IIFA: Modular Inter-app Intent Information Flow Analysis of Android Applications

Abhishek Tiwari, Sascha Groß, and Christian Hammer

University of Potsdam, Potsdam, Germany
{tiwari, saschagross, chrhammer}@uni-potsdam.de

Abstract. Android apps cooperate through message passing via intents. However, when apps have disparate sets of privileges inter-app communication (IAC) can accidentally or maliciously be misused, e.g., to leak sensitive information contrary to users' expectations. Recent research has considered static program analysis to detect dangerous data leaks due to inter-component communication (ICC), but suffers from shortcomings for IAC with respect to precision, soundness, and scalability.
As a remedy we propose a novel pre-analysis for static ICC/IAC analysis. Our main contribution is the first fully automatic ICC/IAC information flow analysis that is scalable for realistic apps due to modularity, avoiding combinatorial explosion: Our approach determines communicating apps using short summaries rather than inlining intent calls between components and apps, which entails simultaneously analyzing all apps installed on a device.
Using benchmarks we establish that IIFA outperforms state-of-the-art analyses in terms of precision and recall. But foremost, applied to the 90 most popular applications from the Google Playstore, IIFA demonstrated its scalability to a large corpus of real-world apps.

Keywords: Android, Inter-Component Communication, Intent, Static Analysis

1 Introduction

To protect sensitive information on Android, various information flow control (IFC) analyses have been developed. These analyze the (potential) flow of information in apps and report a warning if a flow from a sensitive data source to an untrusted/public data sink (like sending sensitive information to the internet) is determined to be possible at runtime. Information flow is not restricted to a single component, but occurs frequently between components of the same [16,11] and even different apps [22]. Our study, using the top 90 apps from the Google play store, revealed more than 10,000 inter-component calls. Scrutinizing the flows between components therefore becomes imperative.

Android's inter-component communication (ICC) mainly leverages so-called *intents.* The major challenge in identifying IFC through intents is identifying which information flows from one component to another. Leveraging static analysis is non-trivial because the receiver and the intent data may be unknown at analysis time, being strings that might be composed at runtime.

Some tools consider intents during information flow analysis [4,14,16,22] but suffer from multiple shortcomings: For ICC flows, they have a mediocre precision and recall, but fail significantly for IAC. To match senders and receivers these approaches merely verify that the receiver-identifying data matches but ignore other intent attributes that need to correspond, which leads to significant imprecision. Further, the majority of related works [2,13,16,22] propose a merging-based approach for IAC, inlining senders and receivers into a huge singleton "app" to analyze, which does not scale up for realistic apps, and even if it did, the complexity to analyze the merged app inflates. However, as most combinations of apps do not communicate via intents the whole effort is mostly futile. Simultaneously, the merging process itself may introduce spurious data flow paths, increasing analysis imprecision. Even worse, each time an app is updated or installed on a device this merging and re-analysis process must be repeated.

Our Contributions. In this work we propose a novel information-flow analysis for IAC (and ICC) based on an intent-flow pre-analysis that evades combinatorial explosion of analyzing all potential communication partners, while excluding infeasible communication paths. Our approach can predict which combinations of apps communicate by separating information flow analysis within app components from thorough matching of communication partners. In a first step we create a database of summary information about senders of intents, their characteristics, as well as apps registered to receive implicit intents. Subsequently, we identify potential communication partners based on a novel matching algorithm which takes the potential intra-component flows into account. These flows are provided by a baseline IFC analysis for all potential intent receivers. We leverage senders' outbound intent data as input to the information flows identified in respective receivers, eliminating the need for inlining or merging apps and thus combinatorial explosion, as only summaries of actual communication partners are subsumed. In case multiple apps are involved in intent communication our approach performs a fixed point iteration through the DB information. Remember that our tool is not a stand-alone IFC analysis tool. Rather, IIFA leverages flows and slices generated by other IFC analyzers. As these tools are already heavily engineered for the intra-app case, we concentrated on the peculiarities of intent communication and evasion of inlining and combinatorial explosion.

