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

Abhishek Tiwari, Sascha Groß, Christian Hammer {tiwari, saschagross, chrhammer}@uni-potsdam.de

University of Potsdam, Germany

December 14, 2018

Abstract

Android apps cooperate through message passing via intents. However, when apps do not have identical 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 considered static program analysis to detect dangerous data leaks due to inter-component communication (ICC) or IAC, but suffers from shortcomings with respect to precision, soundness, and scalability.

To solve these issues we propose a novel approach for static ICC/IAC analysis. We perform a fixed-point iteration of ICC/IAC summary information to precisely resolve intent communication with more than two apps involved. We integrate these results with information flows generated by a baseline (i.e. not considering intents) information flow analysis, and resolve if sensitive data is flowing (transitively) through components/apps in order to be ultimately leaked. Our main contribution is the first fully automatic sound and precise 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, which often requires simultaneously analyzing all tuples of apps.

We evaluated our tool IIFA in terms of scalability, precision, and recall. Using benchmarks we establish that precision and recall of our algorithm are considerably better than prominent state-of-the-art analyses for IAC. 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. IIFA reports 62 problematic ICC-/IAC-related information flows via two or more apps/components.

1 Introduction

Mobile devices are an attractive target for all kinds of dubious activities as they store a plenitude of sensitive data. Therefore, protecting the information stored on smartphones from unauthorized access has become imperative. Manufacturers implemented a permission system that lets the user decide which privileges an app may have, e.g., to access sensitive information or to communicate via

certain channels. However, permissions cannot restrict information flow once access to sensitive information or a communication channel has been granted. Given that the number of available Android apps is enormous and that market places are lacking thorough security checks before publishing an app¹, several cases have been reported where vulnerable or malicious apps leak sensitive information [32, 30].

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. Information flow is not restricted to a single component, but occurs frequently between components of the same [20, 12] and even different apps [30]. 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 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 [20, 30] but suffer from multiple shortcomings: Both approaches basically inline a synthetic "main" method that models the lifecycle of the receiving component/app into the sender of the intent. However, in order to match senders and receivers they merely verify that the intent action (or similar receiver-identifying data) matches. Neither of those two approaches actually determines whether the receiver and the sender use the same type or key in the key-value communication scheme of extra data transmitted via intents. Thus, either some or even all receivers in the inlined lifecycle might not be eligible to read the transmitted data, which can thus result in many spuriously reported data leaks. A more accurate matching could probably be added to these approaches, but only if there is merely one sender and one receiver statement in any given pair of communicating components. In case of multiple sender and/or receiver statements one matching pair might still induce a quadratic number of spurious flows.

Inlining several sender and receiver apps into one huge app to analyze (e.g. 90 apps like in our evaluation study) requires a very precise analysis in order not to magnify the imprecision described in the last paragraph (e.g., context-sensitivity to match multiple senders to the same component lifecycle or even that one sender is only problematic if it has received sensitive intent data from another app). Besides, inlining several apps into one raises scalability issues as even single app ICC is challenging in terms of scalability [20]. Therefore the straight forward would be to eagerly analyze all pairs of apps. Note that at least a quadratic number of combinations must be analyzed to include the effects of IAC to IFC. Realistically, intent communication can involve more than two apps, further aggravating the combinatorial explosion of merging-based approaches. Besides, merging itself is impractical in two dimensions: Merging APKs or other internal data structures does not scale to realistic apps in our experience, and even if it does, the complexity to analyze the merged app inflates, but as most

¹Even Google's official security analysis has been circumvented [24]

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.

Finally, related work suffers from requiring access to source code (or even source code annotations) [15, 14, 4, 25], lacking support for string analysis [2, 20], and lacking support for certain sink functions [2, 20, 15, 17]. All of these drawbacks lead to insufficient precision and soundness issues when these analyses are applied to real-world apps.

Our Contributions. In this work we propose a novel information-flow analysis that evades combinatorial explosion of potential communication partners, while precisely matching type and key information of intent data. Our approach can predict which combinations of apps communicate by separating analysis of apps and matching of communication partners. In a first step we create a database of summary information about senders of intents, their characteristics including types and keys, and outbound intent data, as well as apps registered to receive certain implicit intents. This information can then be matched in a subsequent step to identify potential communication partners. Further we propose a novel matching algorithm, based on a baseline IFC analysis providing potential intra-app flows (including a program slices of the receiver's key value) for all potential intent receivers. We use senders' outbound intent data as input to the information flows identified in respective receivers, which eliminates 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 light-weight fixed point iteration through the DB information. Note 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 compute precise communication paths between components handling complex control flows such as in callback methods (see section 4.2.1). As intent targets are specified via a string parameter, our approach can resolve common string manipulations, which improves our precision significantly for regular (non-obfuscated) apps. As a minor contribution we took great effort to handle the full spectrum of intent communication, supporting explicit and implicit intents as well as dynamically created intent receivers. All previous approaches miss at least one of these features, leading to unsoundness and imprecision. Our analysis is fully automated and does not require the source code of the app under analysis. We aim to answer the following research questions:

- RQ1: How are the precision and soundness of our approach with respect to the state of the art analyses?
- RQ2: Does our approach scale to a realistic corpus of real-world apps?
- RQ3: How do common real-world apps communicate through IAC?

