Combining Static Analysis with Probabilistic Models to Enable Market-Scale Android Inter-component Analysis

Damien Octeau[1, 2], Somesh Jha[1, 3], Matthew Dering[2], Patrick McDaniel[2], Alexandre Bartel[4], Li Li[5], Jacques Klein[5], and Yves Le Traon[5]

[1]Department of Computer Sciences, University of Wisconsin, USA
[2]Department of Computer Science and Engineering, Pennsylvania State University, USA
[3]IMDEA Software Institute, Spain
[4]EC SPRIDE, Technishe Universität Darmstadt, Germany
[5]Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
{octeau, jha}@cs.wisc.edu, {dering, mcdaniel}@cse.psu.edu, alexandre.bartel@ec-spride.de,
{li.li, jacques.klein, yves.letraon}@uni.lu

Abstract

Static analysis has been successfully used in many areas, from verifying mission-critical software to malware detection. Unfortunately, static analysis often produces false positives, which require significant manual effort to resolve. In this paper, we show how to overlay a probabilistic model, trained using domain knowledge, on top of static analysis results, in order to triage static analysis results. We apply this idea to analyzing mobile applications. Android application components can communicate with each other, both within single applications and between different applications. Unfortunately, techniques to statically infer Inter-Component Communication (ICC) yield many potential inter-component and inter-application links, most of which are false positives. At large scales, scrutinizing all potential links is simply not feasible. We therefore overlay a probabilistic model of ICC on top of static analysis results. Since computing the inter-component links is a prerequisite to inter-component analysis, we introduce a formalism for inferring ICC links based on set constraints. We design an efficient algorithm for performing link resolution. We compute all potential links in a corpus of 11,267 applications in 30 minutes and triage them using our probabilistic approach. We find that over 95.1% of all 636 million potential links are associated with probability values below 0.01 and are thus likely unfeasible links. Thus, it is possible to consider only a small subset of all links without significant loss of information. This work is the first significant step in making static inter-application analysis more tractable, even at large scales.

Categories and Subject Descriptors D.2.8 [*Software/Program Verification*]: Statistical methods; F.3.2 [*Semantics of Programming Languages*]: Program analysis

General Terms

Keywords Inter-component communication, Android ICC, static analysis, probabilistic program analysis

1. Introduction

Static analysis is a widely used technique for analyzing software, particularly in the context of security [11, 31]. Static analysis is often followed by a manual confirmation of any issues found in code under scrutiny. If the analysis produces many false positives, then the subsequent manual effort can become prohibitive [25]. This problem is particularly severe in analysis of mobile applications. In Android, applications can reuse other applications' functionality through the use of Inter-Component Communication (ICC), a sophisticated, platform-specific message-passing system. For example, if an application requires dialing a phone number, it can send a completely generic message requesting that an application handle the dialing process. The message need not be targeted at a specific application. In addition, the same communication mechanisms are used to send messages within an application, for example to transition between different user screens.

Misuse of ICC has proven problematic, causing privilege-escalation attacks [8, 10, 14, 15, 22, 32, 48], malicious data access [51] and sensitive data theft [10, 14]. Unfortunately, application markets have many applications, making it hard to have a comprehensive understanding of the ICC ecosystem. Inter-component analysis [24, 26, 28, 42, 46, 49] begins by computing ICC links between message-passing locations and potential message targets. It is important to avoid considering links that may never occur during an execution, since link imprecision propagates to analysis results that are based on ICC. Imprecise results have very limited usefulness, as they increase the amount of manual analysis needed to confirm any potential threat.

Unfortunately, no current analysis technique can infer links in a precise and scalable manner. A variety of tools [16, 37, 38, 46] can statically infer the possible values of *Intents*, which are the main inter-component messages [10]. However, static ICC analysis has inherent limitations that restrict its precision. In particular, Intents are composed of strings of characters that are sometimes impossible to infer precisely and efficiently. Even few imprecisions can result in an explosion of the number of potential ICC links at large scales. This is due to the fact that a conservative matching process has to consider all potential targets for an Intent when one of its field values is not known. For example, ICC inference tools yield millions of potential links at the scale of a single device.

Several techniques aim to associate probabilistic measures with static analysis results. It is possible to propagate probabilistic information about program inputs [12, 34, 35, 41] or to perform prob-

abilistic symbolic execution [19] in order to generate quantitative analysis results. However, these techniques do not address the problem of assigning probabilities to existing, imprecise static analysis results. They are also not concerned with performing analysis at market scale. On the other hand, supervised learning techniques have been used to help classify analysis results [43], but they require manual labeling of a training data set. In our example of ICC analysis, this type of approach would require manual inference of ICC values. The cost of this is too high, since imprecise ICC values are often generated in complex ways. Additionally, source code is almost never available for real-world mobile applications, which further complicates any manual intervention.

In this paper, we show how to overlay a probabilistic model, which is trained using domain knowledge of ICC, on top of static analysis results. We introduce *PRIMO (PRobabilistic ICC MOdeling)*, the first system to triage ICC links based on estimating the probability that they are true positives. The PRIMO system requires *no manual labeling* of analysis results. Our probabilistic model takes into account many aspects of Intents and predicts the expected value of imprecise Intent fields. The model is guided by the insight that Intents are used by developers in predictable ways. More specifically, the patterns of Intent fields and expected targets are similar across message-passing code locations. Some fields may be ambiguously inferred by the static analysis, leading to unfeasible links. However, by utilizing the predictability of ICC patterns, we estimate the probability that ICC links may actually occur.

Since computing ICC links is required for any inter-application analysis, we introduce a formalization of the Intent resolution process that is based on solving set constraints. Our formalism accounts for the case where Intent fields are arbitrary regular expressions, since imprecisely-inferred Intent values are expressed as such. We design an efficient algorithm to compute all ICC links in a large set of applications and analyze its average-case complexity.

By providing an efficient ICC linking process associated with a triage system, we allow market providers to focus their program analysis efforts on the most likely inter-component links. This is especially valuable, since Android application markets have very large numbers of applications. As of October 2015, the Google Play store had over 1.7 million applications [5] and in March 2015 the Amazon Appstore had almost 400,000 applications [40]. In combination with recent efforts to gather large application sets [13, 45], we expect that these techniques will be relevant to security and privacy researchers. Our PRIMO system makes large scale inter-application analyses significantly more tractable. Finally, while this work was motivated by and applied to the problem of ICC analysis in Android, we believe that the approach can generalize to other areas where imprecise analysis results have underlying patterns.

Our paper makes the following contributions:

- We formalize the Intent resolution process by reducing it to a set constraint problem. We design an efficient algorithm to solve the constraints corresponding to ICC links, thereby computing all potential ICC links in an application corpus.
- We present a probabilistic model of ICC message values and show how it can be used to rank ICC links based on the likelihood that they are actual links.
- We implement our techniques in the PRIMO tool and make its source code and documentation available at:

  http://siis.cse.psu.edu/primo/

- We evaluate the ICC link creation algorithm on a corpus of 11,267 applications composed of randomly-selected real world applications, system applications and known malware. We find over 636 million potential links in 30 minutes.

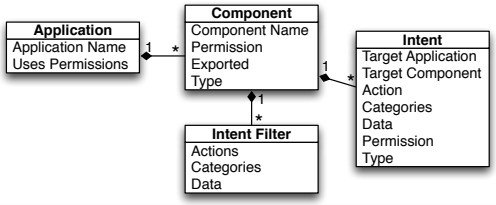


Figure 1. Representation of an Android application.

- We validate our probabilistic model with 10-fold cross-validation and Goodman-Kruskal's γ rank correlation measure[1]. The ground truth is given by the links obtained using precisely known Intent values. Our triage process is applied after introducing imprecisions in these precise values, allowing comparison with the ground truth while simulating real-world imprecision. Our methods yield a ranking of links that is strongly correlated with the ground truth ($\gamma > 0.97$).
- We utilize our model to rank ICC links and find that over 95.1% of all links are likely unfeasible. This shows that our model can be used as a triage system for ICC links.

2. Background

2.1 Android ICC

Figure 1 shows a conceptual representation of an Android application. Android applications are built in components, which perform specific tasks. In practice, implementing a component is done by subclassing one of four classes from the Android framework. *Activities* are the most common component. They represent a user screen. *Services* perform long-running background processing. *Broadcast Receivers* receive system-wide notifications, such as the one that is sent by the operating system when a text message is received. Finally, *Content Providers* provide a way of sharing structured data between applications.

Almost all components are declared in the manifest file that is part of all application packages. The only exception is that Broadcast Receivers can be created and registered dynamically in the application code at runtime. Application components can communicate with one another using two mechanisms, *Uniform Resource Identifiers (URIs)* and *Intents*. URIs are used to address Content Providers. Intents, on the other hand, are messages that are sent between the three other component types[2]. An Intent can be *explicit*, which means that its target is explicitly named. In other words, the Intent specifies the application and the class name for its target. Intents can also be *implicit*, in which case they only specify the functionality that they desire for their target. In this case, the desired functionality is described using three items:

- An *action* string specifies the action to be performed with the Intent. A common action is the `VIEW` action, which is utilized when some data needs to be displayed (e.g., web page).
- A set of *category* strings describes additional information about what should be done with the Intent. For example, `BROWSABLE` indicates that the target can be safely invoked from a browser.
- A set of *data* fields specifies data to be acted upon. This can for instance be a web address or a phone number.

Data can be described by a URI (e.g., file paths or web addresses). A MIME type can also be specified to describe the data to be acted

[1] Goodman-Kruskal's γ [20] measures the similarity of the orderings of two ordinal variables. It is well-suited for our problem because of its resilience to ties in the orderings of the variables.

[2] Since URIs are a small minority of ICC addressing as we discuss in Section 5.2, we omit URIs from Figure 1 and from the remainder of this paper. However, the ideas presented for Intents apply to URIs as well.

```
1  public void sendExplicitIntent() {
2    Intent intent = new Intent();
3    intent.setComponentName("my.second.app", "Dialer");
4    startActivity(intent); }
5  public void sendImplicitIntent() {
6    Intent intent = new Intent();
7    intent.setAction("DIAL");
8    Uri phoneNumber = Uri.parse("tel:1234567890");
9    intent.setData(phoneNumber);
10   startActivity(intent); }
```

(a) Explicit and implicit Intents.

```
1 <activity android:name="Dialer" android:exported="true">
2   <intent-filter>
3     <action android:name="DIAL"/>
4     <action android:name="VIEW"/>
5     <data android:scheme="tel"/>
6     <category android:name="DEFAULT"/>
7   </intent-filter> </activity>
```

(b) Intent Filter for a dialer component.