As a noteworthy novelty, our approach is modular and thus compositional with respect to app installation. Whenever a new (version of an) app is available for analysis, the database is updated (in case of new version) or extended (new app) to include the intents broadcast or received by this app. Only the new app has to be (re-)analyzed, as well as combinations with flows identified in potential receivers. We aim to answer the following research questions:

- **RQ1:** *Does our pre-analysis approach negatively impact the precision or soundness of the results with respect to state-of-the-art analyses?*
- **RQ2:** *Does our approach scale to a realistic corpus of real-world apps?*

We implemented our approach as a tool called *IIFA* and evaluated it on DroidBench [6], the IccTA extension of DroidBench [16], ICC Bench [22], our own benchmarks (https://github.com/mig40000/ICC-Benchmark) evaluating key and type matching of intent extra data, and a large set of apps from the Google Playstore. We compared our results with multiple related analysis tools. Our tool (combined with an external baseline intra-component IFC analysis) achieves perfect precision and soundness on all benchmark sets with respect to the ground truth provided, being more than on par with related IFC tools. Additionally, we demonstrate that IIFA can improve the IAC precision of other base IFC analyses with experiments, and assess the scalability of IIFA, applying it to the 90 most downloaded Playstore apps. Our experiments demonstrate that due to its compositionality IIFA's execution time scales well even to a large corpus of real-world apps. In summary, we provide the following contributions:

- *Compositional DB-backed Analysis.* We propose a modular pre-analysis approach for intent communication, in particular for analyzing inter-app communication, based on summaries for all app components containing intent senders, receivers, and the exact intent characteristics including types and keys of data transmission.
- *Novel Matching Algorithm.* We present a novel algorithm which matches intent senders with intent receivers based on these summaries and even detects flows through more than two components via a fixed-point iteration.
- *Evaluation of IIFA.* We implemented our analysis (IIFA) and evaluated it on multiple large-scale datasets. The evaluation shows that our pre-analysis approach does not negatively impact precision and recall with respect to the most relevant previous work on benchmarks, including a novel suite assessing the correct matching of intent attributes. We demonstrate that we can effectively evade combinatorial explosion, analyzing ICC/IAC information flows of the top 90 real-world apps in approximately 2.2h (excluding the baseline IFC analysis).

2 Background

2.1 Android Components

Android apps are written (mostly) in Java, but instead of defining a *main* method they consist of four component types: *Activities* are user interfaces to be interacted with. *Services* run in the background, intended for computationally expensive operations. *Broadcast receivers* register themselves to receive system or app events. *Content providers* provide data via storage mechanisms. Each app defines a manifest file (*AndroidManifest.xml*) providing essential information about the app, e.g., its components and their capabilities.

Apps are compiled to Dalvik bytecode [10], which is specialized for execution on Android. Together with additional metadata and resources Dalvik bytecode is compressed into an Android Package (APK) that can be published in market places, such as the Google Playstore. Oberheide and Miller [17] demonstrated that the security analysis on the Playstore can easily be circumvented. Even though

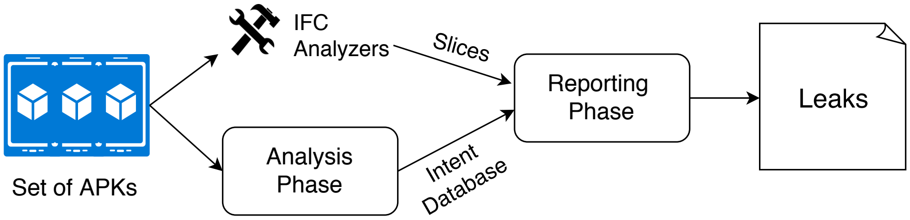

Fig. 1: Analysis Framework

Google's security mechanisms are constantly evolving, potential for malicious and vulnerable software in the Google Playstore remains.

2.2 Android Intents

Android provides a dedicated mechanism for two components to communicate. A component can send an *intent* as a message, e.g., to notify another component of an event, trigger an action of another component, or transmit information to another component. Note the universal nature of intents on Android: Intents can be sent from the system to apps (and vice versa), from one app to another (inter-app communication, IAC), or even from one component to another within the same app (intra-app-communication, ICC) [9].

Additional information can be associated with an intent: The *intent action* specifies an action supposed to be performed by the receiving component. A component declares intent actions to be received via an intent filter in the manifest file. The intent's sender is unknown on the receiver side. The *target component* mandates a specific receiver for an intent. Setting *intent extra data* adds additional information to be used as parameters by the receiving component.

Note that none of this information is mandatory. When a target component is specified an intent is called *explicit*, otherwise *implicit*. Explicit intents are delivered to the given target component only, while implicit intents can be delivered to any component with a matching intent filter. If multiple components could receive an implicit intent, the user is asked to resolve the intent manually, generally displaying a list of potential receiver apps. Li et al. [16] found that at runtime 40.1% of the intents in Google Playstore apps are explicit intents. Broadcast intents are relayed to every component registered for an intent action instead of only one of them. As intents are the universal means of inter-component communication their analysis becomes critical. In this work we propose a modular approach to precisely analyze information flow through Android intents.