We implemented our approach as a tool called *IIFA* and evaluated it on DroidBench, the IccTA extension of DroidBench, ICC Bench, 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, being more than on par with related IFC tools that consider intent communication. Additionally, we demonstrate the superior IAC precision with experiments and an evaluation applying IIFA to the 90 most downloaded Playstore apps. Further, our experiments demonstrate that due to its compositionality IIFA's execution time scales well even to large real-world apps. In summary, we provide the following contributions:

- Compositional DB-backed Analysis. We propose a modular analysis approach for intent communication based on summaries containing intent senders, receivers, and the exact intent characteristics including types and keys of data transmission. To that end, we model all publicly known intent-based communication schemes precisely.
- Novel Matching Algorithm. We present a novel algorithm which matches intent senders with intent receivers based on the summaries, and subsumes transmitted data into the receiver's intra-component information flows to report potential dangerous inter-component and -app information flows. This matching requires no eager pairwise analysis but only investigates potential communication partners. Thus each app is only analyzed once by IIFA and potentially as well by an intra-app IFC analysis.
- Evaluation of IIFA. We implemented our analysis (IIFA) and evaluated it on multiple large-scale datasets. The evaluation shows that our analysis is on par or better than the most relevant previous work in terms of precision and recall with respect to previously presented benchmarks. We demonstrate our improved ICC precision with IAC benchmarks and a real-world evaluation analyzing the top 90 real-world apps. Finally we demonstrate that we can effectively evade combinatorial explosion analyzing these apps in less than two minutes per app (in addition to the traditional IFC analysis) and identifying 62 potentially dangerous information flows through ICC.
- *IAC study*. We performed a study regarding the use of ICC/IAC in Android and present our results outlining IAC patterns.

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., components and their capabilities are defined in the manifest file.

Apps are compiled to Dalvik bytecode [11], 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 [24] demonstrated that the security analysis on the Playstore can easily be circumvented. Even though 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) [10].

Additional information can be associated with an intent: The *intent action* specifies an action supposed to be performed by the receiving component. A component can register to receive intents with a specific intent action by declaring an intent filter in the manifest file (see discussion of Listing 2 below). 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.

It is important to 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. [20] 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.

Listing 2: App B - Receiver: AndroidManifest.xml

```
1 <intent-filter>
   <action android:name="CUSTOM_INTENT.ACTION"/>
   <category android:name = "android.intent.category.DEFAULT" />
4 </intent-filter>
                 Listing 3: App B - Receiver: InFlowActivity
```

```
1 Intent i = getIntent();
2 String imei = i.getStringExtra("data");
3 smsManager.sendTextMessage("1234567890", null, imei, null, null); // sink
```

3 Motivation

In this section we will describe an example workflow of intents. We will then discuss the inefficiency of the current state-of-the-art analysis tools. In Listing 1, App A initiates IAC in the OutFlowActivity class. An implicit intent i is created (line 3) with the intent action "CUSTOM_INTENT.ACTION". The putExtra method (line 4) associates additional data from the variable imei with the key "data" in this intent. The device id stored in variable imei (lines 1 and 2) is sensitive data, as the device can be uniquely identified by this number. The intent is finally triggered via a startActivity call such that it can be received by registered receivers. On the receiver side the intent extra data can be extracted by the receiver and (ab-)used in any way permissible to that app.

Listing 2 and 3 show the code snippets of an example receiver for the above intent. In the manifest file (Listing 2) the receiver declares its capability to support intent filter ("CUSTOM_INTENT.ACTION"). Listing 3 extracts the received intent data corresponding to the key "data" via a getStringExtra method call. This data flows to a data sink (line 3) where it is being leaked off the device. Since the received data is a sensitive information with respect to App A. analysis tools should report this as a potential data leak. Observe that these two apps must be related by an analysis in order to identify the leak and that the precision for determining intent data crucially influences the precision to determine related apps and which data is transmitted.

Current analysis tools for ICC [20]/IAC [30] only match the identifying string (like the intent action in Listing 1-3) and ignore the key "data" or the type (in our case String, see line 2 of the receiver), which must also match for data to be transmitted. If multiple receivers are present in one component then adding checks for these conditions is non-trivial as these approaches basically inline the receiver into the sender, thus the same intent would be used for all receivers even if the key or type of the getExtra-method differs, resulting in spurious reported information flows. Simply merging/inlining all apps installed on a device is prohibitively expensive, and may result in impossible flows created as a result of the merging process, thus in practice these approaches may resort to eagerly inlining pairs, triples, ... of communicating apps, leading to combinatorial explosion of analysis targets. Furthermore, in a realistic scenario apps may be installed on a device at any time. Thus, whenever a new app (version) arrives, these tools need to join apps again.