Figure 2. Intents to start a dialer and the associated Intent Filter.

upon. For ease of exposition, in the remainder of this paper we do not describe all data fields separately.

Components that wish to receive implicit Intents have to declare *Intent Filters*, which describe the attributes of the Intents that they are willing to receive. They can include actions, categories and data descriptors. Components have an *exported* attribute, which when set to true makes the components accessible to other applications through ICC. Components that are not exported are only accessible to other components in the same application. Component access can also be protected by a permission. When a component declares a permission, applications need to request the permission at install time in order to be able to send Intents to the component. This is done using the *uses-permissions* attribute in the manifest file. Intents that target Broadcast Receivers can also be protected by a permission, in which case the application containing the target component needs to request the permission.

Matching Intents with their target is done by the operating system during an *Intent resolution* process. For implicit Intents, it involves matching the action, category and data fields with compatible Intent Filters. In this paper, we statically perform this matching process, without executing the applications. Note that in Figure 1 some fields can be undefined. For example in an explicit Intent the action, categories and data fields are usually null.

2.2 Android ICC Analysis

Figure 2(a) shows examples of Intents. The first method creates an explicit Intent targeted at component `Dialer` in application `my.second.app`. The $startActivity()$ call causes the Intent to be sent to the recipient. The second method creates an implicit Intent and sets its action to `DIAL`. It then sets a phone number as the Intent data. The $startActivity()$ method call causes the Intent resolution process to find a dialer component from which the user can call the phone number. Figure 2(b) presents an example component declaration for a dialer Activity. Its Intent Filter declares handling the `DIAL` and `VIEW` actions and data with a `tel` scheme. The default category declaration at Line 6 is required, since this category string is automatically added to all Intents targeted at Activities. This Intent Filter enables the component to receive the Intent declared in method $sendImplicitIntent()$ of Figure 2(a).

A prerequisite for statically matching Intents with their possible targets is to determine the values of Intents, Intent Filters and URIs [37, 38]. Most Intent Filter values are obtained with a straightforward parsing process of the manifest file. URIs, Intents and dynamically registered Intent Filters, on the other hand, are inferred using a complex static data flow analysis. The Epicc [38] and IC3 [37] tools perform this analysis. IC3 is more precise than Epicc, which is due to a more sophisticated analysis of the strings used for actions, categories and data. Further, IC3 can handle URIs, whereas

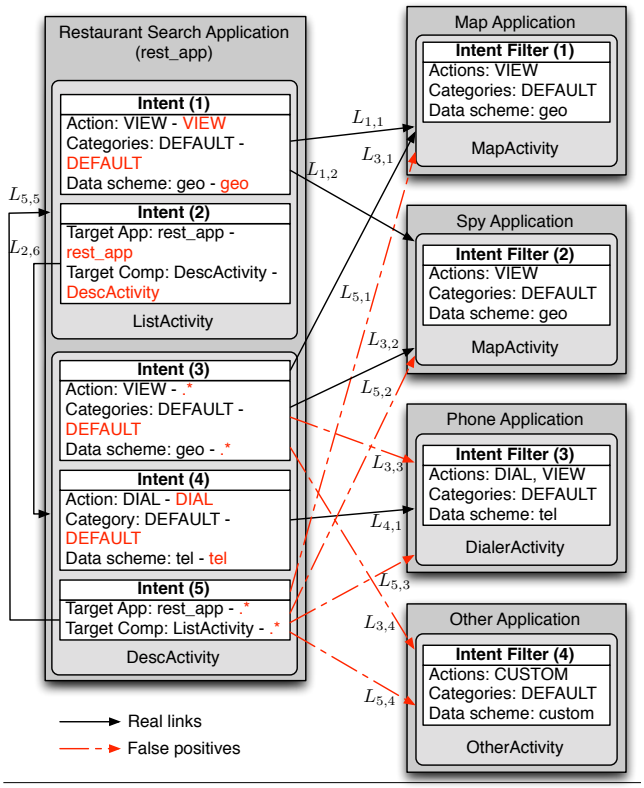


Figure 3. Running example. Fields values in red indicate the values inferred by the ICC inference process, with `.*` being a regular expression that matches any string. See details in Section 2.3.

Epicc cannot. This results in more precise Intent values, since URIs are used to specify the data field of Intents. In our experiments, we use ICC values computed by IC3 [37].

A limitation of statically inferring ICC values is that some of these values are either too computationally expensive to infer statically or they are determined by runtime context. For example, it may not be possible to infer the action field of an Intent because it is generated in a way that cannot be efficiently modeled with static analysis, such as when it flows through a container type. In this case, the action field cannot be used for matching with potential target Intent Filters. This can result in matching Intents with many Filters, even though such links are not possible in practice.

Throughout this paper, we say that a link $L_{i,f}$ between an Intent i and an Intent Filter f is a true positive if the real value of the Intent matches the Intent Filter. On the other hand, $L_{i,f}$ is a false positive if the value of i inferred with static analysis matches f, even though the real value of i does not.

2.3 Running Example

Figure 3 shows a running example that will be used throughout the paper. We consider an application `rest.app` that allows users to find restaurants near their current location. It is composed of two Activity components (i.e., two user screens). The `ListActivity` component displays a list of nearby restaurants. It may send implicit Intent (1) to display a map with all nearby restaurants. It can also send explicit Intent (2) to display details about a particular restaurant. The `DescActivity` component displays descriptions of specific restaurants. It can display a map centered on a restaurant by sending implicit Intent (3), or it may trigger a phone call to the restaurant by emitting implicit Intent (4). Finally, it can start the ListActivity component by sending explicit Intent (5) to it.

```
1 public void onClick(View v) {
2   Location loc =
        LocationManager.getLastKnownLocation("gps");
3   Uri query = Uri.parse("geo:" + loc.getLatitude() + ","
        + loc.getLongitude() + "?q=restaurants");
4   Intent intent = new Intent("VIEW", query);
5   startActivity(intent); }
```

(a) Click handler sending Intent (1) from Figure 3.

```
1 public class MapActivity extends Activity {
2   public void onCreate(Bundle b) {
3     Uri location = getIntent().getData();
4     SmsManager.getDefault().sendTextMessage("12345",
        null, location.toString(), null, null); }}
```

(b) Code leaking location data in the spy application from Figure 3.

Figure 4. Inter-application leak of location data to text message.

On the same theoretical device, there are four other applications with exported components. The map application renders a map with given coordinates. Its `MapActivity` component declares an Intent Filter that accepts URI data with a `geo` scheme, which is meant to indicate geographic coordinates. The spy application declares accepting Intents with geographic data. It then leaks the data to an unauthorized third-party. The phone application declares a dialer component that receives Intents with a `tel` scheme, which contain telephone numbers. Finally, another application has a component that declares a custom action and a custom data scheme. This can be useful if an application performs an action that is not described by the default actions strings.

For each Intent, we first indicate the real field values (that is, as declared by the application developer). We also indicate in red font the field values that are inferred by the static analysis. For example, the real value of the categories field of Intent (3) is accurately inferred as `DEFAULT`. On the other hand, the action field, which is declared to be `VIEW`, is observed by the static analysis process as `.*`. Similarly, the `geo` data scheme is imprecisely inferred as `.*`.

We have represented ICC links that may occur with black arrows. These links are all inferred by the static analysis since it is a conservative process. However, because of the limited precision of static analysis, unfeasible links are also inferred. They are shown in Figure 3 with red dashed arrows. For example, let us consider Intent (3). Since its action and data scheme fields are not inferred as constants, they are matched with Intent Filters that have `DIAL` and `CUSTOM` actions, and `tel` and `custom` data schemes. Similarly, the fields of explicit Intent (5) are all inferred as the `.*` regular expression, which matches all the components in all applications.

Figure 4(a) shows the code that sends Intent (1). It uses a `VIEW` action and a URI query that contains the user's location data. The Intent is sent at Line 5. The system may then deliver it to Activity MapActivity of the spy application from Figure 3. When the spy application is started, it executes the code shown in Figure 4(b). This code extracts the URI data containing the user location and sends it to an attacker using an SMS message. Note that Intent (3) is sent using code similar to Figure 4(a).

Existing work on information flow analysis [6, 26, 28] can detect that sensitive location information is sent using ICC in the restaurant search application. It can also detect that information from an incoming Intent in the spy application can flow to the SMS manager. However, it is currently not possible to infer the end-to-end inter-application leak, due to the large number of statically-computed ICC links when a non-trivial number of applications are analyzed. In this paper, we aim to perform triage on these links, with the goal of prioritizing the true positives over the false positives. The expected outcome is to associate a priority value with each link such that the priority values of real links are greater than the priority values of false positives. In particular, real links to the spy or map applications should have a high priority. In Section 4, we show how to approximate the likelihood that a given link is a true positive by utilizing a probabilistic model of Intents. We subsequently use this approximation to rank the links based on the probability that they are true positives. Links with high priority may be benign or malicious and our techniques are not meant to discriminate between benign and malicious links. Existing work [26, 28] can perform such analysis on the high-priority links.

3. Intent Resolution through Set Constraints

Let us assume that we are given a set of Intents I and a set of Intent Filters F. We would like to find all tuples $(i, f) \in I \times F$ such that there is a potential ICC link between Intent i and Filter f. In this section, we formalize the Intent resolution process as set constraints. That is because Intent and Filter fields can be expressed as sets and solving set constraints has been widely studied [2, 3]. For ease of exposition, we only show the Intent resolution process. Resolving URIs to Content Providers uses similar techniques.

3.1 Insights and Challenges Related to Set Constraints

Problems in program analysis can often be represented as set constraints [3], which can be solved with existing algorithms [1]. However, our problem has several key differences with the traditional formalism. First, even though most of our sets only include constants, our sets can also contain regular expressions because the static analysis does not always yield constant strings. Second, a majority of the set constraints that model our problem are $p(i) \subseteq q(f)$, where $p(i)$ is an Intent field and $q(f)$ is a Filter field. That is, Intents only appear on the left-hand side of almost all our constraints.

