow \{\texttt{True}, \texttt{False}\}.\\
\texttt{AppLauncher} &::= \texttt{App} \times \texttt{Comp} \longrightarrow \{\texttt{True}, \texttt{False}\}.\\
s \in \texttt{SysState} &::= \texttt{Apps} \times \texttt{Active} \times \texttt{IntFilter} \times \texttt{WaitImpInt} \times \texttt{IntentResol} \times \texttt{AppLauncher}.
\end{aligned}
$$

Fig. 3 Model states formalization

`Intent` and `Intent` filters concepts are also formalized in Fig. 2. An `Intent` is 6-tuple that consists of:

- `IntId`: the `Intent` identifier,
- `CmpId`: the identifier of the component hosting the `Intent`,
- `Action`: the action type of the `Intent`,
- `Data`: the data type of the `Intent`,
- (`Category`)*: the set of categories targeted by the `Intent`, and
- `Extra`: the extra data (if any) needed to execute the `Intent`.

An `Intent` filter is quadruple that consists of:

- `IntFilId`: the `Intent` filter identifier,
- (`Action`)*: the set of action types that can be handled by the `Intent` filter,
- (`Category`)*: the set of categories that can be handled by the `Intent` filter, and
- (`Data`)*: the set of data types that can be handled by the `Intent` filter.

The semantic states (Fig. 3) of the model are 6-tuples with the following components:

```
Definition WellDefined  : Prop :=
     WaitIntsImp  /\ UsrIntFsDefd /\ NoRepIntFs /\ NoRepImpInts /\
     ExstAppforWatInt /\ RunInsInSysComp /\ ResExstIntIntF.
Definition WaitIntsImp : Prop :=
     forall (a:App) (i:Intents),(WaitImpInt s) a i ->Implicit_Intent   i.
Definition UsrIntFsDefd : Prop :=
     forall (a:App)(f:IntentFilters), (IntFilters s) a f ->
     (Apps s) a /\ (exists c:Comp, Compf (Manifest a) c f).
Definition NoRepIntFs  : Prop :=
     forall (a1:App)(a2:App)(c1:Comp)(c2:Comp)(f:IntentFilters),
     (IntFilters s) a1 f -> (IntFilters s) a2 f->
     (Apps s) a1 -> (Apps s) a2 -> Compf (Manifest a1) c1 f->
     Compf (Manifest a2) c2 f ->c1=c2 ->a1=a2.
Definition NoRepImpInts : Prop :=
     forall (a1:App) (a2:App)(i:Intents),
     (WaitImpInt s) a1 i -> (WaitImpInt s) a2 i ->a1=a2.
Definition ExstAppforWatInt : Prop :=
     forall (a:App) (i:Intents), (WaitImpInt s) a i ->(Apps s) a.
Definition RunInsInSysComp : Prop :=
     forall (ic:iComp)(c:Comp), (Active s) ic c -> CompDwnloaded c s.
Definition ResExstIntIntF : Prop :=
     forall (i:Intents)(f:IntentFilters), (IntentResol s) i f ->
     exists (a1:App) (a2:App), (WaitImpInt s) a1 i /\ (IntFilters s)a2 f.
```

Fig. 4 The formalization of the predicate `WellDefined` and its sub-predicates

- o `Apps` : a set of installed applications,
- o `Active`: a set of pairs of active instances of components together and their components.
- o `IntFilter`: a set of pairs of applications with their `Intent` filters.
- o `WaitImpInt`: a set of implicit `Intents` still not linked to a matching `Intent` filter (i.e. needing `Intent` resolution).
- o `IntentResol`: a set of pairs of `Intents` and `Intent` filters that were found to match each other.
- o `AppLauncher`: a set of pairs of applications and their launching components.

The notation `IComp` denotes a component instance. The instance is identified via an ID, `InstId`.

The proposed semantics relies on the notion of well-defined system states. The notion captures system states satisfying realistic Android characteristics. These characteristics ensure states respecting the principles of Android OS. Formally, this notion is expressed as a predicate (`WellDefined`) on system states (`SysState`). Figure 4 shows the formalization of the predicate and its sub-predicates in the proof assistant Coq [10]. All the predicates assume a state variable $s \in$ `SysState`. To precisely simulate semantics [18, 23] of `Intent` APIs, we define some predicates and functions in Coq. These are presented and described in Table 1.

The following sub-predicates compose the `WellDefined` predicate:

1. The predicate `WaitIntsImp` holds for a state s if `WaitImpInt` links applications with implicit `Intents` only.
2. The predicate `UsrIntFsDefd` holds for a state s if `IntFilter` links an application a to an `Intent` filter f only if a is included in s and f is included in one of the components in `Manifest a`. The predicate guaranties that treated `Intent` filters are obtained from installed applications.

Table 1 Auxiliary predicates and functions

IsActivity(c)	Holds if c is an activity.
IsService(c)	Holds if c is a service.
IsBroadCast(c)	Holds if c is a broadcast receiver.
ExpInt(i)	Holds if i is an explicit `Intent`.
ImpInt(i)	Holds if i is an implicit `Intent`.
`Intent_hasA`(i)	Holds if the `Intent` i has a value (rather than `None`) in its `Action` field.
`Intent_hasD`(i)	Holds if the `Intent` i has a value (rather than `None`) in its `Data` field.
`Intent_hasC`(i)	Holds if the `Intent` i has a value (rather than `None`) in its `Category` field.
`Intent_hasE`(i)	Holds if the `Intent` i has a value (rather than `None`) in its `Extra` field.
`Intent_hasEV`(i, e, v)	Holds if the `Intent` i has the value (e, v) in its `Extra` field.
`IntentF_hasA`(f)	Holds if the `Intent` filter f has a value (rather than `None`) in its `Action` field.
`IntentF_hasAa`(f, a)	Holds if the `Intent` filter f includes the action a in its `Action` field.
`IntentF_hasD`(f)	Holds if the `Intent` filter f has a value rather than `None` in its `Data` field.
`IntentF_hasDd`(f, d)	Holds if the `Intent` filter f includes the data d in its `Data` field.
`IntentF_hasC`(f)	Holds if the `Intent` filter f has a value (rather than `None`) in its `Category` field.
`IntentF_hasCc`(i, c)	holds if the `Intent` filter f includes the category c in its `Category` field.
CompInApp(c, a)	Holds if c is a component in the application a.
InFComp(c, f)	Holds if the `Intent` filter f belongs to the component c.
RetCmpId(c)	Returns the identifier `CompId` of the component c.
SerHasclntSerInt(c)	Holds if the component c is a service and the value of its attribute `ClientServer` is `True`.
CompDwnloaded(c, s)	Holds if the component c belongs to an application in the state s.
IntFDwnloaded(f, s)	Holds if the `Intent` filter f belongs to an application in the state s.
UsrDefIntF (f, s)	Holds if the `Intent` filter f is defined in any of the components of the state s.
InsCompNotInState (ic, s)	Holds if the component instance ic does not belong to the state s.

3. The predicate `NoRepIntFs` holds for a state s if an `Intent` filter can not belong to two identical components of different apps.
4. The predicate `NoRepImpInts` holds for a state s if no implicit `Intent` waiting for resolution can belong to two different apps.
5. The predicate `ExstAppforWatInt` holds for a state s if implicit `Intents` not having matching components yet belong to system applications. This predicate guaranties that treated `Intents` are system ones and no foreign `Intents` are leaking data.
6. The predicate `RunInsInSysComp` holds for a state s if every running instance ic of a component c has its corresponding component installed.
7. The predicate `ResExstIntIntF` holds for a state s if the matching between `Intent` i and `Intent` filter f implies two conditions. The first condition is that

```
api ∈ API ::= startActivity(i) | startService(i) | sendBroadcast(i) |
              setComponent(i,option cmpId) | setData(i, option dat) |
              putExtra (i, ef, ev) | addCategory(i, cat) |
              setAction (i, option act).
```

Fig. 5 Intent-related APIs

i is an implicit `Intent` in an installed application, $a1$. The other condition is that f belongs to an installed application, $a2$.

We later prove that the predicate `WellDefined` is invariant concerning transition maps of the semantics [18, 23].