To overcome these limitations, our analysis is designed in a modular way: It remembers summaries of the intent characteristics (including key and type) of

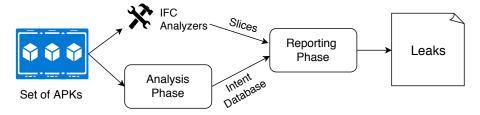


Figure 1: Analysis Framework

each app in a database and applies this knowledge to (intra-app) information flows determined in receiving apps. Due to the database our analysis can recursively resolve dependences when more than two apps are involved in intent communication.

4 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 at 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 method², 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 value's 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 4.1) from database. If the key and the put signature match this getXXXExtra method invocation³, 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.

As an example, consider Listings 1 and 3 again: The *sendTextMessage* (line 3) receives data from the *getStringExtra* method (line 2). Therefore we scan our database for potential intent senders of App B's received intent

 $^{^{2}}$ getXXXExtra methods retrieve type-specific data from a received intent that has been added through the corresponding putXXXExtra method.

³The getXXXExtra's key is determined via backward slicing

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

Package Name	Class Name	Intent Filter	Target Compo- nent	Intent Action	Key	Value	Put Signatur	е
org.telegram messenger	Firebase In- stanceId Service	com.google.firebase.IN- STANCE_ID_EVENT	null	com.google.android. gcm.intent.SEND	"google.to"	String url = "google.com/iid"	putExtra (S String)	string,

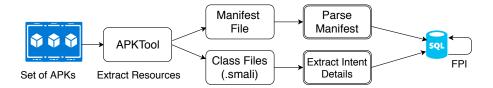


Figure 2: Analysis Phase, FPI stands for fixed point iteration

(line 1): App A sends an implicit intent with matching intent action ("CUS-TOM_INTENT.ACTION", line 3 of Listing 1). The signature of the putExtra method (line 4) has a String parameter, which matches the return type of the getStringExtra method of the receiver, and the keys of the sender and receiver ("data") match. Thus, the source of the transmitted value (i.e. the IMEI of the device) is considered the information source of the flow to the SMS transmission in App B. As the IMEI is sensitive information, an information flow violation is reported.

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.

4.1 Analysis Phase

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

4.1.1 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 [8].

4.1.2 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.

4.1.3 Dynamic Intent Data Extraction

In this module, we scan each class file for methods that initiate an intent (sender methods), e.g., startActivity. The $Android\ documentation\ [10]$ 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 and trace until we 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 make-MainActivity, 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 analysis [13].

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("Test-Key", "Test-Value") will register the string "Test-Value" as data for the key "Test-Key", which can be extracted by a corresponding receiver using the Intent.getExtra("Test-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 [20, 30] 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.

4.1.4 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 4, app A

```
1 // APP A (OutFlowActivity)
2 TelephonyManager tel = (TelephonyManager)
      getSystemService(TELEPHONY_SERVICE);
3 String imei = tel.getDeviceId(); // source
4 Intent i = new Intent("action_test");
5 i.putExtra("data", imei);
6 startActivity(i); // sink
8 // APP B (Intermediate Activity) -- Capable of receiving "action_test"
9 Intent i = getIntent();
10 String imei = i.getStringExtra("data");
11 Intent newIntent = new Intent("action_test2");
12 newIntent.putExtra("secret", imei);
13 startActivity(newIntent);
14
15 //APP C (InFlow Activity) -- Capable of receiving "action_test2"
16 Intent i = getIntent();
17 String imei = i.getStringExtra("secret");
18 smsManager.sendTextMessage("1234567890", null, imei, null, null); // sink
```

Table 2: IntentDB for Listing 4. 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)

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 prior to fixed point iteration. To resolve transitive flows through multiple components we iterate in a fixed point iteration through the 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 qetXXXExtra is invoked. In our example com.appB receives from com.appA. Finally we match the corresponding key-value pair through their qet-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.

4.2 Reporting Phase

In the reporting phase, we process information flows obtained by a baseline IFC analyzer together with the *IntentDB* from the analysis phase. 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

```
public class OutFlowActivity extends Activity{
    protected void onCreate(Bundle savedInstanceState) { // ...
     TelephonyManager tel = (TelephonyManager)
3
          getSystemService(TELEPHONY_SERVICE);
     String imei = tel.getDeviceId(); // source
     Intent i = new Intent(this, InFlowActivity.class);
     i.putExtra("data", imei);
     startActivityForResult(i, 1);
    protected void onActivityResult(int requCd, int resCd, Intent data) {
9
       String imei = data.getStringExtra("data");
10
       smsManager.sendTextMessage("1234567890", null, imei, null, null);
11
           // sink
12 }}
```

Listing 6: Receiver: InFlowActivity

```
public class InFlowActivity extends Activity {
   protected void onCreate(Bundle savedInstanceState) { // ...
   Intent i = getIntent();
   setResult(1, i);
   finish();
}
```

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 source⁴.

For example, data flows from the getStringExtra method of the intent received on line 16 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.