We take advantage of the structure of the problem to solve our constraints. Our algorithm performs regular expression matching as necessary, while also performing fast matching for the common case of constant fields. Further, solving constraints in a traditional way would require performing the constraint solution algorithm for all tuples in $I \times F$. This is inefficient, particularly at large scales. However, by exploiting the fact that Intents only appear at the left-hand side of most constraints, we perform matching through a series of set intersections, taking on average $O(e_{min} \cdot |I|)$, with $e_{min} < |F|$ a constant that will be defined in Section 3.5.

3.2 Set Constraints

A set constraint is a constraint of the form $E_1 \subseteq E_2$, where E_1 and E_2 are set expressions that can be constructed for example by using set constants, set variables, and the union operator. Let $\mathcal{V}$ be a set of set variables and $\mathcal{R}$ a set of regular expressions. The set expressions in our problem have the following form:

$$E ::= \alpha \mid 0 \mid E_1 \cup E_2 \mid E_1 \cap E_2 \mid r,$$

with $\alpha \in \mathcal{V}$, $r \in \mathcal{R}$ and 0 is the empty set expression.

For a regular expression r, $L(r)$ denotes the language described by r. For example, $L(\texttt{.*a})$ is the set of strings ending with `a`. In order to account for the possible presence of regular expressions in E_1 and E_2, we extend the $\subseteq$ relation by saying that $E_1 \subseteq E_2$ if for each element $s_1 \in E_1$, there is an element $s_2 \in E_2$ such that $L(s_1) \cap L(s_2) \neq \varnothing$. For example, with this extension of $\subseteq$ we have $\{\texttt{ab.*}\} \subseteq \{\texttt{.*bc}, \texttt{def}\}$, since $L(\texttt{ab.*}) \cap L(\texttt{.*bc}) \neq \varnothing$. Note that this definition does not imply anything about the *inclusion* of languages described by regular expressions, but instead it only considers *intersection*. In our example, this ensures that we consider Intent field value {`abc`} to match Filter field value {`abc`, `def`}. Thus our process ensures that individual fields are matched in a conservative manner. In most cases the static ICC analysis can find precise values [37], and in these cases the regular expressions reduce to simple constants. That is, they are strings c such that $L(c) = \{c\}$. A system S of set constraints is a conjunction of set constraints: $S = \bigwedge_i E_{i,1} \subseteq E_{i,2}$. Solving S consists in finding all possible assignments of set variables such that S is satisfied.

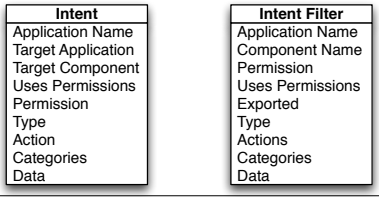


Figure 5. Representation of Intents and Intent Filters used for the linking process.

3.3 Formalizing Intent Resolution

In order to make the resolution process as generic as possible, we assume that each Intent and each Intent Filter contains all attributes necessary for the linking process. For example, in addition to the attributes that are set in the code by an application developer (e.g., action, categories), an Intent is assumed to contain the name of the application sending the Intent and the permissions that the application requests at install time. Figure 5 shows a description of Intents and Intent Filters used for matching. This representation is trivially obtained from the application representation shown in Figure 1. In order to handle explicit Intents in a generic way, we introduce Intent Filters for all components. Every component has at least one Filter with an Application Name and a Component Name attribute. The alternative would consist in using a resolution process between Intents and components for explicit Intents. Instead, we use a single Intent-to-Filter matching process for all Intents.

We represent each attribute of Intents and Intent Filters as a set. For example, an Intent may have no action or a single action. We represent the case with no action with an empty set 0. In the case of a single action string a, we use set $\{a\}$. In the case of a Boolean attribute such as the Exported flag, we use sets $\{\texttt{true}\}$ or $\{\texttt{false}\}$. This uniform representation allows us to represent the entire resolution process as a system of set constraints.

We denote the action of Intent i by i_{action}. We use similar notations for other Intent and Intent Filter fields. Traditional constraint systems translate set equality to inequalities using the fact that for sets A and B, $A = B$ is equivalent to $A \subseteq B \wedge B \subseteq A$. We additionally use the fact that $A = B$ is equivalent to $A \subseteq B \wedge |A| = |B|$, since several constraints involve sets that we know to be of the same size. For example, one constraint is that the component type targeted by an Intent i_{type} is the same as the type of component associated with a Filter f_{type}. Since there is only a single type associated with each Intent and each Filter, we use constraint $i_{type} \subseteq f_{type}$. This allows us to use the generic constraint solving process from Section 3.4 to solve set equality constraints.

Given an Intent i and an Intent Filter f, we say that i matches f if $match(i, f)$ is satisfied, where $match(i, f)$ is true if the following boolean expression evaluates to true:

$$match(i, f) = type(i, f) \wedge visibility(i, f) \wedge perm(i, f) \quad (1)$$
$$\wedge\, (explicit(i, f) \vee implicit(i, f)).$$

This expression is a formalization of the Intent resolution process as described in online documentation [21] and in source code.

As mentioned above, $type(i, f) = i_{type} \subseteq f_{type}$. Also, $visibility(i, f)$ evaluates to true if Filter f is visible from Intent i. Predicate $perm(i, f)$ is true if Intent i has permission to access Filter f and f is allowed to receive i. Finally, $explicit(i, f)$ (respectively, $implicit(i, f)$) is true if fields of i specific to explicit (respectively, implicit) Intents match the fields of f.

The visibility criterion states that i can access f if i and f are in the same application or f is exported. This is expressed as:

$$visibility(i, f) = i_{app\_name} \subseteq f_{app\_name} \vee f_{exported} \subseteq \{\texttt{true}\}$$

The permission condition stipulates that if a Filter is protected by a permission, then the application sending an Intent to it needs to request that permission at install time. Also, if the Intent is protected by a permission, then the receiving Filter must belong to an application that declares the permission. This is expressed as:

$$perm(i, f) = i_{perm} \subseteq f_{uses\_perm} \wedge f_{perm} \subseteq i_{uses\_perm}$$

An Intent explicitly matches a Filter if $explicit(i, f)$ is verified:

$$explicit(i, f) = i_{target\_comp} \neq \varnothing \wedge i_{target\_app} \subseteq f_{app\_name}$$
$$\wedge\, i_{target\_comp} \subseteq f_{comp\_name}$$

On the other hand, the expression is more complex in the case of implicit Intents:

$$implicit(i, f) = i_{target\_comp} = \varnothing \wedge i_{action} \subseteq f_{actions}$$
$$\wedge\, i_{category} \subseteq f_{categories} \wedge data(i, f),$$

where $data(i, f)$ represents the data test for Intent i and Filter f. It consists in checking that the data types, schemes, and other URI data parts are compatible. We do not give its expression for ease of exposition but it is also expressed using similar set constraints.

For example, let us assume that application `rest_app` from Figure 3 requests permissions for Internet and location access. Then the value statically inferred for explicit Intent (5) (denoted by x) is such that:

$$x_{app\_name} = \{\texttt{rest\_app}\} \qquad x_{target\_app} = \{\texttt{.*}\}$$
$$x_{uses\_perm} = \{\texttt{INTERNET}, \texttt{ACCESS\_FINE\_LOCATION}\} \qquad x_{target\_comp} = \{\texttt{.*}\}$$
$$x_{type} = \{\texttt{activity}\}$$
$$x_{action} = x_{data} = x_{categories} = \varnothing \qquad x_{perm} = \varnothing$$

Recall from Section 2.3 that the `.*` values are string values that are imprecisely inferred by static analysis. Implicit Intent (4) (denoted by i) is such that:

$$i_{app\_name} = \{\texttt{rest\_app}\} \qquad i_{target\_app} = i_{target\_comp} = \varnothing$$
$$i_{uses\_perm} = \{\texttt{INTERNET}, \texttt{ACCESS\_FINE\_LOCATION}\} \qquad i_{type} = \{\texttt{activity}\}$$
$$i_{perm} = \varnothing$$
$$i_{action} = \{\texttt{DIAL}\} \qquad i_{categories} = \{\texttt{DEFAULT}\}$$
$$i_{data} = \{\texttt{tel}\}$$

Note that the `DEFAULT` category is added by the operating system to all Intents targeting Activities. For simplification, we have reduced the data field to a URI scheme. As mentioned above, in reality data is described by several fields. Intent Filter (3) (denoted by f) from Figure 3 is modeled by:

$$f_{app\_name} = \{\texttt{phone\_app}\} \qquad f_{comp\_name} = \{\texttt{DialerActivity}\}$$
$$f_{uses\_perm} = f_{perm} = \varnothing \qquad f_{type} = \{\texttt{activity}\}$$
$$f_{action} = \{\texttt{DIAL}, \texttt{VIEW}\} \qquad f_{categories} = \{\texttt{DEFAULT}\}$$
$$f_{data} = \{\texttt{tel}\} \qquad f_{exported} = \{\texttt{true}\}$$

It is possible to verify that $match(x, f)$ and $match(i, f)$ hold true using the above description.

3.4 Solution of Set Constraints with Regular Expressions

In this section, we present an algorithm that can be utilized to find all ICC links between Intents in I and Filters in F. It is more efficient than solving constraints for all tuples in $I \times F$. Further, it takes regular expressions into account, which is necessary to handle field values that are not precisely inferred using static analysis.

For each Intent value, the match set is initialized to be all Intent Filters. For each Intent field value, it finds the compatible Filters in the match set, effectively reducing the size of the match set for each field value. *The key idea behind this algorithm is that for a constant Intent field value, we can immediately obtain all Intent Filters with a compatible constant field value. Regular expression matching is subsequently performed for the minority of non-constant values.*

It is useful to separate Equation (1) into two cases. First, if $i \in I$ is an explicit Intent, we have:

$$match(i, f) = type(i, f) \wedge visibility(i, f) \wedge perm(i, f)$$
$$\wedge\, i_{target\_comp} \subseteq f_{comp\_name} \quad (2)$$
$$\wedge\, i_{target\_app} \subseteq f_{app\_name}$$

Input:

- $intents$: set of all Intents
- $filters$: set of all Filters that should be matched with Intents
- $attribute\_maps$: maps between field values and the Intent Filters that include them

```
1 procedure FindAllLinks(intents, filters, attribute_maps)
2     links := empty set
3     for all Intents i in intents
4         matches := filters
5         for all fields field in i.fields
6             maps := attribute_maps[field]
7             matches :=
              FindFiltersWithAttributes(i.field, matches,
                maps.constant_map, maps.regex_map)
8         Add (i, current_matches) to links
9     return links
```

Algorithm 1: Intent resolution procedure.