Figure 5 presents APIs that are related to `Intents`. The formal semantics of the APIs is presented in the next section and their description is as follows:

- `startActivity(i)` starts an activity that is determined by the `Intent` i.
- `startService(i)` starts a service that is specified by the `Intent` i.
- `sendBroadcast(i)` sends a broadcast that is fixed by the `Intent` i.
- `setComponent(i, option cmpId)` explicitly sets the component that the `Intent` will target.
- `setAction (i, option act)` sets the attribute `Action` of the given `Intent`.
- `setData(i, option dat)` sets the attribute `Data` of the `Intent` i.
- `addCategory(i, cat)` adds a category to the list attribute `Category` of the `Intent` i.
- `putExtra(i, ef, ev)` sets the attributes `ef` and `ev` of the `Intent` i.

These APIs include ones that start a component via an `Intent`. The APIs also include ones that modify attributes of `Intent` objects. Therefore, the APIs are among the ones most used by Android programmers when using `Intents`. The behavior described above for the APIs is based on the official specification by `Google` [11, 13].

6 Semantic rules

This section presents the semantics of the Android APIs (Fig. 5) most related to ICC. The semantics is captured by determining pre and post conditions. For each `api` $\in$ `API`, a pre-condition expresses the conditions of a system state able to execute `api`. However, its post-condition describes the main characteristics of the system state resulting from executing `api`. The relationship $\rightsquigarrow$ denotes the execution of `api`. Figure 6 introduces the conditions `pre`(s) and `post`(s') such that `api` $: s \rightsquigarrow s'$.

We first provide a higher level of explanation for intent resolution and then present its formalization. We then use the formalization in the semantic rules. An `Intent` filter passes the action test if the `Intent` action is one of the filter actions. However, if the action list of filters is not empty and the `Intent` has no action included, the filter passes the test. To pass the category test, `Intent` categories have to be included in filter categories. The data element (of `Intents` and `Intent` filters) can have

Precondition – startActivity (i: Intent):
$(\exists c : Comp).\ \texttt{Intent\_resolution}(i, c) \wedge \texttt{IsActivity}(c) \wedge \texttt{CompDwnloaded}(c, s).$
Postcondition – startActivity (i: Intent):

1. $\forall(ic' : \texttt{IComp})(c' : \texttt{Comp}),\ (\texttt{Active}\ s)\ ic'\ c' \Rightarrow (\texttt{Active}\ s')\ ic'\ c'.$
2. $\forall(i' : \texttt{Intent})(f' : \texttt{IntentFilter}),\ (\texttt{IntentResol}\ s)\ i'\ f' \Rightarrow (\texttt{IntentResol}\ s')\ i'\ f'.$
3. $\texttt{WaitImpInt}\ s = \texttt{WaitImpInt}\ s' \wedge \texttt{IntFilter}\ s = \texttt{IntFilter}\ s' \wedge$
   $\texttt{AppLauncher}\ s = \texttt{AppLauncher}\ s' \wedge \texttt{Apps}\ s = \texttt{Apps}\ s'.$
4. $\texttt{ExpInt}(i) \Rightarrow \texttt{IntentResol}\ s = (\texttt{IntentResol}\ s').$
5. $\exists(c : \texttt{Comp})(ic : \texttt{IComp}),\ \texttt{Intent\_resolution}(i, c) \wedge \texttt{IsActivity}(c) \wedge$
   $\texttt{CompDwnloaded}(c, s) \wedge \texttt{InsCompNotInState}(ic, s) \wedge (\texttt{Active}\ s')\ ic\ c.$
6. $\forall(ic' : \texttt{IComp})(c' : \texttt{Comp}),\ (\texttt{Active}\ s')\ ic'\ c' \Rightarrow (\texttt{Active}\ s)\ ic'\ c' \vee (ic = ic' \wedge c = c')).$
7. $\texttt{ImpInt}(i) \Rightarrow \exists(a_1\ a_2 : \texttt{App})(f : \texttt{IntentFilter}).\ (\texttt{IntentResol}\ s)\ i\ f$
   $\wedge\ (\texttt{WaitImpInt}\ s)\ a_1\ i\ \wedge (\texttt{IntFilter}\ s)\ a_2\ f \wedge \texttt{InFComp}(c, f) \wedge (\texttt{IntentResol}\ s')\ i\ f$
   $\wedge\ (\forall(i' : \texttt{Intent})(f' : \texttt{IntentFilter}),\ (\texttt{IntentResol}\ s')\ i'\ f' \Rightarrow (\texttt{IntentResol}\ s)\ i'\ f' \vee (i =$
   $i' \wedge f = f'))).$

Precondition – setComponent(i: Intent, cid: option CmpId):

1. $\texttt{ImpInt}(i) \Rightarrow \exists(a : \texttt{App}),\ (\texttt{WaitImpInt}\ s)\ a\ i.$
2. $\exists\ (\texttt{cidi:CmpId}), \texttt{cid} = \texttt{cidi} \Rightarrow \exists(\texttt{c:Comp}),\ \texttt{CompDwnloaded}(c, s) \wedge \texttt{RetCmpId(c)= cidi}.$

Postcondition – setComponent(i: Intent, cid: option CmpId):

1. $\texttt{CmpId}\ i = \texttt{cid} \wedge \texttt{Apps}\ s = \texttt{Apps}\ s' \wedge \texttt{Active}\ s = \texttt{Active}\ s' \wedge \texttt{IntFilter}\ s = \texttt{IntFilter}\ s' \wedge$
   $\texttt{AppLauncher}\ s = \texttt{AppLauncher}\ s'.$
2. $\texttt{cid} \neq \texttt{None} \Rightarrow \forall(a : \texttt{App}),\ \neg(\texttt{WaitImpInt}\ s')\ a\ i.$
3. $\forall(a : \texttt{App})(i' : \texttt{Intent}),\ (\texttt{WaitImpInt}\ s')\ a\ i' = (\texttt{WaitImpInt}\ s)\ a\ i' \vee i = i'$
4. $\forall(f : \texttt{IntentFilter}),\ \neg(\texttt{IntentResol}\ s')\ i\ f.$
5. $\forall(i' : \texttt{Intent})(f' : \texttt{IntentFilter}),\ (\texttt{IntentResol}\ s')\ i'\ f' = (\texttt{IntentResol}\ s)\ i'\ f' \vee i = i'.$
6. $\texttt{cid} = \texttt{None} \Rightarrow \forall(a : \texttt{App}),\ (\texttt{WaitImpInt}\ s')\ a\ i \Rightarrow (\texttt{Apps}\ s')\ a.$
7. $\texttt{cid} = \texttt{None} \Rightarrow \forall(a_1 : \texttt{App})(a_2 : \texttt{App}),\ (\texttt{WaitImpInt}\ s')\ a_1\ i \Rightarrow (\texttt{WaitImpInt}\ s')\ a_2\ i \Rightarrow a_1 = a_2.$

Precondition – setAction (i: Intent, act: option Action):
$\texttt{ImpInt}(i) \Rightarrow \exists(a : \texttt{App}),\ (\texttt{WaitImpInt}\ s)\ a\ i.$
Postcondition – setAction (i: Intent, act: option Action):

1. $\texttt{Action}\ i = \texttt{act} \wedge \texttt{Apps}\ s = \texttt{Apps}\ s' \wedge \texttt{Active}\ s = \texttt{Active}\ s' \wedge \texttt{IntFilter}\ s = \texttt{IntFilter}\ s' \wedge$
   $\texttt{AppLauncher}\ s = \texttt{AppLauncher}\ s' \wedge \texttt{WaitImpInt}\ s = \texttt{WaitImpInt}\ s'.$
2. $\forall(f : \texttt{IntentFilter}),\ \neg(\texttt{IntentResol}\ s')\ i\ f.$
3. $\forall(i' : \texttt{Intent})(f' : \texttt{IntentFilter}),\ (\texttt{IntentResol}\ s')\ i'\ f' = (\texttt{IntentResol}\ s)\ i'\ f' \vee i = i'.$

Fig. 6 Pre and post conditions of the semantics of intent-related APIs

a MIME data type and a URI structure. The URI has four separate parts: scheme, host, port, and path. Although these parts are optional, their existence respects linear dependencies.

The action, category, and data tests of intent resolution are formalized in Table 2. Rule 1 (R.H.S.) states that a component c resolves an intent i either explicitly or implicitly. Rule 1 (L.H.S.) states that the explicit resolution occurs when i is explicit and `CmpId` of c coincides with that specified in i. Rules 2 states that the implicit resolution occurs when c includes an intent filter that matches i. That the matching between an intent and an intent filter includes three tests (data, category, and action) is formalized in Rule 3 (L.H.S.). The category and action tests are formalized in Rules 3 (R.H.S.) and 2, respectively. The data test is formalized in Rules 4 (R.H.S.), 5, 6, 7, 8, and 9 . The remaining rules of the figure formalize matching conditions between `URL` components of data elements contained in an `Intent` and an `Intent` filter.