3 Methodology

The fundamental problem of intent analysis for static analysis is the dynamic nature of intents. Static IFC analyses generally leverage dataflow analyses like backwards slicing to determine whether sensitive information (e.g., a device id) may flow into a sink (e.g., internet). However, if a slice contains statements where data is extracted from a received intent, it cannot determine the data's sensitivity without detailed knowledge on possible senders and their semantics.

Figure 1 presents the major building blocks of our analysis framework. In the *analysis phase* a set of APKs under inspection (e.g., all apps installed on a device) is processed and the extracted information stored into a SQL database named *IntentDB*. We collect two sets of information, app-specific information, i.e., package and class name, and registered intent filters to receive implicit intents, as well as intent sender-specific information, i.e., information required to identify potential receiver(s), key, type, and the actual data being sent.

The database is fed into the *reporting phase* together with the receiving app's information flows from a baseline (intra-component) IFC analyzer. If a flow originates at a *getXXXExtra* method[1], we consider the respective sender's outbound data as the actual data source to that flow. Remember that data can only successfully be transmitted via *put/getExtra* methods if the key parameters of both methods match and the signatures of the *put* and *get* methods correspond (e.g. the parameter type of the *put* method equals the return type of the *get* method). Thus we determine all potential senders of this intent based on matching the target component or intent action. For each of these senders we extract the *key*, *value*, and *put* signature (see section 3.1) from the database. If the key and the put signature match this *getXXXExtra* method invocation[2], we determine the sensitivity of the transmitted value based on a categorization of sources. If the value is considered sensitive, we report a potential information flow violation.

Structure of *IntentDB*: Table 1 shows an example entry (from the Telegram messenger app) of the database. As apps consist of several classes, this table has potentially multiple entries for the same app. All entries belonging to one app can be identified by the unique package name. Similarly, each class can send out several intents and hence for each intent sent we will list a separate entry (package name & class name are the same). The column "Put Signature" is considered for mapping the *put* method to the corresponding *getXXXExtra* method at the time of intent resolution. Depending on the non-empty fields, an entry in the database represents an intent receiver and/or sender. If the *Intent Filter* field is set, the app may receive intents. If either the *Target Component* or the *Intent Action* field is set, it acts as an intent sender.

Table 1: Example database for a class that can receive as well as send intents

Package Name	Class Name	Intent Filter	Target Component	Intent Action	Key	Value	Put Signature
org.telegram.messenger	Firebase-InstanceId-Service	com.google.firebase.INSTANCE_ID_EVENT	null	com.google.android.gcm.intent.SEND	”google.to”	String url = ”google.com/iid”	putExtra (String, String)

[1] *getXXXExtra* methods retrieve type-specific data from a received intent that has been added through the corresponding *putExtra* method.

[2] The *getXXXExtra*'s key is determined via backward slicing

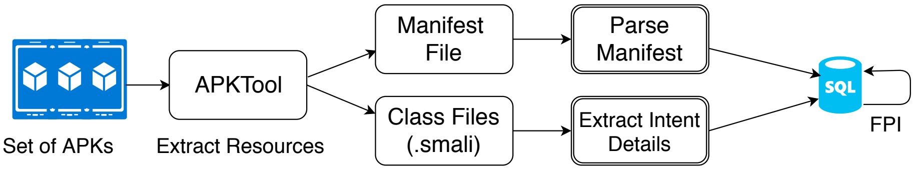

Fig. 2: Analysis Phase, FPI stands for fixed point iteration

3.1 Analysis Phase

Figure 2 depicts the workflow of the analysis phase. In the sequel, we describe the details of each component.

Apktool A set of APKs is processed by *Apktool* [1], which extracts and decodes the resources of an APK (e.g., *manifest.xml*). It decodes the Dalvik bytecode file (*classes.dex*) of the APK to more comprehensible Smali class files [7].

Manifest Parser Parsing the manifest file extracts various app details (first set of information), i.e., *package and class name*, as well as *supported intent filters*. This information is mapped to the first three columns of the table and identifies potential receivers of an intent. Even though intent receivers are typically registered in the manifest file, the *registerReceiver* method can register an intent receiver at runtime. In our experiment with 90 apps, we find 433 dynamically registered receivers ($\approx 5\%$ of all intent receivers). We scan class files for dynamically registered receivers and store them in IntentDB.

Dynamic Intent Data Extraction (Extract Intent Details) In this module, we scan each class file for methods that initiate an intent (sender methods), e.g., *startActivity*. The *Android documentation* [9] defines 25 such methods including 12 variants of *startActivity*, 11 variants of *broadcast*, *startService* and *bindService*.