4.2.1 Handling of startActivityForResult and bindService

startActivityForResult is a special case of intent communication illustrated via a code snippet of an activity in Listing 5 (adapted from [6]). OutFlowActivity (line 5) creates an explicit intent with InFlowActivity as the target component. This intent is provided extra data imei (line 6), containing the actual IMEI of the device (lines 3, 4). startActivityForResult triggers this intent (line 7) with a second argument that is a request code identifying this request. Listing 6 contains the code snippet for the activity InFlowActivity, receiving this intent (line 3). The setResult method (line 4) returns the received intent with the same request code. Upon successful creation of InFlowActivity control returns to the onActivityResult

⁴We utilize the categorization of sources and sinks from R-Droid [3]

(line 9 of listing 5) method of *OutFlowActivity*. The third parameter (*data*) of this method corresponds to the intent returned via the *setResult* method of *InFlowActivity*. This intent, originally sent by *OutFlowActivity*, still contains the secret *IMEI* of the device. The *IMEI* is extracted (line 10) and leaked (line 11) via a text message. Thus data flows from the sender to the receiver and back, as modeled in our information flow analysis.

Similarly, after a Service has been successfully bound via bindService, control will return to an onServiceConnected method (of an object designated as the second parameter of the original bindService call.) This method is being passed an argument from the service intent receiver, and thus data can be returned to the intent sender. In order to identify such flows soundly and precisely our analysis models the data flow according to these patterns at both sides of the communication.

4.3 Domain Knowledge for Java String Class and List Analysis

As the ability to precisely determine intent senders and receivers depends significantly on the ability to identify the String values of target components or intent actions, we enrich IIFA with domain knowledge on the Java String class. 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. Due to our modular design we could also add more expensive analyses like SMT-solvers that handle more cases. However, there are always undecidable cases like encrypted strings or dynamic input.

Similarly, we encode domain knowledge on the API of *LinkedList* to be able to extract list entries from a given index they were stored in. Again, a more precise model of Lists improves analysis precision but in general this problem is undecidable. Other collections could be modeled analogously, which we are planning as future work.

5 Evaluation

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

- Comparative evaluation on benchmark sets. We applied IIFA to three standard evaluation sets for intent communication comprising 41 test cases with ground truth results for each test. We compared the precision and soundness of IIFA to 5 state of the art tools that support intent analysis.
- Evaluation on real-world apps from the Google Playstore. We applied IIFA to the 90 most popular apps from the Google Playstore in order to evaluate its scalability on real-world apps.

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 $1.8~\mathrm{JVM}$ with 4 GB maximum heap size.

5.1 RQ1: Precision and Soundness of IIFA

5.1.1 Benchmark evaluation datasets

In order to evaluate whether the precision and soundness of our approach are on par with state of the art analyses, we use three separate benchmark sets to compare the results of our tool to related approaches. However, as IIFA is not a stand-alone IFC tool but rather only models the information flows through intents, we restrict ourselves to the subset of the three benchmark sets that test the results of intent-based communication:

- The intent-related cases of the original DroidBench test suite [6] (14 test cases)
- The extension proposed by IccTA [20] (18 test cases)
- ICC-Bench, proposed by Wei et al. [30] (9 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 (see Section 4.2.1). 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 ground truth for each test case, which we use to measure precision and soundness.

5.1.2 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:

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

We applied IIFA to the original DroidBench benchmark set, where 14 test cases are relevant for intent communication. On these benchmarks IIFA achieved precision and recall ratios of 100%. Given that the DroidBench tests cover almost identical cases as those in IccTA and ICC-Bench, we do not list details on the results for space reasons. We further applied IIFA to the IccTA extension of Droidbench [6] and ICC-Bench [30], and compared the results to the five most prominent tools for Android intent information flow analysis: FlowDroid [2], AppScan [15] and IccTA [20] are limited to ICC, DidFail [17] and Amandroid [30] come with their own inter-app analysis. Table 3 displays the results of the different tools on both benchmark sets (related work taken from Li et al. [20]), which also summarizes the metrics. In sequel, we compare the results of Amandroid, IccTa and IIFA (Table 3).

Amandroid Amandroid has a mediocre recall rate of 60% due to imprecision in complex sender functions (section 4.2.1), imprecision in the lifecycle model of Services (*startService2*) and string manipulation (*DynRegister2*) and lacking support for Content Providers. Its IAC capabilities are not tested in any benchmark.

IccTA IccTA in general has good precision and recall ratios, but fails on test cases that include complex string manipulations, e.g., *DynRegister2* and *startActivity7*.

In our own experiments IccTA and Amandroid failed to resolve the key and/or type of ICC (see section 4.1.3), confirming the fact that this is not mentioned in their publications.

Our approach: IIFA To evaluate the precision and soundness of IIFA we applied it to the benchmark sets. We used R-Droid [3], a static IFC tool, to generate the intra-app flows, and compared our analysis results with the provided ground truth. We also verified the analysis results manually. As expected it also resolved our own tests with keys and types correctly.