Examples of applying rules of Table 2 are provided by the implicit `Intent` and the intent filter given in the Listings 1 and 2 of Sect. 3. We show some application cases. Rule 3 (L.H.S.) can be applied as follows: the condition $\texttt{category-default} \in (\texttt{Category})^* f$ applies because the category `DEFAULT` belongs to the intent filter. Also the condition $(\texttt{Category})^* i \subseteq (\texttt{Category})^* f$ applies because the intent

Table 2 Formalization of `Intent` Resolution Rules

$$\frac{\texttt{ExpInt}(i) \quad \texttt{CmpId}\ i = \texttt{RetCmpId}(c)}{\texttt{Intent\_resolution}_e(i,c)} \qquad \frac{\texttt{Intent\_resolution}_e(i,c) \vee \texttt{Intent\_resolution}_i(i,c)}{\texttt{Intent\_resolution(i,c)}} \quad (1)$$

$$\frac{\texttt{ImpInt}(i) \quad \exists f \in \texttt{IntentFilter}.\ \texttt{InFComp}(c,f) \wedge \texttt{IntMatchIntF}(i,f)}{\texttt{Intent\_resolution}_i(i,c)} \quad (2)$$

$$\frac{\texttt{category-default} \in (\texttt{Category})^* f \quad (\texttt{Category})^*\ i \subseteq (\texttt{Category})^*\ f}{\texttt{CategoryTest(i,f)}} \qquad \frac{\texttt{ActionTest}(i,f) \quad \texttt{CategoryTest}(i,f) \quad \texttt{DataTest}(i,f)}{\texttt{IntMatchIntF(i,f)}} \quad (3)$$

$$\frac{\texttt{IntentF\_hasA(f)} \quad \texttt{Action i=None} \vee (\exists a : \texttt{Action}.\ (\texttt{Action i} = a) \wedge a \in (\texttt{Action})^* f)}{\texttt{ActionTest(i,f)}} \quad (4)$$

$$\frac{\exists t \in \{1,2,3,4\}.\ \texttt{UriMatch}_t(d_1,d_2)}{\texttt{UriMatch}(d_1,d_2)} \qquad \frac{\neg(\texttt{Intent\_hasD}(i)) \vee \texttt{IntentF\_hasD}(f)}{\texttt{DataTest}_1(i,f)} \quad (5)$$

$$\frac{\exists d_1,d_2 \in \texttt{Data}.\ (d_2 \in (\texttt{Data})^*\ f) \wedge (\texttt{Data}\ i) = d_1 \wedge \texttt{type}\ d_1 = \texttt{type}\ d_2 = None \wedge \texttt{UriMatch}(d_1,d_2)}{\texttt{DataTest}_2(i,f)} \quad (6)$$

$$\frac{\exists d_1,d_2 \in \texttt{Data}.\ (d_2 \in (\texttt{Data})^*\ f) \wedge (\texttt{Data}\ i) = d_1 \wedge \texttt{scheme}\ d_1 = \texttt{scheme}\ d_2 = None \wedge \texttt{type}\ d_1 \neq None \wedge \texttt{type}\ d_1 = \texttt{type}\ d_2}{\texttt{DataTest}_3(i,f)} \quad (7)$$

$$\frac{\exists d_1,d_2 \in \texttt{Data}.\ (d_2 \in (\texttt{Data})^*\ f) \wedge (\texttt{Data}\ i) = d_1 \wedge \texttt{scheme}\ d_1 \neq None \wedge \texttt{type}\ d_1 \neq None \wedge \texttt{type}\ d_1 = \texttt{type}\ d_2 \wedge (\texttt{scheme}\ d_2 \neq None \Longrightarrow \texttt{UriMatch}(d_1,d_2))}{\texttt{DataTest}_4(i,f)} \quad (8)$$

$$\frac{\texttt{DataTest}_1(i,f) \vee \texttt{DataTest}_2(i,f) \vee \texttt{DataTest}_3(i,f) \vee \texttt{DataTest}_4(i,f)}{\texttt{DataTest}(i,f)} \quad (9)$$

$$\frac{\texttt{scheme}\ d_1 \neq \texttt{None} \wedge \texttt{host}\ d_1 \neq \texttt{None} \wedge \texttt{port}\ d_1 \neq \texttt{None} \wedge \texttt{path}\ d_1 \neq \texttt{None} \wedge \texttt{scheme}\ d_1 = \texttt{scheme}\ d_2 \wedge \texttt{host}\ d_1 = \texttt{host}\ d_2 \wedge \texttt{port}\ d_1 = \texttt{port}\ d_2 \wedge \texttt{path}\ d_1 = \texttt{path}\ d_2}{\texttt{UriMatch}_4(d_1,d_2)} \quad (10)$$

$$\frac{\texttt{scheme}\ d_1 \neq \texttt{None} \wedge \texttt{host}\ d_1 \neq \texttt{None} \wedge \texttt{port}\ d_1 \neq \texttt{None} \wedge \texttt{path}\ d_1 = \texttt{None} \wedge \texttt{scheme}\ d_1 = \texttt{scheme}\ d_2 \wedge \texttt{host}\ d_1 = \texttt{host}\ d_2 \wedge \texttt{port}\ d_1 = \texttt{port}\ d_2}{\texttt{UriMatch}_3(d_1,d_2)} \quad (11)$$

$$\frac{\texttt{scheme}\ d_1 \neq \texttt{None} \wedge \texttt{host}\ d_1 \neq \texttt{None} \wedge \texttt{port}\ d_1 = \texttt{None} \wedge \texttt{path}\ d_1 = \texttt{None} \wedge \texttt{scheme}\ d_1 = \texttt{scheme}\ d_2 \wedge \texttt{host}\ d_1 = \texttt{host}\ d_2}{\texttt{UriMatch}_2(d_1,d_2)} \quad (12)$$

$$\frac{\texttt{scheme}\ d_1 \neq \texttt{None} \wedge \texttt{host}\ d_1 = \texttt{None} \wedge \texttt{port}\ d_1 = \texttt{None} \wedge \texttt{path}\ d_1 = \texttt{None} \wedge \texttt{scheme}\ d_1 = \texttt{scheme}\ d_2}{\texttt{UriMatch}_1(d_1,d_2)} \quad (13)$$

hosts no categories that do not belong to the intent-filter tag. Rule 2 is applicable as the condition

$$\texttt{Action i=None} \vee (\exists a : \texttt{Action}.\ (\texttt{Action i} = a) \wedge a \in (\texttt{Action})^* f)$$

is satisfied through the second part of the "OR" connector. This is so because both the intent and the intent filter have the action `VIEW` which is denoted in the rule by a.

In Fig. 6, the precondition of `startActivity(i)` requires the existence of a component c that resolves the `Intent` i. The component has to be an activity (`IsActivity`(c)) and installed in the state s (`CompDwnloaded`(c,s)). The post-condition of `startActivity(i)` is the union of the seven sub-conditions stated in the figure. The descriptions of the subfunctions are as follows:

1. The component instances that were running in s are still running in s'.
2. All pairs of an `Intent` and an `Intent` filter linked under `IntentResol` s, are also linked under `IntentResol` s'.
3. This sub-condition determines the state parts that are not affected by executing the API. Hence these parts are the same under s and s'.
4. If i is explicit, then no intent resolution is necessary to execute the API. Hence `IntentResol` s = `IntentResol` s'.
5. There is an instance ic of the activity c that is running in s'. The instance was created as a result of executing the API. Therefore, ic does not belong to the state s.
6. Suppose that an ic' is running in s'. Then ic' was running in s or $ic = ic'$.
7. If i is implicit, then the process of intent resolution was carried on. Therefore an `Intent` filter f was found in c such that i matches f. This is recorded in `IntentResol` s'.

The semantics of the APIs `startService(i)` and `sendBroadcast(i)` are similar to that of `startActivity(i)`. However, for `startService(i)` and `sendBroadcast(i)`, one of the changes is to replace `IsActivity`(c) with the convenient function: `IsService c` or `IsBroadCast c`. The descriptions of semantics for `setComponent` and `setAction` given in Fig. 6 are similar to that of `startActivity`. The semantics of APIs `setData(i, dat)`, `addCategory(i, cat)` and `putExtra(i, ef, ev)` are similar to that of the API `setAction (i, act)`.