On the other hand, for an implicit Intent we use:

$$match(i,f) = type(i,f) \wedge visibility(i,f) \wedge perm(i,f) \wedge i_{action} \subseteq f_{actions} \wedge i_{category} \subseteq f_{categories} \wedge data(i,f) \quad (3)$$

Algorithm 1 presents the procedure for finding all links between Intents in I and Filters in F. For each attribute type (e.g., action and categories), we maintain a map between each attribute value and the set of Filters that declare it. For example, we maintain a map of actions, where keys are action strings and values are the Filters that declare each action key. For each field, we maintain a map where keys are constants and another one where keys are regular expressions. All the resulting maps are contained in a variable $attribute\_maps$. For example, in Figure 3, we have 4 Intent Filters. The category constant map would contain a single `DEFAULT` key mapping to all Filters. On the other hand, the action constant map would contain 3 entries. Key `VIEW` maps to Filters (1), (2) and (3), key `DIAL` maps to Filter (3) and `CUSTOM` maps to Filter (4). Since all Filters have constant field values, all regular expression maps are empty. Procedure FindAllLinks is called with arguments I, F and $attribute\_maps$. The set of potential recipients is initialized to be F and it is reduced by every field test at Line 7. Note that for ease of exposition we do not show particular cases such as the disjunction in predicate $visibility(i, f)$.

Algorithm 2 shows the FindFiltersWithAttributes procedure for finding all Filters with a set of attributes. In other words, for any set A, it finds all sets B such that $A \subseteq B$. It performs two important functions. First, it matches constant Intent field values with constant Filter field values in an efficient manner. Second, it handles cases where either Intent or Filter field values are regular expressions. Since attributes may be regular expressions, FindFiltersWithAttributes takes as inputs both a map between string constants and Filters and another map between regular expressions and Filters. Finally, it takes as input the set of Filters currently being considered. This procedure computes the intersection of all Filters that have each one of the input attributes. It uses procedure FindFiltersWithAttribute, which finds all Filters that have a given attribute, taking into account regular expressions.

In procedure FindFiltersWithAttribute, if the input attribute is a string constant, then we start by adding all Filters that declare the attribute to the set of Filters to be returned (Line 10). If the input attribute is not a constant, then we need to match it with all constant Filter attributes by iterating through the map of constant Filter attributes (Lines 12-14). For example, if the input attribute is `.*`, then all Filters with a constant attribute will match. Then, we proceed to match $attr$ with all Filters that declare regular expressions (Lines 15-17). For example, if a Filter is found to declare `.*` by the static analysis, then all input attributes will match it.

Input:

- $attrs$: set of Intent attributes to match to Filters
- $filters$: set of Filters under consideration
- $constant\_map$: map between Filter attributes that are simple constants and the Filters that contain them
- $regex\_map$: map between Filter attributes that are regular expressions and the Filters that contain them

```
1  procedure FindFiltersWithAttributes(attrs, filters,
   constant_map, regex_map)
2      result := filters
3      for all attributes attr in attrs
4          found := FindFiltersWithAttribute(attr,
               constant_map, regex_map)
5          result := result ∩ found
6      return result

   // Procedure to find all Filters that include a given attribute.
7  procedure FindFiltersWithAttribute(attr, constant_map,
   regex_map)
8      result := empty set
9      if attr is a constant then
10         Add constant_map[attr] to result
11     else
12         for all pairs (attribute, filters) in constant_map
13             if L(attr) ∩ L(attribute) ≠ ∅ then
14                 Add filters to result
15     for all pairs (attribute, filters) in regex_map
16         if L(attr) ∩ L(attribute) ≠ ∅ then
17             Add filters to result
18     return result
```

Algorithm 2: Procedures to efficiently find all Filters that include a given set of attributes.

Returning to our running example from Figure 3, let us consider the matching process for the `DEFAULT` category field of Intent (1). Since it is a constant, Line 10 would immediately yield all 4 Intent Filters. Also, no Filter has non-constant categories, so Lines 15-17 would not be executed. On the other hand, let us consider the `.*` action value of Intent (3). Lines 12-14 would match with the Filters with `DIAL`, `VIEW` and `CUSTOM` action strings.

The only constraint that requires different treatment is:

$$perm(i,f) = i_{perm} \subseteq f_{uses\_perm} \wedge f_{perm} \subseteq i_{uses\_perm}$$

The first clause is solved by using the method above. The second clause needs special treatment since the Intent attribute set is on the right-hand side of the inequality and the method above is designed to solve constraints with the Filter attribute on the right-hand side. We verify the second clause after all other clauses have been verified by iterating through all candidate Filters as they are added to the final result set. This does not impact performance, since the final set of candidate Filters is typically small. Further, this iteration occurs in any case due to the need to store the resulting links between the Intent being considered and the possible target Filters.

The procedure described in Algorithm 1 performs a sound matching for the set of Intents provided as input. Informally, for each Intent, we start at Line 4 with the set of all targets. At each iteration of the loop at Line 5, we only exclude targets that do not match a particular Intent field. At the end of the loop, after all fields are processed, the set of potential matches includes all targets that verify all clauses of the conjunctions given by Equation (2) (for explicit Intents) or (3) (for implicit Intents).

3.5 Average Complexity Analysis

The key insight behind the complexity analysis of procedure FINDALLLINKS is that most Intent and Filter field values are constants. That is because almost all Filters are known through parsing of the application manifest file. Thus, the cost of the regular expression matching at Lines 15-17 of FINDFILTERSWITHATTRIBUTE is negligible. The Filter attribute maps can be implemented with a hash table, allowing constant-time access to all Filters with a given attribute at Line 10. Since most Intent attributes are also simple constants, the iteration at Lines 12-14 has negligible cost on average. Additionally, we consider the time complexity of deciding if two regular expressions have a non-empty intersection (i.e., if $L(a) \cap L(b) \neq \varnothing$) to be bounded by a constant, since all regular expressions in our problem have length bounded by a constant [18].

The cost of Lines 4-8 of procedure FINDALLLINKS is therefore dominated by the set intersection at Line 5 of FINDFILTERSWITHATTRIBUTES. For each Intent, across all calls to FINDFILTERSWITHATTRIBUTES this line is computing $F \cap \bigcap_{i=1}^{m} A_i$, where F is the set of all Intent Filters, m is the number of Intent field elements and A_i are the sets of Filters matching each Intent field element. The time complexity of the set intersection operation $F \cap \bigcap_{i=1}^{m} A_i$ is bounded by $m \cdot min_i(|A_i|)$. Let A_{min} be the set such that $min_i(|A_i|) = |A_{min}|$. In our case m is bounded by a small constant because the number of fields is low and for fields that are sets the number of elements is low (typically 0 or 1), thus the time complexity of Line 5 is $O(|A_{min}|)$. Note that in order to achieve the $m \cdot |A_{min}|$ complexity, the first intersection that is computed must have A_{min} as one of its operands.

For any field *field*, let e_{field} be the expected number of Intent Filters matching a single element of *field*. That is, e_{field} is the weighted average number of Intent Filters that have matches for elements of *field*. For example, in our running example from Figure 3, the expected number of matches of the fields of the implicit Intents (i.e., Intents (1), (3) and (4)) is:

$$e_{action} = \frac{1}{3} \cdot 3 + \frac{1}{3} \cdot 4 + \frac{1}{3} \cdot 1 \approx 2.67$$

$$e_{categories} = 1 \cdot 4 = 4$$

$$e_{scheme} = \frac{1}{3} \cdot 2 + \frac{1}{3} \cdot 1 + \frac{1}{3} \cdot 4 \approx 2.33$$

Let e_{min} denote the minimum of all e_{field} values. The average complexity of Line 5 of FINDFILTERSWITHATTRIBUTES is $O(e_{min})$. To achieve this complexity, we must ensure that the fields at Line 5 of procedure FINDALLLINKS are ordered such that the first field $field$ is the one for which $e_{min} = e_{field}$. For example, this can be done by first sampling the data to estimate e_{min}. We can summarize our complexity analysis with the following theorem.

Theorem 1. *The average running time of procedure* FINDALLLINKS *is* $O(e_{min} \cdot |I|)$.

4. A Probabilistic Model of Intents

4.1 Overview

Recall from the previous sections that our observations of ICC values using static analysis may be noisy. For a given string value st, we may either observe its actual value, or a regular expression r such that $st \in L(r)$. When the latter is inferred, our Intent resolution process infers links that may not occur at runtime. Given a link between an Intent and a Filter, we would like to estimate the likelihood that the link is a true positive. This is challenging, because whether a string value st is inferred as st or as a regular expression does not depend on the value itself. It depends on how the application programmer generates it and on the specific static analysis technique. Thus we cannot model the mapping between st and its observed value with a specific probability distribution.

For example, in Figure 3 the action field of Intent (3) is inferred by static analysis as `.*` instead of the actual value `VIEW`. The fact that the `VIEW` value is inferred as `.*` does not depend on the `VIEW` value itself, but instead it depends on the code of the `DescActivity` component. In other words, we cannot model the process that maps real fields values to statically-inferred values.

Thus, we model Intent usage patterns, observing that application developers often use similar ICC patterns across different applications. For example, in order to dial a telephone number, developers send an implicit Intent with field values as shown in method $sendImplicitIntent()$ in Figure 2(a). The corresponding Intent has a `DIAL` action, a `DEFAULT` category and a `tel` scheme. The intuition is to measure the relative frequency of known field patterns to infer the likelihood that unknown fields values match a given Filter. For example, in Figure 3, consider link $L_{3,1}$ between Intent (3) and Filter (1). We would like to compute the likelihood that the Intent action matches `VIEW` and the scheme matches `geo`, given that the category is inferred as `DEFAULT`.