- **Identifying Target Component/Intent Action** For every sender method we compute its backward slice to find the corresponding intent initialization(s). The goal is to identify its target component (for an explicit intent) or intent action (implicit intent). The intent type depends on the intent's constructor but can be altered using the *explicit-transformation* methods *makeMainActivity*, *makeRestartActivityTask*, *setClass*, *setClassName*, *setComponent*, *setPackage* or *setSelector*, which can also *change* the target component after the fact. We analyze these cases to extract the actual target: In the case of an explicit intent, we identify the name of the target component. For an implicit intent, we extract the intent action. Any app defining this intent action as supported intent filter (dynamically or in its manifest file) is a potential receiver of this intent. Unfortunately, one cannot always statically determine intent details (e.g., intent action) as they may be influenced by runtime information, which is a general limitation of static analysis. We conservatively approximate such situations, i.e., may include several potential intent actions into the database. Future work may rule out non-matching substrings of potential target name/action strings similar to reflection analyis [12].

Listing 1.1: Transit Flow

```
// APP A (OutFlowActivity)
TelephonyManager tel = (...) getSystemService(TELEPHONY_SERVICE);
String imei = tel.getDeviceId(); // source
Intent i = new Intent("action_test");
i.putExtra("data", imei);
startActivity(i); // sink
// APP B (Intermediate Activity) -- Capable of receiving "action_test"
Intent i = getIntent();
String imei = i.getStringExtra("data");
Intent newIntent = new Intent("action_test2");
newIntent.putExtra("secret", imei);
startActivity(newIntent);
//APP C (InFlow Activity) -- Capable of receiving "action_test2"
Intent i = getIntent();
String imei = i.getStringExtra("secret");
smsManager.sendTextMessage("1234567890", null, imei, null, null); // sink
```

– **Identifying Key-Value Pairs** There are several methods to associate extra data with an intent, generally leveraging *key-value* pair schemes. Senders register a value specifying the key, e.g., *Intent.putExtra(“key”, “value”)* will register the string *“value”* as data for the key *“key”*, which can be extracted by a corresponding receiver using the *Intent.getExtra(“key”)* method. Trying to receive a key with a non-matching data type results in no value being transmitted. Therefore precise analysis mandates a correct matching of *get* and *put* methods. Unlike related work [16,22] we handle the respective *put/get* method pairs for all basic data types and store the precise signature of any *put* method in *IntentDB* to consider matching types and keys when resolving values received by *getXXXExtra* methods at intent receivers.

Fixed Point Iteration Intent communication may involve more than two apps/components. In our experiments with 90 apps, we find 54 cases where more than two components were involved in a transitive information flow. In such a case, *IntentDB* contains a *getXXXExtra* method in the column *Value*. For example, in Listings 1.1, app A is sending the device id (secret data) to app B. App B forwards this data to app C, and finally app C leaks it via an SMS. The first 3 rows of Table 2 show the table *IntentDB* after analyzing all APKs. To resolve transitive flows through multiple components we perform a fixed point iteration through all entries of *IntentDB* for which *Value* contains a *getXXXExtra* method. The *com.appB* entry in Table 2 is such an example where data from a received intent is being sent out via another intent. In order to identify the received data, we determine all apps from which this component could receive the intent on which *getXXXExtra* is invoked. In our example *com.appB* receives from *com.appA*. Finally, we match the corresponding *key-value* pair through their *get-put* signatures and create a new entry, replacing the original source (*getXXXExtra* method) by the transmitted value. The created entry for our example is shown in gray in Table 2. To accommodate for modular analysis and thus potential new compatible senders, we retain the old database entry (row 2).

The reporting phase described in the next section now matches the added row with the intent received in App C to reveal the transitive information flow of sensitive data to the SMS sink.

Table 2: *IntentDB* for Listing 1.1. Fixed point iteration adds the last row

Pckg. Name	Class Name	Intent Filter	Target Component	Intent Action	Key	Value	Put Signature
com.appA	OutFlow Activity	null	null	action_test	”data”	*Device ID*	putExtra (String, String)
com.appB	Interm. Activity	action_test	null	action_test2	”secret”	getStringExtra(”data”)	putExtra (String, String)
com.appC	InFlow Activity	action_test2	null	null	null	null	null
com.appB	Interm. Activity	action_test	null	action_test2	”secret”	*Device ID*	putExtra (String, String)

3.2 Reporting Phase

In the reporting phase, we process information flows obtained by a baseline IFC analyzer together with *IntentDB*. For ICC/IAC we are only interested in flows with sources that are potential intent receivers, i.e., a *getXXXExtra* method (together with its key and signature). For every *getXXXExtra* method in a reported information flow, we extract all potential senders to this receiver from *IntentDB*, i.e., apps that use an intent with a matching target component or a matching intent action. Finally, we match *get-put* method pairs and *keys* to determine senders that actually send data to this receiver and report it as a (potential) leak if the transmitted data stems from a sensitive source[3].