In the following, we prove that the predicate `WellDefined` is preserved by one-step execution of any of the APIs of Fig. 5. This means that executing an API starting from a `WellDefined` state results in a `WellDefined` state. The proofs of all following lemmas and theorems were verified in Coq and their verification is available online.[4] Hence, we only show the proof of Lemma 1.

Lemma 1 *1.* $(\mathit{WaitIntsImp}(s) \wedge \mathit{API} : s \rightsquigarrow s') \Rightarrow \mathit{WaitIntsImp}(s')$.
2. $(\mathit{UsrIntFsDefd}(s) \wedge \mathit{API} : s \rightsquigarrow s') \Rightarrow \mathit{UsrIntFsDefd}(s')$.
3. $(\mathit{NoRepIntFs}(s) \wedge \mathit{API} : s \rightsquigarrow s') \Rightarrow \mathit{NoRepIntFs}(s')$.

Proof 1. The proof is by induction on the set `API`. We show only the case where `API` = `setComponent`(i, `cid`). By assumption, for the state s we have:

$$\forall(a : \texttt{App})(i : \texttt{Intent}), (\texttt{WaitImpInt}\ s)\ a\ i\ \Rightarrow \texttt{ImpInt}(i). \qquad (14)$$

[4] https://github.com/maelzawawy/LekInt.

We need to show the same for the state s'. Therefore we assume

$$a : \texttt{App},\ i' : \texttt{Intent}, \texttt{and}\ \ (\texttt{WaitImpInt}\ s')\ a\ i'. \tag{15}$$

Now it is enough to show that $\texttt{ImpInt}(i')$. For i, i', we have $i = i'$ or $i \neq i'$. We prove the required in both cases:

- **Case** $i \neq i'$: by the postcondition of `API` (sub-condition 3), we conclude

$$(\texttt{WaitImpInt}\ s')\ a\ i' = (\texttt{WaitImpInt}\ s)\ a\ i'.$$

  By 15, we have $(\texttt{WaitImpInt}\ s)\ a\ i'$. Now by 14, we conclude $\texttt{ImpInt}(i)'$, as required.
- **Case** $i = i'$: we have two subcases:
  - **Case** $\texttt{cid} \neq \texttt{None}$: by the postcondition of `API` (sub-condition 2), we conclude

$$\neg(\texttt{WaitImpInt}\ s')\ a\ i.$$

    This is a contradiction to our assumption in 15. Hence the required is proved.
  - **Case** $\texttt{cid} = \texttt{None}$: by the postcondition of `API` (sub-condition 1), we conclude

$$\texttt{CmpId}\ i = \texttt{None}.$$

    By definition of implicit intent, this implies $\texttt{ImpInt}(i)$, which implies $\texttt{ImpInt}(i')$.

2. The proof is by induction on the set `API`. We show only the case where $\texttt{API} = \texttt{startService}(i)$. By assumption, for the state s we have:

$$\begin{aligned}&\forall(a : \texttt{App})(f : \texttt{IntentFilter}), (\texttt{IntFilter}\ s)\ a\ f\ \Rightarrow\\ &(\texttt{Apps}\ s)\ a\ \wedge (\exists c : \texttt{Comp},\ (\texttt{Compf}\ (\texttt{Manifest}\ a))\ c\ f).\end{aligned} \tag{16}$$

We need to prove the same for the state s'. Hence we assume

$$a : \texttt{App},\ f : \texttt{IntentFilter},\ (\texttt{IntFilter}\ s')\ a\ f. \tag{17}$$

Now it is enough to show that

$$(\texttt{Apps}\ s')\ a \wedge (\exists c : \texttt{Comp},\ (\texttt{Compf}\ \ (\texttt{Manifest}\ \texttt{a}))\ \ \texttt{c}\ \ \texttt{f}).$$

By 17 and the postcondition of `API` (sub-condition 3), we conclude $(\texttt{IntFilters}\ s)\ a\ f$. Hence by 16 we conclude:

$$(\texttt{Apps}\ s)\ a\ \wedge (\exists c : \texttt{Comp},\ (\texttt{Compf}\ (\texttt{Manifest}\ a))\ c\ f).$$

By applying the postcondition of `API` (sub-condition 3) again, we get the required.

3. The proof is by induction on the set API. We show only the case where API = putExtra(i,ef,ev). By assumption, for the state s we have:

$$\forall(a_1, a_2 : \texttt{App})\ (c_1, c_2 : \texttt{Comp})\ (f : \texttt{IntentFilter}),\ (\texttt{IntFilter}\ s)\ a_1\ f \Rightarrow (\texttt{IntFilter}\ s)\ a_2\ f \Rightarrow (\texttt{Apps}\ s)\ a_1 \Rightarrow (\texttt{Apps}\ s)\ a_2 \Rightarrow (\texttt{Compf}\ \ (\texttt{Manifest}\ \ a_1))\ c_1\ f \Rightarrow (\texttt{Compf}\ \ (\texttt{Manifest}\ \ a_2))\ c_2 f \Rightarrow c_1 = c_2 \Rightarrow a_1 = a_2. \tag{18}$$

We need to prove the same for the state s'. Hence we assume

$$(a_1, a_2 : \texttt{App}),\ (c_1, c_2 : \texttt{Comp}),\ (f : \texttt{IntentFilter}),\ (\texttt{IntFilter}\ s')\ a_1\ f,\ (\texttt{IntFilter}\ s')\ a_2\ f, (\texttt{Apps}\ s')\ a_1,\ (\texttt{Apps}\ s')\ a_2,\ (\texttt{Compf}\ \ (\texttt{Manifest}\ \ a_1))\ c_1\ f,\ (\texttt{Compf}\ \ (\texttt{Manifest}\ \ a_2))\ c_2 f,\ c_1 = c_2. \tag{19}$$

Now it is enough to show that $a_1 = a_2$. By the postcondition of API (sub-condition 1), in 19, we can replace $(\texttt{IntFilter}\ s')\ a_1\ f$, $(\texttt{IntFilter}\ s')\ a_2\ f$, $(\texttt{Apps}\ s')\ a_1$, and $(\texttt{Apps}\ s')\ a_2$, with $(\texttt{IntFilter}\ s)\ a_1\ f$, $(\texttt{IntFilter}\ s)\ a_2\ f$, $(\texttt{Apps}\ s)\ a_1$, and $(\texttt{Apps}\ s)\ a_2$, respectively. Then we get the precondition of implication of 18. Hence, we conclude $a_1 = a_2$.

□

Lemma 2 *1.* $(\mathit{ExstAppforWatInt}(s) \wedge \mathit{API} : s \rightsquigarrow s') \Rightarrow \mathit{ExstAppforWatInt}(s')$.
2. $(\mathit{NoRepImpInts}(s) \wedge \mathit{API} : s \rightsquigarrow s') \Rightarrow \mathit{NoRepImpInts}(s')$.
3. $(\mathit{RunInsInSysComp}(s) \wedge \mathit{API} : s \rightsquigarrow s') \Rightarrow \mathit{RunInsInSysComp}(s')$.
4. $(\mathit{ResExstIntIntF}(s) \wedge \mathit{API} : s \rightsquigarrow s') \Rightarrow \mathit{ResExstIntIntF}(s')$.

The following theorem results from Lemmas 1 and 2 .

Theorem 1 $(\mathit{WellDefined}(s) \wedge \mathit{API} : s \rightsquigarrow s') \Rightarrow \mathit{WellDefined}(s')$.

7 Semantics utilization

This section employs the semantics presented in the previous section. The semantics is a base towards formalizing basic and advanced aspects of the Android framework ICC. Here, we demonstrate the utility of proposed semantics towards developing a robust Android ICC framework.

Modifying some fields (action, data, category, and extra) in an Intent does not change the Intent type. However, updating the attribute CmpId possibly change the Intent type. These facts and similar ones are formalized in the Lemmas 3 and 4 .