In the next sections, for any link $L_{i,f}$ between an Intent i and an Intent Filter f, we express the likelihood that $L_{i,f}$ is a true positive as a probability $P_{i,f}$, which is formally defined in Section 4.2. If all fields of Intent i are unambiguously known, then trivially $P_{i,f} = 1$. In Figure 3, this is the case for link $L_{1,1}$, since all fields of Intent (1) are inferred precisely. On the other hand, for a link $L_{i,f}$ that is deemed unfeasible, $P_{i,f} = 0$. For any other link for which some or all fields are not precisely known, $P_{i,f}$ is estimated by using a probabilistic model. The model is trained with data collected for Intents for which all fields are completely known. The training data provides us with empirical probabilities of occurrence of various Intent field patterns. For example in Figure 3, let us estimate the probability $P_{3,1}$ that link $L_{3,1}$ is a true positive. The only implicit Intents for which all fields are unambiguously known (i.e., our training set) are Intents (1) and (4). Both Intents are considered for our probability computation because they have category value `DEFAULT` like Intent (3). However, only one of these two Intents (i.e., Intent (1)) also matches Intent Filter (1) because it has action value `VIEW` and scheme value `geo`. Thus the likelihood $P_{3,1}$ is one out of two ($\frac{1}{2} = 0.5$). A rigorous computation of $P_{3,1}$ is shown at the last paragraph of Section 4.3. On the other hand, for link $L_{3,4}$ we have $P_{3,4} = \frac{0}{2} = 0$, because none of the two Intents (1) or (4) has action value `CUSTOM` and scheme value `custom`. Thus, our triage would prioritize $L_{1,1}$ over $L_{3,1}$, which itself would be prioritized over $L_{3,4}$. Section 4.3 formalizes this intuition.

The explicit Intent model is built on similar ideas. It is based on the observation that most explicit Intents are used for intra-application communication. In our running example intra-application links $L_{2,6}$ and $L_{5,5}$ would be prioritized over inter-application links $L_{5,1}$, $L_{5,2}$, $L_{5,3}$ and $L_{5,4}$, since explicit inter-application links are likely false positives. Link $L_{2,6}$ would itself be prioritized over link $L_{5,5}$ because it is a known true positive, since Intent (2) has a precise value. Section 4.4 describes the formalism for this idea.

We will restrict our study to the n fields that are subject to static analysis imprecision, including the action and the categories fields among others. On the other hand, other fields are always precisely known (e.g., the target component type). As described in Section 3.3, the fields of an Intent $i = (i_1, \ldots, i_n)$ and a matching Filter $f = (f_1, \ldots, f_n)$ have a set inclusion relationship:

$$i_1 \subseteq f_1, \ldots, i_n \subseteq f_n.$$

4.2 Modeling ICC Links

Let random variables $I_1, \ldots, I_n$ represent the true field values of a given Intent and let random variables $\hat{I_1}, \ldots, \hat{I_n}$ represent the values inferred using static analysis (the *observed* values).

Definition 1. *The probability that a link between Intent $i = (i_1, \ldots, i_n)$ and a matching Filter $f = (f_1, \ldots, f_n)$ is a true positive is:*

$$P_{i,f} = P\left(\bigwedge_{k=1}^{n} I_k \subseteq f_k | \bigwedge_{k=1}^{n} \hat{I}_k = i_k\right), \quad (4)$$

where $I_i \subseteq f_i$ denotes the event that I_i takes a value matching f_i.

Informally, $P_{i,f}$ is the probability that the true field values of i match the field values of f, given the observed values for the fields of i. Note that we only calculate these probabilities for links that are computed by using the matching process described in Section 3.4.

Let us consider the example of link $L_{3,1}$. The probability that it is a true positive is:

$$\begin{aligned} P_{3,1} = P(&I_{action} \subseteq \{\texttt{VIEW}\} \wedge I_{categories} \subseteq \{\texttt{DEFAULT}\} \\ &\wedge I_{scheme} \subseteq \{\texttt{geo}\} \\ &|\hat{I}_{action} = \{\texttt{.*}\} \wedge \hat{I}_{categories} = \{\texttt{DEFAULT}\} \\ &\wedge \hat{I}_{scheme} = \{\texttt{.*}\}). \end{aligned}$$

This is the probability that the fields of Intent (3) match the fields of Intent Filter (1), given that the static analysis infers value `.*` for the action and scheme of Intent (3) and value `DEFAULT` for its category.

Let C be the set of constant strings in the programs being analyzed. Inferred field values may either be in C, or they may be described by a non-constant regular expression. We assume without loss of generality that the fields with constant values have consecutive indices $1, \ldots, l$, for some l. In order to obtain a tractable model of ICC objects, we make the following assumption.

(A1) For a given l such that $1 < l < n$, the distribution of field values inferred by static analysis $\hat{I}_{l+1}, \ldots, \hat{I}_n$ and the distribution of actual values $I_{l+1}, \ldots, I_n$ are conditionally independent given inferred field values $\hat{I}_1, \ldots, \hat{I}_l$. In other words, given the static analysis results for fields 1 through l, there is no correlation between the observed values of fields $l+1$ through n and their actual values. This is because, as we mentioned in Section 4.1, the fact that a given value is inferred imprecisely or not is not dependent on the value itself but instead it is related to other parts of the application code and the static analysis technique.

4.3 Implicit Intents

We begin by stating a theorem that will be used to estimate the probability that links caused by implicit Intents are true positives. The proof for this theorem is presented in Appendix A.1. The intuition behind this theorem is that because of Assumption (A1), we can ignore the inferred values for imprecise fields in Equation (4).

Theorem 2. *Let $i = (i_1, \ldots, i_n)$ be an Intent value as inferred using static analysis and $f = (f_1, \ldots, f_n)$ an Intent Filter value. Assume that observed values $i_1, \ldots, i_l$ are in C and that all other observed values $i_{l+1}, \ldots, i_n$ are not. Then:*

1. *If $l = n$ (all fields are constant strings), then $P_{i,f} = 1$.*
2. *If $l = 0$ (that is, all field values are non-constant regular expressions), then we have:*

$$P_{i,f} = P\left(\bigwedge_{k=1}^{n} I_k \subseteq f_k\right).$$

3. *For any $l \in \{2, \ldots, n-1\}$, we have:*

$$P_{i,f} = P\left(\bigwedge_{k=l+1}^{n} I_k \subseteq f_k | \bigwedge_{k=1}^{l} \hat{I}_k = i_k\right).$$

Let K be the set of Intents for which all field values are known, i.e. our *training data set*. For any Intent in K, $P_{i,f}$ trivially evaluates to 1. For other cases, we essentially estimate the probabilities over the corpus of Intents in K. That is, for $1 < l < n$, we make the following approximation:

$$P_{i,f} \approx \tilde{P}\left(\bigwedge_{k=l+1}^{n} I_k \subseteq f_k | \bigwedge_{k=1}^{l} \hat{I}_k = i_k\right),$$

where $\tilde{P}$ denote an empirical probability, computed over the set K of precise Intents. For any precise Intent in K, the real values I_k and the observed values $\hat{I}_k$ are the same, for any k in $\{1, \ldots, n\}$, which gives us:

$$P_{i,f} \approx \tilde{P}\left(\bigwedge_{k=l+1}^{n} \hat{I}_k \subseteq f_k | \bigwedge_{k=1}^{l} \hat{I}_k = i_k\right). \quad (5)$$

This is the relative frequency of precise Intents with fields matching $f_{l+1}, \ldots, f_n$ among all precise Intents with field values $i_1, \ldots, i_l$.

When no field is known ($l = 0$), we use the marginal empirical probability that a precise Intent in K matches Intent Filter f:

$$P_{i,f} \approx \tilde{P}\left(\bigwedge_{k=1}^{n} \hat{I}_k \subseteq f_k\right),$$

Let us return to link $L_{3,1}$ from Figure 3. Set K consists of precise Intents (1), (2) and (4). Equation (5) yields:

$$\begin{aligned} P_{3,1} \approx \tilde{P}(&\hat{I}_{action} \subseteq \{\texttt{VIEW}\} \wedge \hat{I}_{scheme} \subseteq \{\texttt{geo}\} \\ &|\hat{I}_{categories} = \{\texttt{DEFAULT}\}) = \frac{1}{2} = 0.5, \end{aligned} \quad (6)$$

since there are two precise Intents ((1) and (4)) with category $\{\texttt{DEFAULT}\}$, but one of these two Intents (i.e., Intent (1)) has an action compatible with $\{\texttt{VIEW}\}$ and a *scheme* included in $\{\texttt{geo}\}$.

4.4 Explicit Intents

In order to accurately infer the behavior of explicit Intents, we need to utilize domain-specific knowledge of their typical usage. Explicit Intents are mostly used as a way to transition between screens within the same application. In other words they are used to switch between Activity components. Since an explicit Intent requires by definition that the target component name and application package be known, they cannot easily be used to start components across applications. This is because developers do not know the structure of other applications on a given device. For example, in Figure 3, the developer of the restaurant search application uses implicit Intents to address the map application because she cannot anticipate that the map application contains activity MapActivity. Additionally, except for a few system applications, developers cannot safely assume that a given application is available on a device. Finally, the names of application components are highly specific to each application, therefore we cannot get empirical data that can be reused across applications. For example, the map application has a component named MapActivity, which is very specific to this single application. It cannot be used as training data to estimate the potential targets of explicit Intents in other applications. Thus, we cannot simply apply Theorem 2 for explicit Intents.

In order to model explicit Intents, we introduce random variables I_p and I_c to be the real application package and component targeted by an explicit Intent. We model the name of the application that is sending a given Intent by I_a. We use random variables $\hat{I}_p$ and $\hat{I}_c$ to model the corresponding observed values. Finally, we denote by f_p and f_c the application package and component name of the receiving Filter[3].

Definition 2. *The probability that a link between an explicit Intent i and a matching Filter f is a true positive is:*

$$P_{i,f} = P\left(I_p \subseteq f_p \wedge I_c \subseteq f_c | \hat{I}_p = i_p \wedge \hat{I}_c = i_c \wedge I_a = i_a\right).$$

[3] Recall from Section 3.3 that for simplicity we consider all components to be protected by an Intent Filter, even when such a Filter is not actually declared by the application.

In other words, this is the probability that the target package and component of i match f given that the statically inferred target package and application of i are i_p and i_c and that the application sending i is i_a. This definition is the same as Equation (4), except that we have explicitly added the condition that the application that is sending Intent i is i_a. Note that i_a is always precisely known. In our model, we make the following two simplifying assumptions.

(A2) All applications have the same likelihood of being targeted by or sending explicit inter-application ICC. Overall, we expect explicit inter-application ICC to be a very rare edge case, therefore the probability of explicit inter-application ICC is always very close to 0. This is confirmed by experimental results in Section 5.4.

(A3) When several components within a single application may receive an explicit Intent, we assume that they may receive it with equal probability. All declared components are usually used in a given application. There may obviously be components that are called more often than others, but in most cases we do not expect a significant difference between any two components. Thus this yields a reasonable first approximation.