Answer to RQ1: Our tool IIFA is on par with (or even outperforms) all related state-of-the-art tools on the first three benchmark sets, and resolves our own experiments while the most precise related works suffer from lacking type/key matching.

Clearly these benchmarks, gathered from related work, do not involve typically ignored (as hard to analyze) language features like reflection, or native code. However, they do cover a range of difficult features with respect to intent analysis, like dynamically composed strings, arrays with both sensitive and insensitive data, etc. The intention of this research question was to assert that the scalability gains (see RQ2) do not negatively impact other essential properties of our analysis, and to demonstrate the impact of precise matching of types and keys for transmitted intent data. Even though small, this microbenchmark evaluation demonstrates that we are more than on par with the best related work.

5.2 RQ2: Evaluating the scalability of IIFA

5.2.1 Evaluation on real-world apps

We applied IIFA to real-world apps to assess whether the analysis scales to a realistic corpus of large real-world apps. We therefore downloaded the 90 most popular apps from the Google playstore. IIFA successfully analyzes each of these apps. We compared the scalability of IIFA with the most related approach IccTA [20] on real world apps. We chose IccTA as it achieves significantly better precision and soundness numbers in the benchmark evaluation, as Amandroid's median runtime and IccTA's are similar [20], and as most other tools do not support ICC analysis for more than two components. To analyze inter-app communication, IccTA's GitHub page proposes ApkCombiner [19], which merges two or more apps into one. However, in our experiments ApkCombiner only worked well with small numbers of benchmark set apps. When applied to numerous pairs of real world apps, it failed with an error message. But even if the joining process was not problematic, it would aggravate the single-app scalability issues [20], which were confirmed in a separate recent comparative study [27], and would therefore lead to combinatorial explosion of apps to be analyzed. In our study with the top 90 apps from Google Play Store, we find that 60% of all intents are implicit. IccTA 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.

In contrast, IIFA analyzes the communication compositionally based on small summaries in a database and combines only the transmitted ICC/IAC data with

Table 3: Comparison results. ✓: true positive, ★: false positive, o: false negative

		IccTA	<u> </u>				
T	Explicit	Flow	App	Did	Aman	IccTA	Our
Test Case	ICC	Droid	Scan	Fail	droid	1cc 1 A	Tool
startActivity1	T	√ ★	√ ★	0	√	√	√
startActivity2	Т	√ (4 ★)	√ (4 ★)	0	√	√	√
startActivity3	Т	√(32 ★)	√ (32 ★)	0	✓	√	✓
startActivity4	F	* *	* *	*			
startActivity5	F	* *	* *	*			
startActivity6	Т	* *	* *		*		
startActivity7	Т	* *	* *		*	*	
startActivityForResult1	T	√	√	0	√	√	√
startActivityForResult2	Т	√	0	0	0	√	<u>√</u>
startActivityForResult3	T	√ ★	0	0	o *	√	\checkmark
startActivityForResult4	Т	√ √ *	✓ 0	0 0	√ o *	✓ ✓	√ ✓
startService1	Т	√ ★	√ ★	0	√	√	√
startService2	Т	√ ★	√ ★	0	0	√	√
bindService1	Т	√ ★	√ ★	0	0	√	√
bindService2	Т	0	0	0	0	✓	\checkmark
bindService3	Т	0	0	0	0	√	√
bindService4	Т	√ * ∘	√ ★ ○	0 0	0 0	✓ ✓	√ ✓
sendBroadcast1	F	√ ★	√ ★	0	√	√	√
	,	ICC-Bei	nch				
Explicit1	Т	\checkmark	-	0	✓	✓	\checkmark
Implicit1	F	✓	-	√	√	√	✓
Implicit2	F	√	-	√	√	√	√
Implicit3	F	√	-	√	√	√	√
Implicit4	F	√	-	√	√	√	√
Implicit5	F	√ ★	-	√	√	√	√
Implicit6	F	✓	-	✓	√	√	\checkmark
DynRegister1	F	0	-	0	√	√	√
DynRegister2	F	0	-	0	0	0	✓
	Sum, Pr		ecall and				
✓, higher is better	20	10	6	15	24	25	
⋆, lower is better	53	50	2	4	1	0	
o, lower is better	5	6	19	10	1	0	
Precision $p = \checkmark/(\checkmark + \star)$	27.3%	16.7%	75%	78.9%	96%	100%	
Recall $r = \sqrt{/(\sqrt{+\circ})}$	80%	62.5%	24%	60%	96%	100%	
F ₁ -measure $2pr/(p+r)$	0.41	0.26	0.36	0.68	0.96	1	