Lemma 3 *1.* $(\mathit{ExpInt}(i) \wedge \mathit{setAction}(i, act) : s \rightsquigarrow s') \Rightarrow \mathit{ExpInt}(i)$.
2. $(\mathit{ImpInt}(i) \wedge \mathit{setAction}(i, act) : s \rightsquigarrow s') \Rightarrow \mathit{ImpInt}(i)$.
3. $(\mathit{ExpInt}(i) \wedge \mathit{setData}(i, dat) : s \rightsquigarrow s') \Rightarrow \mathit{ExpInt}(i)$.
4. $(\mathit{ImpInt}(i) \wedge \mathit{setData}(i, dat) : s \rightsquigarrow s') \Rightarrow \mathit{ImpInt}(i)$.
5. $(\mathit{ExpInt}(i) \wedge \mathit{addCategory}(i, cat) : s \rightsquigarrow s') \Rightarrow \mathit{ExpInt}(i)$.
6. $(\mathit{ImpInt}(i) \wedge \mathit{addCategory}(i, cat) : s \rightsquigarrow s') \Rightarrow \mathit{ImpInt}(i)$.
7. $(\mathit{ExpInt}(i) \wedge \mathit{putExtra}(i, ef, ev) : s \rightsquigarrow s') \Rightarrow \mathit{ExpInt}(i)$.
8. $(\mathit{ImpInt}(i) \wedge \mathit{putExtra}(i, ef, ev) : s \rightsquigarrow s') \Rightarrow \mathit{ImpInt}(i)$.

Lemma 4 *1.* $\mathtt{ImpInt}(i) \wedge \mathtt{cid} = \mathtt{None} \wedge \mathtt{setComponent}(i, \mathtt{cid}) : s \rightsquigarrow s' \Rightarrow \mathtt{ImpInt}(i)$.
2. $\mathtt{ImpInt}(i) \wedge \mathtt{cid} \neq \mathtt{None} \wedge \mathtt{setComponent}(i, \mathtt{cid}) : s \rightsquigarrow s' \Rightarrow \mathtt{ExpInt}(i)$.
3. $\mathtt{ExpInt}(i) \wedge \mathtt{cid} \neq \mathtt{None} \wedge \mathtt{setComponent}(i, \mathtt{cid}) : s \rightsquigarrow s' \Rightarrow \mathtt{ExpInt}(i)$.

Our proposed semantics does not cache results of `Intent` resolution process. Hence, if two APIs use the same explicit `Intent`, the system will resolve the `Intent` twice. Therefore, the system does not benefit from the first resolution to the second one. This results in resource wastage which can be handled. Further, such instances have serious security implications. For example, if the first `Intent` resolution is responsible for sensitive data leakage, it will be repeated in the second `Intent` resolution.

Before running the `Intent` resolution process, it is convenient to check whether a matching component has been found recently. If this is the case, the system should check whether the component is responsible for a security breach. If there is no history of security breaches, then the system can declare the component to be a match. Otherwise, the system runs the `Intent` resolution process. The proposed updates can be implemented in our proposed semantics employing the following changes. The first change is to add the component $\mathtt{BadIntResol} : \mathtt{Intent} \times \mathtt{IntentFilter}$ to the definition of semantic state (`SysState`). This augmentation, allows a state to store the history of `Intent` resolution that led to security breaches. The other change is to replace Rule 3 (R.H.S.) with the following three rules:

$$\frac{\neg(\mathtt{IntentResol}\ s)(i, f) \quad \mathtt{ActionTest}(i, f) \quad \mathtt{CategoryTest}(i, f) \quad \mathtt{DataTest}(i, f)}{\mathtt{IntMatchIntF}_2(i, f)} \tag{20}$$

$$\frac{(\mathtt{IntentResol}\ s)\ i\ f \quad \neg(\mathtt{BadIntentResol}\ s)\ i\ f}{\mathtt{IntMatchIntF}_1(i, f)} \tag{21}$$

$$\frac{\mathtt{IntMatchIntF}_1(i, f) \vee \mathtt{IntMatchIntF}_2(i, f)}{\mathtt{IntMatchIntF}^{mod}(i, f)} \tag{22}$$

With these changes, we can prove that executing `Intent` APIs such as `startActivity(i)` is secure as formalized in Theorem 2. The theorem guarantees that executing $\mathtt{startActivity}(i)$ for an implicit intent i does not lead to invoking activities having history of security breaches.

Theorem 2 *Suppose* $\mathtt{API} \in \{\mathtt{startActivity}, \mathtt{startService}, \mathtt{sendBroadcast}\}$. *Suppose also that* $\mathtt{ImpInt}(i) \wedge \mathtt{API}(i) : s \rightsquigarrow s'$. *Then there exists* $c \in \mathtt{Comp}$, $ic \in \mathtt{IComp}$, *and* $f \in \mathtt{IntentFilter}$ *such that:*

1. $\mathtt{IsActivity}(c) \vee \mathtt{IsService}(c) \vee \mathtt{IsBroadCast}(c)$,
2. $\mathtt{CompDwnloaded}(c, s) \wedge \mathtt{InsCompNotInState}(ic, s) \wedge (\mathtt{Active}\ s')\ ic\ c \wedge \mathtt{InFComp}(c, f)$,
3. $(\mathtt{IntentResol}\ s)\ i\ f \wedge (\mathtt{IntentResol}\ s)\ i\ f \Rightarrow \neg(\mathtt{BadIntentResol}\ s)\ i\ f$, *and*

4. $\neg(\texttt{IntentResol}\ s)\ i\ f \Rightarrow \texttt{ActionTest}(i, f)$, $\texttt{CategoryTest}(i, f)$, *and* $\texttt{DataTest}(i, f)$.

According to the theorem above the API execution results in an instance ic of a component c that has an `Intent` filter f such that the following conditions hold. The instance ic does not belong to s, but belongs to s'. If f has recently been identified as a match to i, then c has no history of security breaches. This is formalized using the condition $\neg(\texttt{BadIntentResol}\ s)\ i\ f$. If there is no history of matching between i and f ($\neg(\texttt{IntentResol}\ s)\ i\ f$), then f has passed the action, category, and data tests.

Executing `startActivity` creates new instance. In a state s and for an explicit intent i, the execution of the `startActivity`(i) API results in an instance ic of a component c such that the following holds. c is an activity and ic does not belong to s, but belongs to and is running in s'. The id of c coincides with that specified in attribute `CmpId` of i. This is formalized in the following lemma.

Lemma 5 *Suppose* $\texttt{ExpInt}(i)$ *and* $\texttt{startActivity}(i) : s \rightsquigarrow s'$. *Then there exists* $c \in \texttt{Comp}$ *and* $ic \in \texttt{IComp}$ *such that* $\texttt{InsCompNotInState}(ic, s) \wedge (\texttt{Active}\ s')\ ic\ c \wedge \texttt{CompDwnloaded}(c, s) \wedge \texttt{IsActivity}(c) \wedge \texttt{CmpId}\ i = \texttt{RetCmpId(c)}$.

It is not possible to execute `sendBroadcast`(i) **in a state** s **if** i **is an implicit intent that can not be resolved in** s. Suppose that in state s, i is an implicit intent that was not resolved recently. Additionally, suppose that a is the action value of i. Further, each intent filter in the system does not include a in its action list, it is not possible to execute the APIs {`startActivity`(i), `startService`(i), `sendBroadcast`(i)} in the state s. This is formalized in the following lemma, proof is available in the verification code.

Lemma 6 *Suppose that* $\texttt{ImpInt}(i) \wedge \texttt{Action}\ i = a \wedge \forall f \in \texttt{IntentFilter}, \neg(\texttt{IntentResol}\ s)\ i\ f$, *and* $\forall(c : \texttt{Comp})(f : \texttt{IntentFilter}), \texttt{CompDwnloaded}(c, s) \wedge \texttt{InFComp}(c, f) \Rightarrow \texttt{act} \notin (\texttt{Action}^*\ f)$. *Then* $\neg\texttt{pre\_startActivity}(i, s) \wedge \neg\texttt{pre\_startService}(i, s) \wedge \neg\texttt{pre\_sendBroadcast}(i, s)$.

It is not possible to execute `startService`(i) **in a state** s **if,** i **is an explicit intent pointing to a component that is not installed in** s. We assume that in state s i is an explicit intent. We expect that $CmpId$ field of i does not coincide with that of any component in s. In such situations, it is not possible to execute the APIs {`startActivity`(i), `startService`(i), `sendBroadcast`(i)} in the state s. This is formalized in the following lemma.

Lemma 7 *Suppose* $\texttt{ExpInt}(i) \wedge \forall c \in \texttt{Comp}, \texttt{CompDwnloaded}(c, s) \Rightarrow \texttt{CmpId}\ i \neq \texttt{RetCmpId(c)}$. *Then* $\neg\texttt{pre\_startActivity}(i, s) \wedge \neg\texttt{pre\_startService}(i, s) \wedge \neg\texttt{pre\_sendBroadcast}(i, s)$.