In order to model the likelihood of intra-application ICC, we let I_n denote the event that an Intent has an intra-application target. The following theorem will be used to infer $P_{i,f}$ in the case of explicit Intents. Recall from Section 4.3 that C denotes the set of constant string fields. The intuition behind the theorem is that if the target package is unambiguously known, then $P_{i,f}$ is determined by the fact that by Assumption (A3) components of a given application are equally likely to be targeted. On the other hand, if the target package is not known, then we use both Assumption (A2) and (A3) to estimate $P_{i,f}$ by considering that all matching components of all applications are equally likely to be targeted. Additionally, $P_{i,f}$ is weighted by the probability that I_n is true.

Theorem 3. *Let $i = (i_p, i_c, i_a)$ be an explicit Intent value as inferred using static analysis and $f = (f_p, f_c)$ a matching Intent Filter. Let π be the number of components that have a type compatible with i in application i_a. We also assume that there are E applications $\{i_{a_1}, \ldots, i_{a_E}\}$ other than i_a, with components that may receive i. Let $\{\pi_1, \ldots, \pi_E\}$ be the number of components compatible with i for each of these applications. Then:*

1. *If $i_p \in C$ (the target package is known), then:*

$$P_{i,f} = \begin{cases} \frac{1}{\pi} & \text{if } f_p = i_a \text{ (} i \text{ and } f \text{ are in the same application)} \\ \frac{1}{\pi_k} & \text{if } f_p = i_{a_k} \text{ for any } k \in \{1, \ldots, E\} \end{cases}$$

2. *If $i_p \notin C$ (the target package is not known), then:*

$$P_{i,f} = \begin{cases} \frac{1}{\pi} P(I_n) & \text{if } f_p = i_a \text{ (} i \text{ and } f \text{ are in the same application)} \\ \frac{1}{E\pi_k} P(\bar{I_n}) & \text{if } f_p = i_{a_k} \text{ for any } k \in \{1, \ldots, E\} \end{cases}$$

Interested readers may refer to Appendix A.2 for the proof of this theorem. In order to tackle case 2, we simply estimate $P(I_n)$ using the set K of precise Intents by counting the relative frequency of inter-application and intra-application explicit ICC. An interesting particular case is when $i_c \in C$. Then we have $\pi = 1$ for the case $f_p = i_a$ and $\pi_k = 1$ for the case $f_p = i_{a_k}$. Thus in the first case $P_{i,f} = 1$ and in the second case we have:

$$P_{i,f} = \begin{cases} P(I_n) & \text{if } f_p = i_a \\ \frac{1}{E} P(\bar{I_n}) & \text{otherwise} \end{cases}$$

Let us consider links $L_{5,1}, L_{5,2}, L_{5,3}, L_{5,4}$ and $L_{5,5}$ from Figure 3. They all have explicit Intent (5) as a source, for which $i_p \notin C$ and $i_c \notin C$. Using the notations from Theorem 3, we have $E = 4$ because there are 4 applications (excluding `rest_app`) that may receive Intent (5). For each of these applications, $\pi_k = 1$, since they all have a single compatible component. On the other hand, $\pi = 2$. The set of precise explicit Intents is reduced to Intent (2), which has an intra-application target. As a result, the empirical probability $\tilde{P}(I_n)$ of having intra-application explicit ICC is 1 and $\tilde{P}(\bar{I_n}) = 0$[4]. This allows us to calculate:

$$P_{5,1} = P_{5,2} = P_{5,3} = P_{5,4} = \frac{1}{E\pi_k} \tilde{P}(\bar{I_n}) = \frac{1}{4 \cdot 2} \cdot 0 = 0 \quad (7)$$

$$P_{5,5} = \frac{1}{\pi} \tilde{P}(I_n) = \frac{1}{2} \cdot 1 = 0.5 \quad (8)$$

Links $L_{i,f}$	Probabilities $P_{i,f}$
$L_{1,1}, L_{1,2}, L_{4,1}, L_{2,6}$	1
$L_{3,1}, L_{3,2}, L_{3,3}, L_{5,5}$	0.5
$L_{3,4}, L_{5,1}, L_{5,2}, L_{5,3}, L_{5,4}$	0

Table 1. Values of $P_{i,f}$ obtained for the links in Figure 3.

4.5 Triage of ICC Links

When $P_{i,f}$ has been computed for all Intents i and matching Filters f, we perform triage of all links by prioritizing links with the largest values of $P_{i,f}$. That is because $P_{i,f}$ represents the likelihood of a link being a true positive.

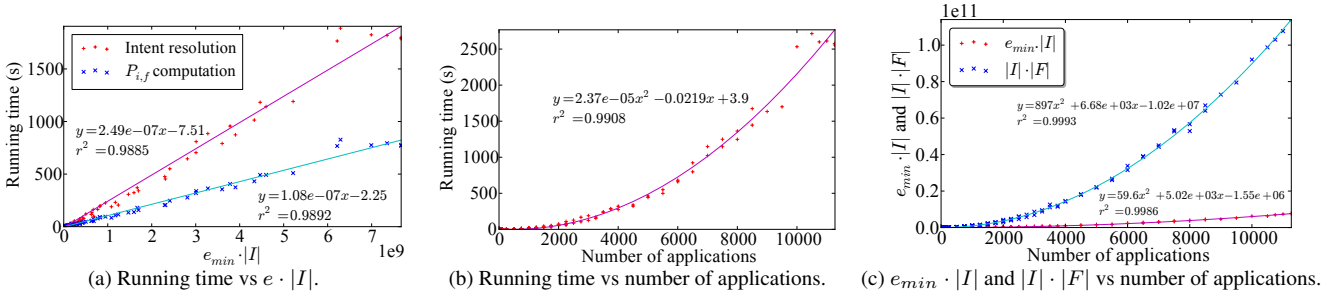
Table 1 shows the final ranking with the values of $P_{i,f}$ for the links from Figure 3. Links $L_{1,2}$, $L_{1,2}$, $L_{4,1}$ and $L_{2,6}$ have the highest priority. Their $P_{i,f}$ values are computed trivially since the fields of the corresponding source Intents are unambiguously known. Links $L_{3,1}$, $L_{3,2}$ and $L_{3,3}$ have lower priority, as the corresponding Intent fields are imprecise. Probability values $P_{3,2}$ and $P_{3,3}$ are computed in a way similar to $P_{3,1}$ (see Equation (6)). Recall that $P_{5,5}$ was computed in Equation (8). Finally, the lowest priority is assigned to links $L_{3,4}$, $L_{5,1}$, $L_{5,2}$, $L_{5,3}$ and $L_{5,4}$, since they are very likely false positives. Probability $P_{3,4}$ is computed in a way similar to Equation (6) and probabilities $P_{5,1}$, $P_{5,2}$, $P_{5,3}$ and $P_{5,4}$ were calculated in Equation (7). Note that since they are very likely false positives, simply discarding them would be a reasonable strategy. It is worth noting that the real links to the spy application have a high probability value ($P_{1,2} = 1$ and $P_{3,2} = 0.5$), thus in our analysis scenario they would be analyzed with high priority, quickly leading to the detection of the information leak shown in Figure 4.

5. Evaluation

We have implemented the concepts introduced in this paper in a tool called PRIMO (PRobabilistic ICC MOdeling). Our experimental data set initially included 10,500 applications selected at random from a corpus of over 453,525 applications downloaded from the Google Play store between January and September 2013[5] [13]. Additionally, we used 1,247 known malicious applications [50] and 162 applications preinstalled on a Samsung Galaxy Note. The latter is useful, as many applications send ICC messages that are handled by preinstalled applications such as the browser or the camera application. All applications were converted to Java bytecode using Dare [36] before being analyzed with IC3 [37]. Since IC3 could not process a number of applications for several reasons (mainly errors caused by timeout and insufficient memory), the final number of applications considered for matching was 11,267. In a preprocessing step, we discarded 1.8% of the Intent values that we knew were artifacts of the static analysis process (e.g., empty values).

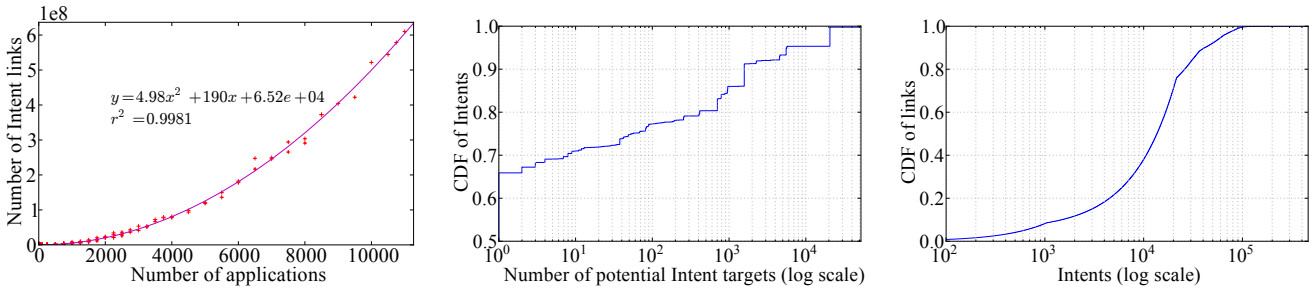
[4] While it may seem like our toy example is not representative, we show in Section 5.4 with a large data set that in real-world applications $\tilde{P}(\bar{I_n})$ is very close to 0.

[5] More recent applications utilize the same ICC primitives and would not yield different results.



(a) Running time vs $e \cdot |I|$. (b) Running time vs number of applications. (c) $e_{min} \cdot |I|$ and $|I| \cdot |F|$ vs number of applications.

Figure 6. Running time performance metrics.



(a) Intent links vs number of applications. (b) CDF of Intents depending on connectivity. (c) CDF of links.

Figure 7. Study of ICC connectivity.

The experiments described in this section attempt to answer four central questions:

1. Are the computational costs of performing Intent resolution and computing probability values feasible in practice and does the resolution time grow according to our theoretical prediction?
2. How prevalent are ICC links?
3. How accurately does PRIMO rank ICC links? In other words, does PRIMO rank false positives lower than real links?
4. What proportion of links are likely false positives?