For example, data flows from the *getStringExtra* method of the intent received on line 14 (Listing 1.1) to the data sink *sendTextMessage* in App C. Our analysis thus matches any sender of the intent action *action_test2* and finds two rows in IntentDB (Table 2). We check whether any of those uses the key *secret*, which both of them do. Then we match the signature of *getStringExtra* with the sender's Put Signature, where again both match. Finally, we verify if one of the potentially transmitted values (*Device ID, getStringExtra(data)*) is sensitive, thus reporting the former as an illicit information flow.

Partial Support for String and Array Access Resolution IIFA understands the Smali signature of String methods and applies partial evaluation in order to recover strings created by concatenation, substring, and other String manipulation methods. Concretely, it extracts parameters, applies the respective functionality and returns the resulting string. More contrived examples like converting a string to an array of chars (to be manipulated) are beyond the scope of our tool as we are currently not targeting obfuscated code.

4 Evaluation

We empirically evaluated our tool, IIFA, in two steps:

- *Comparative evaluation on benchmark sets.*
- *Evaluation on real-world apps from the Google Playstore.*

All experiments were performed on a MacBook Pro with a 2,9 GHz Intel Core i7 processor and 16 GB DDR3 RAM and MacOS High Sierra 10.13.1 installed. We used a version 1.8 JVM with 4 GB maximum heap size.

[3] We utilize the categorization of sources and sinks from R-Droid [3]

Table 3: Summary of Tool Results for Micro-Benchmarks

ICC Comparison							
IccTA Extension + ICC-Bench + Attribute-Mismatch-ICC-Benchmark (34 test cases)							
Precision, Recall and F1-measure	FlowDroid	AppScan	DidFail	DIALDroid	Amandroid	IccTA	Our Tool
Precision $p = \checkmark/(\checkmark + \star)$	25%	16.7%	75%	71%	62%	80%	100%
Recall $r = \checkmark/(\checkmark + \circ)$	80%	62.5%	24%	80%	60%	96%	100%
F_1-measure $2pr/(p+r)$	0.41	0.26	0.36	0.75	0.61	0.87	1
IAC Comparison							
DroidBench IAC + Attribute-Mismatch-ICC-Benchmark (10 test cases)							
Precision $p = \checkmark/(\checkmark + \star)$	0%	0%	63%	73%	52%	0%	100%
Recall $r = \checkmark/(\checkmark + \circ)$	0%	0%	21%	56%	76%	0%	100%
F_1-measure $2pr/(p+r)$	0	0	0.31	0.63	0.43	0	1

4.1 RQ1: Precision and Soundness of IIFA

Benchmark evaluation datasets Remember that IIFA is not a stand-alone tool. Therefore its intention cannot be to replace any of the related works that analyze intra-component information flows. Rather we are propagating ICC/IAC analysis as a pre-analysis, and our experiments in this section are to show that this design decision preserves the precision or soundness of the analysis results. In order to evaluate the precision and soundness, we use four separate benchmark sets and compare the results of IIFA (combined with a basic intra-component information flow analysis) to related approaches that aim at analyzing both intra-component and ICC/IAC information flows simultaneously.

- The intent-related cases of the DroidBench test suite [6] **(14 test cases)**
- The extension proposed by IccTA [16] **(18 test cases)**
- ICC-Bench, proposed by Wei et al. [22] **(9 test cases)**
- Our extension[4], Attribute-Mismatch-ICC-Benchmark, which evaluates correct matching of types and keys for data exchanged **(7 test cases)**

Note that the mentioned benchmark sets include several advanced usage scenarios of intents. An example of these scenarios is the usage of callback methods that are triggered after an event has been delivered to its target, which requires information tracking at both sender and receiver sides. Another challenge is string manipulation, e.g., of keys for intent extra data. Finally one case passes an intent with sensitive data through multiple components before finally leaking the stored data. The authors of each benchmark set provide a ground truth for each test case, which we use to measure precision and soundness.