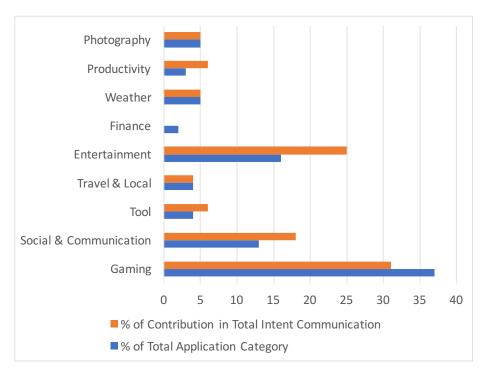


Figure 3: Intent Usage

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. In our experiments with related tools such as IccTA, R-Droid, and AmanDroid, information flow analysis of a real-world app from our test set consistently took more than 30 minutes. Let us assume t_{IFC} is the total time taken by an IFC analyzer to analyze one app, which according to our experiments is at least 30 minutes. IIFA adds approximately 1.5 minute per app. We refer to this sum of t_{IFC} and the time taken by IIFA as t_{sum} . Thus, the total time taken by IIFA for 90 apps is $90 \times t_{sum} \ (\approx 2 \text{ days})$. Whereas, IccTA takes $90 \times 90 \times t_{IFC}$ (\approx half a year) to analyze ICC between any two apps, let alone tuples of greater size. Clearly, the total time is dominated by the combinatorial explosion of merging apps. Therefore, IIFA would take at most 1.2% of the time taken by IccTA.

Answer to RQ2: IIFA analysis avoids combinatorial explosion and, in an analysis of the 90 most downloaded real-world apps, is thus orders of magnitude faster than combining the bytecode programs eagerly.

5.3 RQ3: Evaluating IAC in real-world apps

In total we identified 10,669 calls to start an intent (either as an Activity, Broadcast or Service), 76% of these leverage startActivity/startActivityForResult. Further, the data in IntentDB show that (statically) 60% of the intents are implicit. The Android documentation defines 42 different types of getXXXExtra methods. IIFA determines that 54.6% of these methods retrieve String values, either via Bundle.getString(String) (38.8%) or String.getStringExtra(String) (15.7%). For 2% of the sent intents IIFA was unable resolve the target component or intent action and conservatively approximated it to either including multiple actions (0.6%) or even a dummy action (1.4%), which requires manual inspection to resolve the potential strings (e.g. due to dynamic input from a file).

Figure 3 provides a categorization of the 90 apps along with their intent usage. 37% of the apps are *Games*, which contribute the most to intent communications (31%). Interestingly, we find that 6% of the total intent communication is triggered by a single a *Communication* app, whatsApp. It contributes to 35% of the intent communication in the *Communication* category.

IIFA's database contains senders with 380 distinct Intent actions, 100 out of them with corresponding receivers in our test set (excluding system apps and OS). Figure 4 displays these 100 intent actions and their numbers of senders and receivers. The second graph zooms in on the ten most widely used intent actions. Note that the numbers of actual intent receivers is lower due to type and key matching of intent extra data, demonstrating the precision of our analysis. android.intent.action.VIEW is most widely used (73 senders and 52 receivers) followed by android.intent.action.SEND (50 senders and 15 receivers). android.intent.action.DIAL is an Intent action to invoke the OS phone dialer. As Viber, a voip app, also registers to receives it, users will be asked to select how to make a phone call. Other apps not providing voip ought not receive it. IIFA determines potential rogue apps via a simple database lookup. To the best of our knowledge, we are the first to predict and analyze these communication patterns.

IIFA detects 62 ICC-based information flows from sensitive sources among the 90 apps. Our results shows that even widely used apps share sensitive information via implicit intents, which may lead to intent interception and intent hijacking attacks [23]. We manually validated these claims in the following apps: Katwarn provides hazard and disaster warnings and has been downloaded over one million times. IIFA finds that one of it's activities (Guardian Angel Service) shares the last known location via an implicit intent. An app that registers the respective intent filter can try to intercept this intent (intent interception) to obtain the location details without having permission to access the device's location. Similarly, the shopping app ebay (via activity EventItemsFragment) and the location & travel app Google Earth share internal device resources via an implicit intent, which are thus also prone to intent interception. IIFA also determines that ebay shares sensitive tags via both implicit (in CheckoutActivity activity) and explicit intents (in PaypalCreditPromotionsActivity activity). While tags do not always contain sensitive information, they may contain sensitive objects. For example, a tag can contain a url, e.g., BankA.com, to be opened by an intent created by the receiver. If this tag is shared via an implicit intent, a malicious app that registers the respective intent filter may manipulate this tag to open another url, e.g., BankB.com (intent hijacking as part of a phishing

attack).

Answer to RQ3: IIFA analyzes the IAC patterns and may detect rogue apps registering for Intent actions they are not supposed to handle. We detect a number of problematic information flows that may be abused by malevolent apps that have not been reported previously.

5.4 Evaluation Summary and Discussion

We empirically evaluated IIFA in two steps. IIFA does not suffer from combinatorial explosion (RQ2) as it analyzes each app once (with potentially one additional intra-app IFC analysis by an external tool), and its precision and recall are at least on par with related work.