The answers to these questions determine how useful our techniques are for performing triage of ICC links at large scale. Highlights of our study include:

- Performing Intent resolution took about 30 minutes for all 11,267 applications. Runtime grew in time $O(e_{min} \cdot |I|)$, in accordance with theoretical predictions.
- Intent resolution yielded over 636 million ICC links, 80% of which were originating in only 6% of Intents.
- The ranking computed by PRIMO is strongly correlated with the ground truth, with Goodman-Kruskal γ values consistently over 0.9721 (a value of 1 indicates perfect correlation).
- Ranking using PRIMO showed that over 95.1% of all links were likely false positives, with $P_{i,f}$ values under 0.01. Most low-rank links were potential explicit inter-application links.

5.1 Intent Matching Performance

We first evaluated the performance of the Intent resolution process described in Section 3.4. The performance of our algorithm determines to what extent the resolution process can scale to large numbers of applications to enable analysis of inter-application flows.

Figure 6(a) plots the Intent resolution computation time as a function of the product $e_{min} \cdot |I|$ of the expected number of matches for the first clause in our matching algorithm and the number of Intents. In order to compute these data points, we randomly selected subsets of our corpus of various sizes and we performed Intent resolution on these subsets. Linear fitting shows that the average running time was indeed proportional to this product ($r^2 = 0.9885$), which validates our average time complexity analysis from Section 3.5. On Figure 6(a), we also show that the computation of $P_{i,f}$ values also grew with $e_{min} \cdot |I|$. As a result, our Intent resolution process can be used even for large numbers of applications. For all $11,267$ applications in our corpus, the entire process took $2,566$ seconds, or 43 minutes, which included 30 minutes for the Intent resolution and 13 minutes for the computation of the probability values.

We then compared our algorithm with the naïve one (considering each Filter for each Intent). With a randomly selected corpus of 342 applications from the Play store, it took 696 seconds. On the other hand, our algorithm took only 29 seconds, which is a decrease of over 95%. In Figure 6(b) we plot the running time of our algorithm as a function of the number of applications considered. The running time is a quadratic function of the number of applications. Since the naïve algorithm scales very poorly we did not run it for larger data sets. However, since we trivially know that its average time complexity grows with the $|I| \cdot |F|$ product whereas our algorithm grows with $e_{min} \cdot |I|$, in Figure 6(c) we show how these two products grow as the number of applications grows. As we would expect, both show quadratic growth, however $e_{min} \cdot |I|$ grows significantly more slowly than $|I| \cdot |F|$. The benefit of using our matching algorithm rather than the naïve one is clear.

5.2 ICC Connectivity

In Figure 7(a), we show how the number of potential Intent links grows with the number of applications. With all 11,267 applications in our sample, there were 636,493,285 ICC links. The fact that the number of links explodes as the number of applications increases is the motivation for the triage work presented in this paper. Indeed, it would be very difficult to consider all these links at the scale of an application market. Therefore it is necessary when performing any inter-application analysis to only consider the links that are the most likely to actually be feasible.

In the remainder of this section, we report the results of performing Intent resolution and link triage in our entire data set of 11,267 applications. There were a total of 126,981 components, which declared 81,787 Intent Filters for receiving implicit

Field	Action	Categories	Scheme	Host	Path	Port	Type	Package (implicit)	Class	Package (explicit)
Imprecise	26%	1%	85%	96%	95%	0%	88%	98%	98%	61%
Partially precise	5%	0%	0%	1%	2%	0%	0%	0%	1%	0%

Table 2. Ratio of imprecision in Intent fields. Statistics only include Intent values where at least one field is not precise. They only take into account cases where the fields are defined (in other words, a null field is not counted). For the *package* field, we distinguish between implicit and explicit Intents, since both types may contain it. We also distinguish between the case where a field is completely unknown (that is, it is `.*`) in the second row and the case where the field is partially known (it is `.*` concatenated with a constant string) in the third row.

Intents. There were 546,073 different Intent values, 336,645 (62%) of which were explicit and 209,428 (38%) of which were implicit. There were 40,275 URI values and only 872 Content Providers. Of all Intent values, 20% had at least one field that was not precise. Looking more closely at the distribution of the 636 million links, in Figure 7(b) we plot the cumulative distribution function (CDF) of Intents as a function of the number of links in which each Intent is involved. Most Intents in our sample had a low number of potential links, with 66% having only one potential recipient and 77% having less than 100 potential targets. The maximum number of potential targets for a single Intent was 52,079. On the other hand, a small portion of Intents had over 1,000 links. Figure 7(c) shows the CDF of the links as a function of the number of Intents considered. As expected from the previous figure, most links were caused by a small number of Intents. In fact, only 6% of all Intents accounted for 80% of all ICC links.

5.3 Validation of the Probabilistic Model

In order to verify that our probabilistic model yields expected results, we started by verifying the computed probabilities for 40 links. We confirmed that the results were consistent with manually-inferred values. We then performed k-fold cross-validation of our probabilistic approach. More specifically, we first separated Intents that only had precise fields (set K in Section 4) from imprecise ones. We then extracted the distribution D_{imp} of field imprecisions in the latter set. That is, we measured the empirical frequencies of imprecisions for each field. Next, we divided set K into k parts $K_1, \ldots, K_k$ and proceeded to k iterations of the validation procedure. At each iteration i, we selected K_i to be our validation set and $\cup_{j \neq i} K_j$ to be our training set. We used the training set to extract empirical probabilities of Intent field patterns as needed for the ranking procedure described in Section 4.

To get a basis for comparison, links computed from the precise Intents in our validation set gave us the ground truth. We then altered these Intents in order to obtain imprecise ones. The imprecisions we introduced were based on imprecision frequencies observed in the wild, in order to realistically simulate real data. Then we computed links again with the altered Intents in our validation set and compared the result with the ground truth. Through this procedure, we obtained a sound comparison between our computed links and the ground truth, while simulating a realistic distribution of imprecisions. Table 2 shows the ratios of imprecise fields in Intents for which at least one field is not completely known, excluding null field values. The imprecisions introduced in the validation sample were based on this distribution. For example, for implicit Intents in which the action field is defined (i.e., almost all of them), we replaced it in 26% of cases with `.*` and we replaced part of it with `.*` in 5% of cases. For partially replaced strings, the replaced part was randomly chosen. We note that in imprecise Intents where data fields are defined, they are commonly a source of imprecision. This is because URI data is known to be more challenging to infer with static analysis [37].

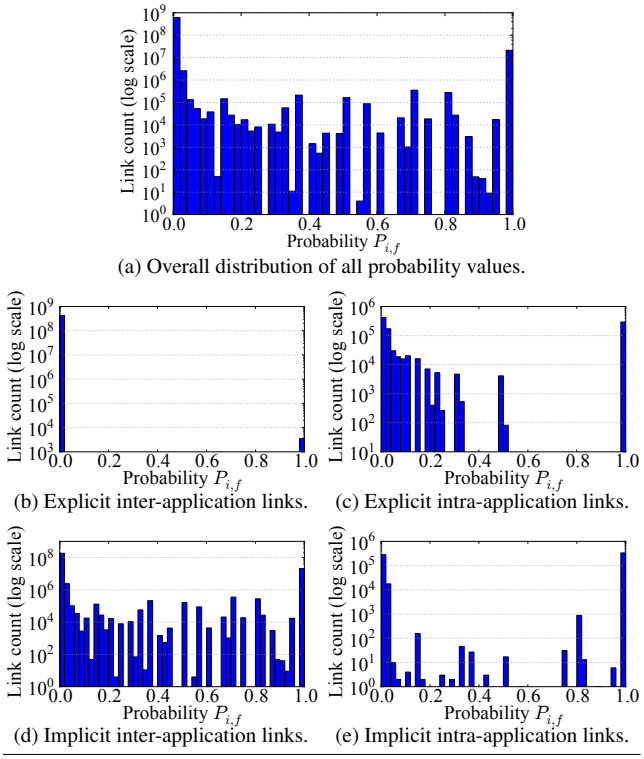
More formally, at iteration i we first computed the feasible links $\mathcal{L}_\mathcal{G}$ from Intents in K_i to the Filters in our data set. These links were known to be feasible since they were computed using precise Intents. Then we introduced imprecisions in the fields of the Intents in K_i according to distribution D_{imp}. Repeating the link computation then yielded a new set of ICC links $\mathcal{L}_\mathcal{V}$ such that $\mathcal{L}_\mathcal{G} \subset \mathcal{L}_\mathcal{V}$. That is, in addition to the feasible links, $\mathcal{L}_\mathcal{V}$ also comprised unfeasible links that were computed because of the imprecisions introduced in the fields of the Intents in K_i. Computing the probability values $V_L = P_{i,f}$ as described in Section 4 gave us a ranking $V_{L_1} \leq V_{L_2} \leq \ldots$ of the links in $\mathcal{L}_\mathcal{V}$. We also had the corresponding ground truth values G_L, such that for any link L in $\mathcal{L}_\mathcal{V}$, $G_L = 1$ if $L \in \mathcal{L}_\mathcal{G}$ and 0 otherwise. This gave us a ground truth ranking of the links such that any feasible link had higher priority than any unfeasible link.

The accuracy of our probabilistic model is measured by the extent to which the computed ranking of the V_L values agrees with the ground truth ranking of the G_L values. In order to measure this rank correlation, we computed the Goodman-Kruskal γ statistic [20]. A γ value of 1 implies complete correlation, while values of 0 and -1 indicate no correlation and inverse correlation, respectively. Since this coefficient is designed for ordinal categorical variables, we discretized the interval $[0, 1]$ of probabilities into segments of width 0.01. A value of γ that is close to 1 indicates that our model can accurately predict the ranking of link priorities with only limited Intent field information. We chose this statistic because it is resilient to ties in the rankings (values in the ground truth are all 0 or 1).

We performed k-fold cross validation for even values of k between 6 and 24. Since we obtained k values of γ during cross-validation, we computed the average γ value for each k. We observed that our system had very high accuracy, with all average values of γ in the [0.9721, 0.9724] interval.

Looking more closely at the validation results, we realized that the high value of γ is mostly caused by the fact that our model correctly infers that a vast majority of $P_{i,f}$ values are very low (equal to or below 0.01). Most of the cases where the ranks of the ground truth were different from our inferred ranking were in probability values different from 0 or 1. That is because, since our model is based on statistical data, the difference between close probability values is not always meaningful. For example, consider a set l of potential links with probability value 0.2 and another set of links h with probability value 0.3. Even though values in h are more likely to occur, out of a large data set, some links from set l will be real links. This implies that ranks between the ground truth and our model can be different. This problem is inherent to our statistical modeling although it may be alleviated by increasing the complexity of the model (e.g., to consider additional features).