Comparative evaluation Based on true positives (tp), false positives (fp), and false negatives (fn) we use the following metrics to compare the performance of IIFA with the related tools:

$$\begin{array}{ccc} \textbf{Precision} & \textbf{Recall} & F_1\textbf{-measure} \\ p = \frac{tp}{tp+fp} & r = \frac{tp}{tp+fn} & \frac{2pr}{p+r} \end{array}$$

We applied IIFA to the original DroidBench benchmark set, where 14 test cases are relevant for intent communication. On these benchmarks IIFA achieved

[4] https://github.com/mig40000/ICC-Benchmark

Table 4: IAC analysis support vs Android API

Tool	IAC support	API 19	>API 19
AmanDroid	✓̸	✓	–
DidFail	✓	– ↯	– ↯
AppScan	×	–	–
DroidSafe	✓̸	✓	– ↯
DIALDroid	✓	✓	↯
FlowDroid	×	–	–
IccTA	✓̸	✓	– ↯

✓ supported, × not supported
✓̸ Not supported by default,
additional configuration required

✓ supported, – fails, ↯ crashes

perfect precision and recall ratios. We further applied IIFA to the IccTA extension of Droidbench [6], ICC-Bench [22], and Attribute-Mismatch-ICC-Benchmark, and compared the results to the six most prominent tools for Android intent information flow analysis: FlowDroid [2] and AppScan [13] are limited to ICC, IccTA [16] and AmanDroid [22] require an additional tool to support IAC, DidFail [14] and DIALDroid [4] come with their own inter-app analysis. Table 3 summarizes the results of the different tools on these benchmark sets.

Matching the key and/or type of intent extra data is not reported in Li et al. [16], DidFail [14], or AmanDroid [22]. They merely create a lifecycle method that connects the sender of an intent with (the respective) receiver(s), thus creating a data flow between these components. In our experiments both IccTA and Amandroid failed to detect a key and/or type mismatch during ICC (see section 3.1) due to missing checks of these constraints, resulting in significant precision loss with respect to previous benchmark suites alone.

Our approach: IIFA We apply IIFA's pre-analysis to the same benchmarks, leveraging R-Droid [3] to generate the intra-app flows[5] as it does not interfere with our IAC/ICC model. Comparing the results with the ground truth revealed perfect precision and soundness results.

Answer to RQ1: *Our design of IIFA as a pre-analysis does not negatively impact precision and soundness, but enables precise matching not only of intent actions but also of the key and/or type of intent extra data without additional constraint solving to exclude infeasible flows.*

4.2 RQ2: Evaluating the scalability of IIFA

Evaluation on real-world apps We applied IIFA to the 90 most popular apps from the Google playstore, which arguably contain some of the most challenging apps for program analysis, e.g. due to their size. IIFA successfully analyzes each of these apps. Table 4 (extended from [20,21]) lists the related works along with their IAC analysis capabilities in general and for API levels of Android. It is important to observe that none of the tools (that support IAC analysis) is able

[5] Note that any other tool that resolves intra-component flows (in particular those of Table 3 except for DIALDroid) would also have been a possible base analysis, but may have interfered with our ICC/IAC model.

to analyze an app using API version greater than 19 (i.e. Android KitKat). As this Android version was released in 2013, it is almost impossible to find APKs amenable for analysis by these tools, rendering them obsolete for IAC analysis of realistic Android apps. While probably technically solvable with some engineering effort, we will argue in the sequel, that only pre-analysis is scalable to analyze the IAC information flows for all apps installed on a given device.

As 60% of all intents are implicit, analyzing IAC flows becomes paramount to detect hidden or accidental leaks of sensitive data contrary to user expectations. ApkCombiner was proposed to merge APKs in order to extend standard ICC analysis mechanisms (where all potential communicating components are in one APK) to IAC. As mentioned in the previous paragraph, APKCombiner fails for practically all relevant APKs. But even if merging was possible (some related work merges directly in their tools) analyzing the resulting APK faces combinatorial explosion of potential communication paths and would require additional constraint solving technologies to prune unrealizable inter-component data flow paths due to mismatching communication partners (e.g. intent action) or keys/types of the data exchanged via intend extra data.

In contrast IIFA propagates a divide-and-conquer approach where ICC/IAC communication partners are determined and constraints solved in a pre-analysis based on summary information extracted from each app in isolation. The base IFC analyses then only need to provide intra-component information flows (program slices), which is what most of these tools have originally been designed and leverage intricate optimizations for. To demonstrate the advantage of the pre-analysis approach we created a potential IAC flow from a widely used real-world app *Katwarn* to a synthetic app (written using target API level 19 to enable analysis by these tools). One of *Katwarn*'s activities (*GuardianAngelService*) shares the last known location via an implicit intent. An app that declares the corresponding intent filter (*kwrn:ga:location:update*) may receive this intent. Our synthetic app declares the *kwrn:ga:location:update* intent filter and illicitly intercepts this intent. Analyzing this IAC without IIFA failed as APKCombiner was incapable of merging these two apps, and tools that create their own paths, e.g., DIALDroid, crashed.