Due to the nature of our analysis we rely on program slices generated from a baseline IFC analyzer. Therefore our combined analysis as presented in the evaluation section inherits all advantages and disadvantages as well as potential implementation bugs of the underlying analysis. In order to rule out any potential interference of our analysis with the baseline analysis we chose R-Droid as our baseline analysis as it is relatively precise but does not attempt to resolve intent communication on its own. At the same time as hardly any analysis tool considers the grand obstacles for static program analysis like native code and reflection, our combined analysis also lacks these features. Further, we concentrate on intent-based communication in this work and ignore other, more atypical forms of inter-component communication such as static (global) variables or content providers. Some baseline analyses support those, in which case a combination of IIFA with such a tool would also do.

6 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 supporting multi-threading and AsyncTasks and resolves common string manipulations for information flow purposes. R-Droid does not support intents but conservatively reports every flow to an intent sender function as a leak.

AppScan [15] 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. Further, AppScan requires the source code of the inspected apps.

IccTA [20] leverages static taint analysis to analyze ICC flow leaks. It ignores some "rarely used ICC methods such as startActivities" [20], 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 [19], but as IccTA already reports scalability issues, merging apps will aggravate this situation and may require eager combinations of all tuples of apps, resulting in combinatorial explosion.

Wei et al. proposed Amandroid [30], which computes control and data flow graphs to resolve intents and inlines the invoked component's lifecycle. However,

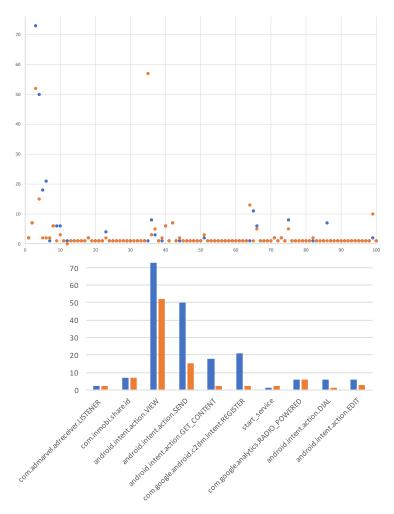


Figure 4: Intent Actions (x-axis) per sender (blue) and receiver (orange, y-axis)

Amandroid ignores several sink functions, and like IccTA, types and keys for intent data resolution.

DroidSafe [12] improves intent resolution via precise points-to analysis and string resolution. The authors claim an extension for IAC but only hit at the implementation strategy and no experiments with IAC are reported. As their ICC resolution does not take types and keys into account results are imprecise.

Klieber et al. proposed DidFail [17], that analyzes information flow in Android applications. However, DidFail is limited to the analysis of Activities and implicit intents.

Zhang et al. [31] propose a hybrid approach to detect intent-based privacy leaks on Android. They leverage dynamic analysis to detect ICC, which may thus miss ICCs based on coverage. 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 [26] analyzes Android ICC precisely but focuses on ICC-related vulner-

abilities. It does not fully resolve information flows [30]. Jiang and Xuxian [16] proposed ContentScope, which detects integrity and confidentiality vulnerabilities based on dataflow analysis, but is limited to Content Providers. Li et al. [21] proposed PCLeaks, a data-flow analysis detecting information leaks and component hijacking in Android applications. PCLeaks cannot handle several ICC sink methods such as *startActivities* and multi-threaded programs.

Hunang et al. [14] proposed a type system to prevent data disclosure. Unlike our approach, they require annotations in the source code, which may be a burden to app developers. Barros et al. [4] developed a type system to precisely resolve intents and reflections, which requires annotated source code. They inherit the imperfections of their underlying framework Epicc [26]. ScanDroid [9] 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. Enck et al. [7] proposed TaintDroid, a dynamic taint-analysis tool that supports intents. As a dynamic approach, finding leaks of sensitive data in vulnerable apps depends on coverage. TaintDroid cannot differentiate between different sources of sensitive data.

Octeau et al. [25] proposed IC3, an analysis tool for Android intents, which requires a specification of the intended program behavior written in a declarative language COAL. However, writing a COAL specification requires source code access and expert knowledge. Liu et al. [22] proposed MR-Droid, which generates an information flow graph and computes risk scores for various vulnerabilities. However, it does not natively support groups of more than two applications and misses various details of creating ICC links. DroidDisintegrator by Schuster et al. [28] applied dynamic analysis using a device emulator. This emulator monitors ICC, generates policies and enforces them directly on the app. However, it is limited to intra-application information flow and is prone to false positives and negatives. Lee et al. [18] proposed SEALANT, a hybrid approach for information flow control that aims to intercept unwanted messages at runtime. It only supports one level of taint and thus is not able to distinguish between various sources sensitive information. Shen et al. [29] proposed an approach for information flow control that requires modifications to the operating system, adding additional flow permissions.

7 Conclusion

In this work we propose a novel approach to analyze information flows through Android's intents. We create a database of precise intent communication summaries and match values transmitted to the receivers based on their intra-app flows. Thus, IIFA avoids the combinatorial explosion of merging all potential communication partners. We implemented our approach in a tool called IIFA and compared it to five related tools on two standard 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. Finally we applied IIFA to the 90 most popular apps from the Google Playstore and showed that due to our compositional

approach the runtime performance of IIFA is moderate, scales to large real-world apps, and detects precise communication patterns and illicit IAC flows.