Turning an implicit intent i **into an explicit one and then executing** `startService`(i) **results in a new service instance in the final state**. Suppose that in a state s, i is an implicit intent. Suppose that `cid` is the Id of the service c. Suppose that i is turned into explicit intent via assigning `cid` to the filed $CmpId$ of i.

Then executing startService(i) results in a new service instance of c in the final state. This is formalized in the following Theorem.

Theorem 3 *Suppose that* $ImpInt(i) \wedge setComponent(i, cid) : s \rightsquigarrow s' \wedge startService(i) : s' \rightsquigarrow s''$. *Then there exists* $c \in Comp$ *and* $ic \in IComp$ *such that:* $InsCompNotInState(ic, s) \wedge (Active\ s'')\ ic\ c \wedge IsService(c) \wedge RetCmpId(c) = cid$.

Executing sendBroadcast(i) with an implicit intent i results in a new Broadcast instance matching i. Suppose that in a state s, i is an implicit intent. Suppose that act is the value of the action property is i. We assume that i was not resolved recently. Then, executing sendBroadcast(i) results in a new a broadcast instance. The broadcast of the instance is installed in the initial state which includes a filter that matches i. This is formalized in the following theorem.

Theorem 4 *Suppose* $ImpInt(i) \wedge Action\ i = act \wedge (\forall f \in IntentFilter, \neg (IntentResol\ s)\ i\ f) \wedge sendBroadcast(i) : s \rightsquigarrow s'$. *Then there exists* $c \in Comp$, $f \in IntentFilter$, *and* $ic \in IComp$ *such that:* $InsCompNotInState(ic, s) \wedge (Active\ s')\ ic\ c \wedge CompDwnloaded(c, s) \wedge IsBroadCast(c) \wedge InFComp(c, f) \wedge act \in (Action^*\ f)$.

8 Data-leakage aware Intent resolution (DLAIR)

This section presents a new security-aware algorithm (DLAIR) for Intent resolution. The algorithm relies on the formal model presented in the previous sections of the paper in addition to $\mathcal{L}ek\mathcal{I}nt$, our lightweight tool that identifies suspicious paths related to Android app Intents. In this context, $\mathcal{L}ek\mathcal{I}nt$ can be thought of as a security analysis tool for reporting execution paths with Intents that are suspicious of leaking sensitive user data in Android apps. DLAIR is outlined in Algorithm 1. DLAIR and the design, implementation, and evaluation of $\mathcal{L}ek\mathcal{I}nt$ (as the main pillar of DLAIR) are presented in the following subsections.

8.1 Design of DLAIR

DLAIR, as listed in Algorithm 1, takes as input an Intent i of the app p. For i, the algorithm builds two sets of matching Intent Filters (Good_Filters and Suspicious_Filters). Theses sets are initialized in steps 1 and 2 of the algorithm. The set Good_Filters has matching filters of i that are secure and the set Suspicious_Filters has matching filters of i that are suspicious of leaking sensitive data. In Step 3, the algorithm checks if i is implicit. Step 4 calls $\mathcal{L}ek\mathcal{I}nt$ that returns a triple $[\gamma, \omega, \beta]$, where:

1. γ is a Boolean that is 1 if the execution path hosting i is suspicious of leaking sensitive data,
2. ω specifies the component type targeted by i, and
3. β is the permission required to access the component targeted by i, if the component is protected by a permission. Otherwise, β is none.

Algorithm 1 `Data-leakage aware Intent resolution (DLAIR)`

Input: An intent i in an application p.
Output: (`Good_Filters`, `Suspicious_Filters`).
Steps:

1: `Good_Filters` $\leftarrow \{\}$;
2: `Suspicious_Filters` $\leftarrow \{\}$;
3: **if** `Implicit_Intent(i)` **then**
4: $[\gamma, \omega, \beta] \leftarrow \mathcal{L}ek\mathcal{I}nt(p, i)$;
5: **if** $\gamma == 1$ **then**
6: **if** ω == `broadcast` **then**
7: **if** $\beta \in$ `signature permissions` **then**
8: `Good_Filters` $\leftarrow \{f \mid f \in \texttt{IntMatchIntF}^{mod}(i, f)\}$;
9: **else**
10: `Good_Filters` $\leftarrow \{f \mid f \in f^{sig\_perm} \wedge f \in \texttt{IntMatchIntF}^{mod}(i, f)\}$;
11: **if** $\omega \in \{$`activity, service`$\}$ **then**
12: `Good_Filters` $\leftarrow \{f \mid f \in f^{sig\_perm} \wedge f \in \texttt{IntMatchIntF}^{mod}(i, f)\}$;
13: **else**
14: `Good_Filters` $\leftarrow \{f \mid f \in \texttt{IntMatchIntF}^{mod}(i, f)\}$;
15: **else** %(Explicit_Intent(i))%
16: `Good_Filters` $\leftarrow \{f \mid f \in \texttt{IntMatchIntF}(i, f)\}$;
17: `Suspicious_Filters` $\leftarrow \{f \mid f \in \texttt{IntMatchIntF}(i, f)\} \setminus$ `Good_Filters`;
18: **return** (`Good_Filters`, `Suspicious_Filters`);

Recall that Android allows `Intents` targeting broadcasts to be protected by permissions. This is not the case for activities and services. Therefore, if ω is broadcast (Step 6), and if β is a signature permission (Step 7), then Step 8 uses our proposed algorithm of `Intent` resolution ($\texttt{IntMatchIntF}^{mod}$) to build the set `Good_Filters`. If β is not signature permission or the component type is not a broadcast, then the algorithm secures the matching process by accepting only filters protected by signature permissions. This is done in Steps 10 and 12. If $\gamma = 0$, then there is no need to care for permissions. This is implemented in Step 14. If i is not implicit, then in Step 16, the traditional algorithm of Intent resolution (outlined in Fig. 2) is used to find the set of `Good_Filters`. In all cases, Step 17 defines `Suspicious_Filters` as the set of all filters produced by the traditional algorithm of Intent resolution (outlined in Fig. 2) minus `Good_Filters`.

It is necessary to note that `DLAIR` is not a taint analysis. While taint analysis is static and aims at fining data leaks, `DLAIR` is dynamic analysis. Taint analysis usually is applied once (on one app) at installation time. `DLAIR` can be realized as an enrichment of the process of `Intent`-resolutions. This enrichment aims at avoiding repetition of insecure `Intent`-resolutions. Hence `DLAIR` works on multiply applications simultaneously. While Taint analysis is typically applied as an extra layer above the Android OS, `DLAIR` can conveniently be integrated into the Android OS. *Flowdroid* [3] is built on the *Soot* tool [27] which replies on optimizing the Java byte code of apk files. This optimization increases significantly the rune time of the tool. Though disabling the optimization part of *Soot* is possible, the default configuration of *Soot* is still not fast enough as realized in many tools, such as Dare [22] and ded [14]. This fact has led researchers to utilize other tools (such as *Androguard*) as reviewed in [24]. Accuracy and soundness arguments [19] of *Soot* are among the challenges not

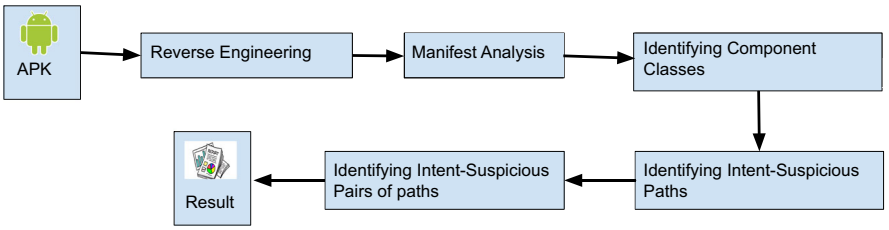

Fig. 7 Overview of $\mathcal{L}$ekInt

solved yet [24]. This partially justifies that DLAIR gave better results than *Flowdroid* as explained later in detail.

8.2 Design of $\mathcal{L}$ekInt

$\mathcal{L}$ekInt is useful as a lightweight technique for testing the security of the execution path of a given Intent. In this context, an execution path is **intent-suspicious** if it contains the following three Dalvik instructions in the following specific order:

- The first instruction defines an intent object,
- The second instruction invokes a source API that reads sensitive data.
- The third instruction invokes a sink API that takes the intent of the first instruction as an argument.