IIFA's scalability vs Merging-based analysis Every tool except DidFail and DIALDroid requires merging of APKs to extend their ICC capabilities to IAC. To analyze inter-app communication, *ApkCombiner* [15] was proposed in order to merge two or more apps into one. However, *ApkCombiner* supports only Android app versions published on or before 2014, and crashes with the recent apps, thus being practically unusable. But even if the merging process itself was not problematic, it would aggravate the scalability issues reported in related work [16], and confirmed in an independent recent comparative study [21]. Practically attempting to analyze a huge APK with all of the top 90 apps from the Google Playstore would lead to a combinatorial explosion of communication paths between potentially communicating components to be analyzed, precluding any precise static analysis. In particular as we found in our study with the top 90 apps from the Google Playstore that 60% of intents are implicit and thus the receiver may not be unique. Alternatively, one would have to eagerly merge all combinations of (at least) two complete apps. This is at least 8,100 combinations for our 90 apps already, most of which are not communicating. However, this approach assumes that there is no communication involving more apps than the maximum tuple size. Unfortunately there is no guarantee for this assumption unless analyzing all tuples of larger size (which does not scale anymore). In contrast, IIFA analyzes the communication compositionally based on small summaries in a database and combines only the transmitted ICC/IAC data with the intra-app flows of respective intent receivers. The average per app execution time of IIFA over 90 apps is 87.91 seconds. On average, the analysis phase took 46.81 seconds (maximum 52.20, minimum 32.40 seconds) and the reporting phase 41.10 seconds (maximum 48.60, minimum 35.10 seconds). Considering that (intra-app) static IFC analyses usually require a large amount of time to analyze real world apps, IIFA's additional cost is quite feasible for a realistic usage scenario.

Answer to RQ2: *IIFA's modular analysis avoids combinatorial explosion of the potential flow paths in a single merged APK (containing all potential communication partners), or of analyzing all tuples of a given size. Therefore, we were able to analyze the ICC/IAC flows of the 90 most downloaded real-world apps in approx. 2.2h (ignoring the time to compute intra-component slices).*

5 Related Work

Arzt et al. proposed Flowdroid [2], a static taint analysis tool that includes an extensive component lifecycle model. Flowdroid was originally designed for intra-component analysis and cannot analyze string manipulations. R-Droid [3] is an information flow analysis tool that resolves common string manipulations. It does not support intents but conservatively reports every flow to an intent sender function as a leak. AppScan [13] is a commercial tool to detect vulnerabilities in mobile and web apps, including information leaks in Android apps. However, it only supports intra-app ICC analysis and requires the source code of the inspected apps.

IccTA [16] leverages static taint analysis to analyze ICC flow leaks. It ignores some "rarely used ICC methods such as startActivities" [16], multi-threading and slightly involved string analysis, which may lead to missed information leaks. While the IccTA paper reports no experience with IAC, their GitHub page proposes the usage of APKCombiner [15], but as IccTA already has scalability issues, merging apps will aggravate this situation and may require eager combinations of all tuples of apps, resulting in combinatorial explosion. DIALDroid [4] is designed to analyze ICC and IAC flows. However, it often fails (IAC) or aborts (ICC) to analyze implicit intents. Additionally, DIALDroid is unable to detect flows involving more than two components. Wei et al. proposed Amandroid [22], which computes control and data flow graphs to resolve intents and inlines the invoked component's lifecycle. However, Amandroid ignores several sink functions as well as extra types and keys for intent data resolution. DroidSafe [11] improves intent resolution via precise points-to analysis and string resolution. As their

ICC resolution does not take extra types and keys into account their results are imprecise. Klieber et al. proposed DidFail [14] that analyzes information flow in Android applications. However, DidFail is limited to the analysis of Activities and implicit intents. Zhang et al. [23] proposed AndroidLeaker, a hybrid approach to detect intent-based privacy leaks on Android. They require instrumenting all apps under inspection, which may not be feasible due to self-integrity checks. Unlike IIFA, AndroidLeaker requires manual adaption of sources and sinks for each new Android version. Epicc [19] analyzes Android ICC precisely but focuses on ICC-related vulnerabilities. It does not fully resolve information flows [22]. ScanDroid [8] analyzes Android intents via a constraint system. However, the lack of distinction between component contexts leads to imprecise analysis results. DroidChecker [5] is a taint analysis tool for Android apps supporting intents. Due to imprecise permission handling, DroidChecker is neither sound nor complete. Further, it cannot handle dynamic features of Java, such as polymorphism. Octeau et al. [18] proposed IC3, an analysis tool for Android intents, which requires expert knowledge in the form of source code annotations.