References

- [1] Apache 2.0. Apktool. GitHub, Jul 2017. URL: https://ibotpeaches.github.io/Apktool/ [cited 29 Jul 2017].
- [2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick Mc-Daniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycleaware taint analysis for android apps. Acm Sigplan Notices, 49(6):259–269, 2014.
- [3] Michael Backes, Sven Bugiel, Erik Derr, Sebastian Gerling, and Christian Hammer. R-droid: Leveraging android app analysis with static slice optimization. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 129–140. ACM, 2016.
- [4] Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Michael D Ernst, et al. Static analysis of implicit control flow: Resolving java reflection and android intents (t). In Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, pages 669–679. IEEE, 2015.
- [5] Patrick PF Chan, Lucas CK Hui, and Siu-Ming Yiu. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 125–136. ACM, 2012.
- [6] Steven Arzt Christian Fritz and Siegfried Rasthofer. Droid-benchmarks. https://github.com/secure-software-engineering/DroidBench. Accessed: December. 2017.
- [7] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems* (TOCS), 32(2):5, 2014.
- [8] Jesus Freke. Baksmali. https://github.com/JesusFreke/smali.
- [9] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. Scandroid: Automated security certification of android. Technical report, University of Maryland, 2009.
- [10] Google. Android intent documentation. https://developer.android. com/reference/android/content/Intent.html. Accessed: May. 2017.
- [11] Google. Dalvik byteycode documentation. https://source.android.com/devices/tech/dalvik/dalvik-bytecode. Accessed: May. 2017.

- [12] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In NDSS. Citeseer, 2015.
- [13] Neville Grech, George Kastrinis, and Yannis Smaragdakis. Efficient Reflection String Analysis via Graph Coloring. In Todd Millstein, editor, 32nd European Conference on Object-Oriented Programming (ECOOP 2018), volume 109 of Leibniz International Proceedings in Informatics (LIPIcs), pages 26:1–26:25, Dagstuhl, Germany, 2018. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2018/9231, doi:10.4230/LIPIcs.ECOOP.2018.26.
- [14] Jianjun Huang, Xiangyu Zhang, and Lin Tan. Detecting sensitive data disclosure via bi-directional text correlation analysis. In *Proceedings of the* 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 169–180. ACM, 2016.
- [15] IBM. Ibm security appscan source. https://www-03.ibm.com/software/ products/en/appscan. Accessed: May. 2017.
- [16] Yajin Zhou Xuxian Jiang and Z Xuxian. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS)*, 2013.
- [17] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 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*, pages 1–6. ACM, 2014.
- [18] Youn Kyu Lee, Gholamreza Safi, Arman Shahbazian, Yixue Zhao, Nenad Medvidovic, et al. A sealant for inter-app security holes in android. In Proceedings of the 39th International Conference on Software Engineering, pages 312–323. IEEE Press, 2017.
- [19] Li Li. Apk combiner. GitHub, December 2014. URL: https://github.com/lilicoding/ApkCombiner.
- [20] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In Proceedings of the 37th International Conference on Software Engineering-Volume 1, pages 280–291. IEEE Press, 2015.
- [21] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2014 IEEE 13th International Conference on*, pages 388–397. IEEE, 2014.
- [22] Fang Liu, Haipeng Cai, Gang Wang, Danfeng Daphne Yao, Karim O Elish, and Barbara G Ryder. Mr-droid: A scalable and prioritized analysis of inter-app communication risks. Proc. of MoST, 2017.

- [23] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.
- [24] Jon Oberheide and Charlie Miller. Dissecting the android bouncer. SummerCon2012, New York, 2012.
- [25] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite constant propagation: Application to android intercomponent communication analysis. In Proceedings of the 37th International Conference on Software Engineering-Volume 1, pages 77–88. IEEE Press, 2015.
- [26] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In Proceedings of the 22nd USENIX security symposium, pages 543–558, 2013.
- [27] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 176–186. ACM, 2018.
- [28] Roei Schuster and Eran Tromer. Droiddisintegrator: intra-application information flow control in android apps. In ACM Asia Conference on Computer and Communications Security (ASIACCS) to appear, 2016.
- [29] Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora, Babu Dhandapani, Eric John Lehner, Steven Y Ko, and Lukasz Ziarek. Information flows as a permission mechanism. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pages 515–526. ACM, 2014.
- [30] Fengguo Wei, Sankardas Roy, Xinming Ou, 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*, pages 1329–1341. ACM, 2014.
- [31] Zipeng Zhang and Xinyu Feng. Androidleaker: A hybrid checker for collusive leak in android applications. In Kim Guldstrand Larsen, Oleg Sokolsky, and Ji Wang, editors, *Dependable Software Engineering. Theories, Tools, and Applications*, pages 164–180, Cham, 2017. Springer International Publishing.
- [32] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP)*, 2012 IEEE Symposium on, pages 95–109. IEEE, 2012.