Figure 7 presents an overview of $\mathcal{L}$ekInt which is composed of the following modules:

- *Reverse engineering module:* aims at decompiling the apk file to get the manifest and dex files.
- *Manifest analysis module:* collects lists of activities, broadcast receivers, and services. The analysis decide which elements of these lists are protected by permissions.
- *Identifying Component Classes module:* identifies the dex code of classes of different components used in the app. The following stages use these classes.
- *Identifying Intent-Suspicious paths module:* investigates paths of methods of component classes. Then paths are analyzed to determine which paths are suspicious according to our definition above. For a suspicious path, the analysis also specifies whether the third instruction, of intent-suspicious definition, is protected with permission. This can be the case for sink methods such as sendBroadcast. The analysis also specifies the type of component object (if any) started by the path.
- *Identifying Intent-Suspicious Pairs of paths module:* discovers more involved patterns on leaking data. This pattern was determined from analyzing results of state-of-the-art techniques reporting data leakage such as Flowdroid. In this case, the pattern is composed of two suspicious paths; the first path contains the first and second instructions of the definition of intent-suspicious paths. This path invokes another method that includes the second path which contains the third instruction of definition of intent-suspicious paths. Details specified for the suspicious path of the previous module are also collected for the two paths of this module.

8.3 Implementation of $\mathcal{L}$ekInt

The details of implementing modules of Fig. 7 are as follows:

- The reverse engineering is done using Androguard and produces three objects a, d, dx.
- The manifest analysis is implemented using several Androugard APIs including `a.find_tags('receiver')` to get a list of receiver tags. Similarity tags of other components can be obtained. The API `tag.get(a._ns ('permission'))` is used to check if the component is protected using a permission.
- The implementation of *Identifying Component Classes* relies on the following APIs to get dex classes of different components: `a.get_services()` and `dx.get_class_analysis(c.get_name())`.
- For the implementation of *Identifying Suspicious Intent paths* and *Identifying Intent-Suspicious Pairs of paths* phases, the paths of methods of components are built using the APIs `basic_blocks.gets()` and `CurBblock.childs`. The analysis of the collected paths relies on identifying path instructions by the API `block.get_instructions()`. The analysis of instructions details utilizes the APIs `get_name()` and `get_operands()`.

Our list of source and sink APIs were derived from ones used by state-of-the-art techniques such as *Flowdroid*. The list includes all common sources and sinks related to intents and bundles only. This list is available online with our result files. Among sink APIs in our list are the sends broadcast ones and the send service API.[5]

8.4 Performance evaluation

This section presents the results of our experiments conducted for evaluating $\mathcal{L}$ekInt. We got the dataset from a reputable benchmark, namely Andro-Zoo [2]. We downloaded a random set of 2000 applications whose *sha256* is available online with our result files. We limited the download to applications that Androzoo obtained from the *Google Play Store*. All experiments were done on a Dell (Vostro) device with processor: Intel(R) Core(TM) i7-3612 QM CPU @ 2.10 GHz, 8.00GB RAM, and Windows 10 (64-bits) operating system. All implementations were written in Python. Table 3 shows the statistics collected for comparing $\mathcal{L}$ekInt against *Flowdroid* [3] as one of the state-of-the-art techniques that are most common and related to $\mathcal{L}$ekInt. We make all our results and data files available online.[6] These files include 2000 text files produced by $\mathcal{L}$ekInt and 1999 text produced by *Flowdroid* for the applications of the dataset. The Coq files that include the proofs of our semantics are also available online.

Our experiments answer the following research questions:

[5] https://developer.android.com/reference.

[6] https://github.com/maelzawawy/LekInt.

Table 3 Statistics comparing $\mathcal{L}$ekInt against *Flowdroid*

Aspect	$\mathcal{L}$ekInt	*Flowdroid*
Number of leaks	49,015	4664
Number of apps with reported leaks	824	425
Total time (in sec)	95,740	149,322
Average time per app (in sec)	47.8	74.6
Number of analyzed classes	150,914	*
Number of activities from manifest	29,564	*
Number of activities from manifest protected with permissions	30	*
Number of services from manifest	5165	*
Number of services from manifest protected with permissions	566	*
Number of receivers from manifest	4496	*
Number of receivers from manifest protected with permissions	974	*
Leaks related to `Intents` starting activities	13707	*
Leaks related to `Intents` starting services	2761	*
Leaks related to `Intents` starting broadcast	2574	*
Leak paths using permissions	435	*

The symbol $*$ means that the technique does not provide the statistics

RQ1. **Applicability, Effectiveness, and Performance**: How does $\mathcal{L}$ekInt compare against alternative approaches applicability, discovering suspicious paths related to `Intents`, and in performance?
RQ2. **Accuracy**: How accurate are the results of $\mathcal{L}$ekInt against alternative approaches?
RQ3. **Common Component Leaking Data**: Is there a specific component that is targeted most by Intents used in paths leaking data?
RQ4. **Permissions**: How common is the use of permissions in protecting components in manifest and in protecting intents in the native code?

Applicability, Effectiveness, and Performance: The applicability of $\mathcal{L}$ekInt is proved by its results obtained for 2000 real-life Android applications. Moreover, we designed two Android applications that include different patterns of paths leaking sensitive data. These paths include `Intents` that target different types of components. To cover all possible cases in our applications, some of `Intents` and components were protected by permissions and others were not. $\mathcal{L}$ekInt succeeded in reporting all the leaking paths in the example applications. On the other hand, *Flowdroid* failed to report leaks in these two applications which we make available online along with their $\mathcal{L}$ekInt and *Flowdroid* results.

Figure 8 (Right-hand-side) provides comparisons for leaks reported by $\mathcal{L}$ekInt and *Flowdroid*. The total number of leaks reported by $\mathcal{L}$ekInt and *Flowdroid* are 49015 and 4664, respectively. It is worth mentioning that *Flowdroid* reported more leaks than this number, but we considered only leaks related to our problem. For example, *Flowdroid* typically reports leaks in external components that are not among the native code of the app under analysis. Examples of

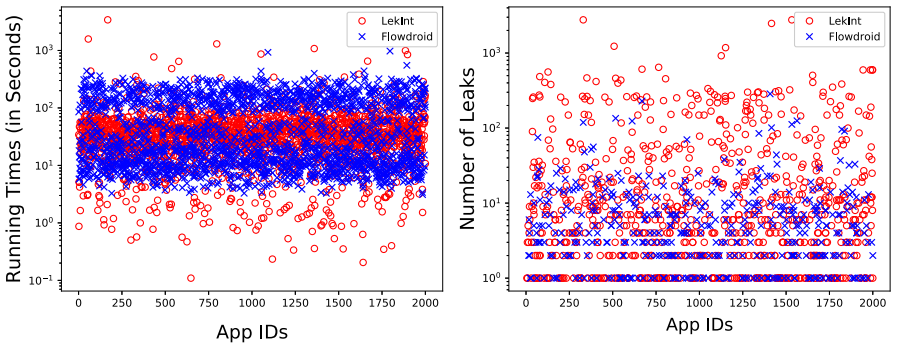

Fig. 8 Comparing runtime and reported vulnerabilities in $\mathcal{L}\texttt{ek}\mathcal{I}\texttt{nt}$ and *Flowdroid*

Table 4 Accuracy of $\mathcal{L}\texttt{ek}\mathcal{I}\texttt{nt}$ against *Flowdroid*

Technique	True positive	False positive	True negative	False negative
$\mathcal{L}\texttt{ek}\mathcal{I}\texttt{nt}$	26	13	53	8
Flowdroid	13	4	62	21

external components are `com.facebook.CampaignTrackingReceiver` and `com.google.android.gms.ads.AdActivity`. Such leaks were not considered in our evaluation. The reported number of leaks proves that $\mathcal{L}\texttt{ek}\mathcal{I}\texttt{nt}$ is more effective than *Flowdroid*.

Figure 8 (Left-hand-side) provides comparisons for running times of $\mathcal{L}\texttt{ek}\mathcal{I}\texttt{nt}$ and *Flowdroid*. The average running times per application for $\mathcal{L}\texttt{ek}\mathcal{I}\texttt{nt}$ and *Flowdroid* are 47.8 and 74.6 seconds, respectively. This proves that the performance and efficiency of $\mathcal{L}\texttt{ek}\mathcal{I}\texttt{nt}$ are better than that of *Flowdroid*.