6 Conclusion

In this work we propose a novel pre-analysis approach to analyze information flows through Android's intents. Using a database of precise intent communication summaries, IIFA avoids the combinatorial explosion of inlining all potential communication partners. We compared IIFA to six related tools on several standard and a novel benchmark sets. IIFA's precision and recall rates are on par or even better than previous tools, which demonstrates that our scalability improvements do not come at the cost of other essential analysis properties.

Acknowledgements

This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through the project SmartPriv (16KIS0760) and the German Research Foundation (DFG) via the collaborative research center ”Methods and Tools for Understanding and Controlling Privacy” (SFB 1223), project B02.

References

1. 2.0, A.: Apktool. GitHub (Jul 2017), https://ibotpeaches.github.io/Apktool/
2. Arzt, S., Rasthofer, S., Fritz, e.a.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices **49**(6), 259–269 (2014)
3. Backes, M., Bugiel, S., Derr, E., Gerling, S., Hammer, C.: R-droid: Leveraging android app analysis with static slice optimization. In: 11th ACM on ASIACCS. pp. 129–140. ACM (2016)
4. Bosu, A., Liu, F., Yao, D.D., Wang, G.: Collusive data leak and more: Large-scale threat analysis of inter-app communications. In: AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017. pp. 71–85 (2017)

5. Chan, P.P., Hui, L.C., Yiu, S.M.: Droidchecker: analyzing android applications for capability leak. In: Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks. pp. 125–136. ACM (2012)
6. Christian Fritz, S.A., Rasthofer, S.: Droid-benchmarks. https://github.com/secure-software-engineering/DroidBench, accessed: December. 2017
7. Freke, J.: Baksmali. https://github.com/JesusFreke/smali
8. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of android. Tech. rep., University of Maryland (2009)
9. Google: Android intent documentation. https://developer.android.com/reference/android/content/Intent.html, accessed: May. 2017
10. Google: Dalvik byteycode documentation. https://source.android.com/devices/tech/dalvik/dalvik-bytecode, accessed: May. 2017
11. Gordon, M.I., Kim, D., Perkins, J.H., Gilham, L., Nguyen, N., Rinard, M.C.: Information flow analysis of android applications in droidsafe. In: NDSS (2015)
12. Grech, N., Kastrinis, G., Smaragdakis, Y.: Efficient Reflection String Analysis via Graph Coloring. In: Millstein, T. (ed.) ECOOP. vol. 109, pp. 26:1–26:25 (2018)
13. IBM: Ibm security appscan source. https://www-03.ibm.com/software/products/en/appscan, accessed: May. 2017
14. Klieber, W., Flynn, L., Bhosale, A., Jia, L., Bauer, L.: Android taint flow analysis for app sets. In: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis. pp. 1–6. ACM (2014)
15. Li, L.: Apk combiner. GitHub (December 2014), https://github.com/lilicoding/ApkCombiner
16. Li, L., Bartel, A., Bissyandé, e.a.: Iccta: Detecting inter-component privacy leaks in android apps. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. pp. 280–291. IEEE Press (2015)
17. Oberheide, J., Miller, C.: Dissecting the android bouncer. SummerCon2012, New York (2012)
18. Octeau, D., Luchaup, D., Dering, M., Jha, S., McDaniel, P.: Composite constant propagation: Application to android inter-component communication analysis. In: Proceedings of the 37th International Conference on Software Engineering-Volume 1. pp. 77–88. IEEE Press (2015)
19. Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Le Traon, Y.: Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In: Proceedings of the 22nd USENIX security symposium. pp. 543–558 (2013)
20. Pauck, F., Bodden, E., Wehrheim, H.: Do android taint analysis tools keep their promises? In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2018, ACM, New York, NY, USA (2018)
21. Qiu, L., Wang, Y., Rubin, J.: Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 176–186. ACM (2018)
22. Wei, F., Roy, S., Ou, X., et al.: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1329–1341. ACM (2014)
23. Zhang, Z., Feng, X.: Androidleaker: A hybrid checker for collusive leak in android applications. In: Larsen, K.G., Sokolsky, O., Wang, J. (eds.) Dependable Software Engineering. Theories, Tools, and Applications. pp. 164–180. Springer International Publishing, Cham (2017)