Accuracy: $\mathcal{L}\texttt{ek}\mathcal{I}\texttt{nt}$ reported suspicious leaks in 824 apps, while the *Flowdroid* reported leaks in 425 apps. We manually checked the results reported for a random set of 100 apps for measuring the accuracy of $\mathcal{L}\texttt{ek}\mathcal{I}\texttt{nt}$ against *Flowdroid*. Table 4 illustrates the results of our manual investigation.

Compared to $\mathcal{L}\texttt{ek}\mathcal{I}\texttt{nt}$, *Flowdroid* has higher false-negative and lower false positive. This is in line with the approximation adopted by $\mathcal{L}\texttt{ek}\mathcal{I}\texttt{nt}$. However, the efficiency and the high number of true positives reported by $\mathcal{L}\texttt{ek}\mathcal{I}\texttt{nt}$ confirm its accuracy and usefulness.

Common Component for Leaking Data and Permissions Figure 9 answers research questions 3 and 4. The lower set of bars show that activity components collected from manifest files of the dataset are much more than service and broadcast-receiver components. However, a high percentage (21%) of these broadcast components are protected by permissions against 0.1% only for activity components. This is clear from the middle set of bars of the figure. The abbreviation MC in the y axis stands for manifest components. For some leaking paths, $\mathcal{L}\texttt{ek}\mathcal{I}\texttt{nt}$ was able to determine the type of component targeted by the Intent of the path. The upper set of bars of Fig. 9 shows that 72% of these paths targeted activities. Therefore it is com-

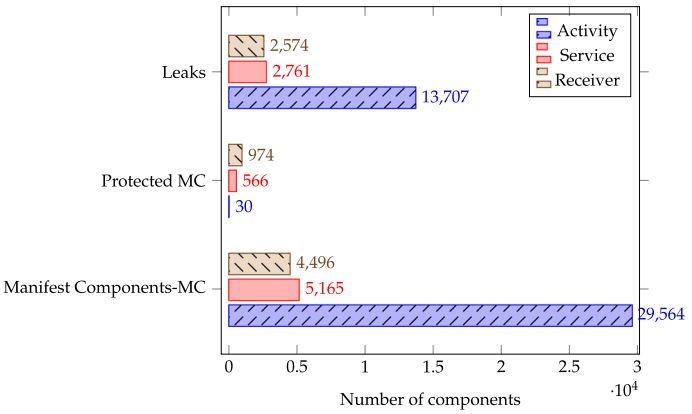

Fig. 9 Comparing statistics collected by $\mathcal{L}$ekInt for different types of Android components and for leaking paths protection by permissions

mon to use activities rather than other types of components to leak data. Only 0.8% of the paths leaking data were protected by permissions. This supports the logic of our suggested algorithm for Intent resolution *DLAIR*. This is so because *DLAIR* tends to trust Intents protected by permissions.

Use Cases: For some apps of our data set *Flowdroid* did not report any leaks but our technique $\mathcal{L}$ekInt did. Some of these apps have the following details:

1. The first app has an execution path in the activity named "TiActivity". The path leaks data in a sequence of instructions that start another activity. The source API, in this case, is `getIntent` and the sink API is `putExtra`.
2. The second app has a leaking execution path that uses an Intent to start the activity "FuturePaymentConsentActivity". The source API in this case is `getIntent` and the sink API is `putExtra`.
3. The third app has a leaking path in its activity named "BaseFragmentActivity". The path uses an Intent to start a service.

Also for our designed app (available online with results) $\mathcal{L}$ekInt discovered all the inserted patterns of leaks but *Flowdroid* did not.

Author Contributions All authors participated equally.

Funding Open access funding provided by The Science, Technology &; Innovation Funding Authority (STDF) in cooperation with The Egyptian Knowledge Bank (EKB).

Data availability The details of the dataset and results of our experiments are available online on: https://github.com/maelzawawy/LekInt.

Declarations

Animal research (Ethics) Not applicable.

Consent to participate (Ethics) Not applicable.

Consent to publish (Ethics) Consent to submit and publish has been received explicitly from all co-authors.

Plant reproducibility Not applicable.

Clinical Trials Registration Not applicable.

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Aafer Y, Du W, Yin H (2013) Droidapiminer: Mining api-level features for robust malware detection in android. In: International conference on security and privacy in communication systems. Springer, pp 86–103
2. Allix K, Bissyandé TF, Klein J, Le Traon Y (2016) Androzoo: collecting millions of android apps for the research community. In: 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR). IEEE, pp 468–471
3. 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
4. Barros P, Just R, Millstein S, Vines P, Dietl W, Ernst MD, et al (2015) Static analysis of implicit control flow: resolving java reflection and android intents (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp 669–679
5. Betarte G, Campo J, Gorostiaga F, Luna C (2017) A certified reference validation mechanism for the permission model of android. arXiv preprint arXiv:1709.03652
6. Betarte G, Campo J, Luna C, Romano A (2016) Formal analysis of android's permission-based security model. Sci Ann Comput Sci 26(1):27
7. Bornstein D (2008) Dalvik vm internals. In: Google I/O developer conference, vol 23, pp 17–30
8. Burd B, Mueller JP (2020) Android application development all-in-one for dummies. Wiley, Hoboken
9. Chin E, Felt AP, Greenwood K, Wagner D (2011) Analyzing inter-application communication in android. In: Proceedings of the 9th international conference on Mobile systems, applications, and services. ACM, pp 239–252
10. Chlipala A (2013) Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant. MIT Press, Cambridge
11. Developers A (2011) What is android? Dosegljivo: http://www.academia.edu/download/30551848/andoid-tech.pdf
12. Developers A. Developer guides: Application fundamentals. https://developer.android.com/guide/components/fundamentals.html. Accessed in 2021
13. Developers A. Developer guides: Intents and intent filters. https://developer.android.com/guide/components/intents-filters.html. Accessed in 2021
14. Enck W, Octeau D, McDaniel PD, Chaudhuri S (2011) A study of android application security. In: USENIX security symposium, vol 2
15. Faruki P, Bharmal A, Laxmi V, Ganmoor V, Gaur MS, Conti M, Rajarajan M (2015) Android security: a survey of issues, malware penetration, and defenses. IEEE Commun Surv Tutor 17(2):998–1022
16. Feizollah A, Anuar NB, Salleh R, Suarez-Tangil G, Furnell S (2017) Androdialysis: analysis of android intent effectiveness in malware detection. Comput Secur 65:121–134
17. Hall SP, Anderson E (2009) Operating systems for mobile computing. J Comput Sci Coll 25(2):64–71

18. Li Y, Yao F, Lan T, Venkataramani G (2016) Sarre: semantics-aware rule recommendation and enforcement for event paths on android. IEEE Trans Inf Forensics Secur 11(12):2748–2762
19. Livshits B, Sridharan M, Smaragdakis Y, Lhoták O, Amaral JN, Chang BYE, Guyer SZ, Khedker UP, Møller A, Vardoulakis D (2015) In defense of soundiness: A manifesto. Commun ACM 58(2):44–46
20. Meier R (2012) Professional Android 4 application development. Wiley, Hoboken
21. Nimodia C, Deshmukh H (2012) Android operating system. Softw Eng 3(1):10
22. Octeau D, Jha S, McDaniel P (2012) Retargeting android applications to java bytecode. In: Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering, pp 1–11
23. Payet E, Spoto F (2014) An operational semantics for android activities. In: Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation, pp 121–132
24. Reaves B, Bowers J, Gorski SA III, Anise O, Bobhate R, Cho R, Das H, Hussain S, Karachiwala H, Scaife N et al (2016) * droid: Assessment and evaluation of android application analysis tools. ACM Comput Surv (CSUR) 49(3):1–30
25. Schmerl B, Gennari J, Cámara J, Garlan D (2016) Raindroid-a system for run-time mitigation of android intent vulnerabilities
26. Tang J, Cui X, Zhao Z, Guo S, Xu X, Hu C, Ban T, Mao B (2017) Nivanalyzer: A tool for automatically detecting and verifying next-intent vulnerabilities in android apps. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp 492–499
27. Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V (2010) Soot: a java bytecode optimization framework. In: CASCON First Decade High Impact Papers, pp 214–224
28. Wognsen ER, Karlsen HS, Olesen MC, Hansen RR (2014) Formalisation and analysis of dalvik bytecode. Sci Comput Program 92:25–55
29. Xu K, Li Y, Deng RH (2016) Iccdetector: Icc-based malware detection on android. IEEE Trans Inf Forensics Secur 11(6):1252–1264