We also observed a relatively small number of links (about 0.1% of the total number of links) for which a probability of 0 was incorrectly inferred for real links. These were mostly caused by the case where an Intent Filter field value is specific to a given application. This is common when application developers dynamically register Broadcast Receivers that only receive Intents from within the same application. Tackling this problem would require improving the model to include smoothing techniques to avoid a $P_{i,f}$ value of 0 in the case of unseen data. There were also a small number of cases where we did not have training data for a given combination of Intent field values and thus Equation (5) was undefined. In this



(a) Overall distribution of all probability values.

(b) Explicit inter-application links.

(c) Explicit intra-application links.

(d) Implicit inter-application links.

(e) Implicit intra-application links.

Figure 8. Distribution of link probabilities.

case, our implementation defaulted to a value of 0 (see Section 5.4 for more details). In this case, we could improve our results by considering additional features other than field values to compute probability values. Finally, about 0.005% of links were incorrectly inferred as true links ($P_{i,f} = 1$), even though they are not in reality possible. Looking at some examples, we realized that these errors occurred when an imprecise explicit Intent was matched with a single compatible Service component within the same application. In accordance with Theorem 3, our model yielded $P_{i,f} = P(I_n)$ with $P(I_n)$ rounded to 1 for the given training sample (recall that we discretized probability values with a 0.01 step size). The real Intent value was in fact one of the rare cases of inter-application explicit ICC. In future work, we will improve our model to better handle these corner cases.

5.4 Link Probabilities

Figure 8(a) shows the distribution of the computed probabilities. We can observe that a large majority of links have a low $P_{i,f}$ value and are thus likely false positives. Note that this includes 25,641 links in which Equation (5) is not defined because the combination of precise Intent fields that is considered to compute their $P_{i,f}$ value does not exist in the training data (and thus a conditional probability does not exist for these values). In these cases, our current implementation defaults to a value of 0. We can observe that 95.11% of links have a $P_{i,f}$ value under a conservative threshold of 0.01. Looking more closely at the characteristics of these links, Figures 8(b) through (e) shows the distribution of $P_{i,f}$ for explicit, implicit, inter-application and intra-application links. We can see that a large majority of links (67.41%) is caused by inter-application explicit links. As we studied in detail in Section 4.4, such links are likely false positives. In our sample of precise Intents, the empirical probability of having an explicit inter-application link ($\tilde{P}(\bar{I}_n)$ in Section 4.4) was 0.01. Manually checking a small sample of these links confirms that they are caused by explicit Intents for which

Field	Action	Categories	Type	Scheme	Host	Path	Port
Range	3,519	57	150	69	2,913	2,990	390

Table 3. Number of different values for each implicit Intent field.

both the target package and component are statically inferred as a `.*` regular expression (that is, an unknown value).

Figures 8(d) and (e) show that $P_{i,f}$ values for implicit Intents are homogeneously spread. This is expected, as Theorem 2 takes into account the likelihood of all Intent field patterns, which is highly variable. Notice how most implicit links (32.33% of all links or more than 99.70% of implicit links) are inter-application ICC. Most of them are likely false positives as well. This indicates that the significant growth of potential links is due to the fact that many inter-application links that may not occur at runtime are created whenever an Intent value is imprecise.

6. Discussion

PRIMO is based on Intent values computed using an ICC inference tool and as such it inherits the same limitations. More specifically, it cannot tackle links created by using native code or Java reflection. Additionally, it does not estimate the probability that a message-sending code location is reached at runtime.

Fragments of the set-constraint formalism are equivalent to other formalisms (e.g., tree automata can represent a fragment of set constraints). We found that modeling the Intent-resolution logic as set constraints was very natural. However, after the formulation, we could use other techniques, based on other formalisms, to solve the resulting set constraints. We will investigate this avenue in the future.

The probabilistic model is also limited by several factors. We note that the model of implicit Intents is useful because Intent patterns are relatively predictable. Table 3 shows the range of field values as seen in all 209,428 implicit Intents. We notice that the number of possible field values is very low in comparison to the number of Intents considered. The values for the action field appear to have the largest range. However, looking more closely at the distribution of these values, we notice that most individual values are application-specific and thus are very rarely found overall. When dealing with these values, our model does not perform well, due to the lack of training data. On the other hand, the top 5 values for the action field were found in 67% of Intents where it is precisely defined. This implies that our model is in general able to recognize patterns seen in training data. A particularly common pattern involves the `VIEW` action, the `DEFAULT` category and some data type. Thus, when one of these three fields is missing from this common pattern, our model can particularly effectively infer the missing value. A number of other patterns are common, notably involving the `DIAL` and `CALL` actions, which are usually associated with the `DEFAULT` category and the `tel` data scheme.

In general, insufficient training data implies that some feasible implicit Intent patterns are not known and therefore are inferred as less likely than they actually are. Some other patterns may be inferred as more likely than they actually are depending on biases in the training data. However, we expect that the prevalence of this issue will decrease as the number of applications studied increases. Note that the model presented in this paper is not meant to be the final word on estimating Intent values. Rather, we aim to form a framework for probabilistic inference of Intent values. That is why we put the emphasis on clearly articulating all of our assumptions in Section 4. We expect that this will enable others to refine this model by relaxing some of these assumptions to potentially achieve greater accuracy. Also, additional features of the problem may be helpful in order to infer a more accurate ranking of the links.

The ideas developed in this paper are applied to Android ICC inference, but we believe that they can apply to other contexts as well. Once fields of interest are defined, other models can take both field values and relationships between fields into account. Indeed, the case of explicit Intents can generalize to the inference of relationships (e.g., equality) between fields. On the other hand, the ideas used for implicit Intents can generalize to the inference of patterns between actual field values. The assumptions we made on the distribution of values (i.e., assumptions (A2) and (A3)) may be relaxed. Instead of assuming a uniform distribution, we could also obtain probability values that are parameterized by other distributions. Similarly, our approach can apply to other contexts where distributions are more complex.

Finally, we note that alternative approaches are worth exploring in future work. In our opinion, ranking allows for resource allocation: an analyst may consider links in decreasing order of priority until resources (such as, for example, time) are exhausted. However, classification may also be performed.

7. Related Work

Probabilistic approaches have been used to assign probability values to static analysis results. They include probabilistic symbolic execution [19], abstract interpretation [12] and model checking [23]. However, these techniques do not attempt to rank existing analysis results. They could complement our analysis, since in this paper we have considered a link to be a true positive if the real value i_r of the source Intent matches the destination Filter. However, the Intent-sending location might send different possible Intent values depending on the execution path, and i_r could be the most unlikely value (even though it is precisely inferred). Several related techniques [17, 41] could help estimate the probability that each precise Intent value is sent at a given location.

Machine learning has been used to classify static analysis results in Aletheia [43], but it requires manual labeling of static analysis results, whereas our approach does not. Z-Ranking [27] also aims to help sort through analysis results by ranking them. It builds a statistical model on the hypothesis that errors in programs are sparse. It has some similarity with the way that we design our model for explicit Intents, since we assume that explicit inter-application ICC is also very scarce. However, in general ICC links are common and thus the model in Z-Ranking would not be appropriate for Intents. Both Merlin [30] and PRIMO are based on the idea of leveraging probabilities for getting more accurate results from imprecise inputs. However, the goal of Merlin is to infer information flow specifications to avoid incorrect user specifications. On the other hand, our goal is to augment static-analysis results in which imprecision is not caused by the user but by the analysis itself. In addition, the probability distributions in Merlin model analyst expectations, whereas in our work the distributions are trained with and determined by the analysis results that are known to be precise.

JSNice [39] also uses statistical methods to model program properties. More specifically, it infers variable types and names in JavaScript code. The methods developed for JSNice may be useful for inferring likely ICC values. However, in its current form it makes no attempt at estimating the values of variables. Our work is different in several ways. JSNice uses conditional random fields in an attempt to maximize the joint probability of the computed properties, which ensures that the correlations between properties to be inferred are accurately captured. On the other hand, the way our model is defined makes the implicit assumption that imprecise Intent values at different program points are not correlated with one another (since all $P_{i,f}$ values for imprecise Intents are computed independently of each other). While the model in JSNice captures a more comprehensive picture of the program by maximizing joint probabilities, we believe our model to be more scalable for our particular problem. JSNice produces 4,114 type annotations in 396 programs in at least 36 ms per program on average (depending on experimental parameters). This implies that a type annotation is inferred on average in at least 3 ms (excluding the time needed to train the model). On the other hand, we are able to assign $P_{i,f}$ values on average in slightly over 0.001 ms per link.

The idea of performing triage in Android is not new [7, 44]. MAST [7] uses Multiple Correspondence Analysis (MCA) to rank applications based on metadata features. Our approach is different and complementary in several ways. Chiefly, our approach utilizes code analysis, which MAST does not. Thus, a possible workflow could consist in using MAST first to select a number of potentially problematic applications, followed by a deeper analysis that uses the high priority ICC links computed using IC3 [37] and PRIMO.

ICC has received a lot of attention in recent years. Fuzz testing was used to investigate the robustness of Android ICC [33]. ECVDetector [47] and PCLeaks [29] were proposed to detect potential component vulnerabilities. ComDroid [10] infers properties of ICC objects and Epicc [38] and IC3 [37] compute ICC values. However, these have not addressed the problem of computing ICC links. Recent works [26, 29, 46] detect inter-component leaks in Android applications, extending intra-component information flow analysis [6]. They build ICC links for their analysis, but they use a naive matching algorithm to build the links, which prevents them from being scalable. For example, we note that [26] only involves three applications. This paper complements these tools by providing principled and efficient Intent resolution; it is a significant step in enabling them to perform market-scale ICC leak detection.

Set constraints have been widely studied [2–4, 9]. Although our problem is expressed using set constraints, these existing works do not address the problem where regular expression matching is needed. Further, since our constraints have a very specific structure (namely, the Filter attributes are on the right-hand side of almost all constraints), we are able to use a more efficient solution algorithm.

8. Conclusion

We have shown how to overlay a probabilistic model on top of static analysis results to help sift through them. We have applied this idea to static ICC analysis in mobile applications. As a prerequisite, in order to enable the principled generation of ICC links, we have introduced a formalism for ICC links based on set constraints. We have designed an efficient link resolution process, which enables ICC link generation with large sets of applications. Generating over 636 millions ICC links in a corpus of 11,267 applications took 30 minutes. Since there are typically many links due to imprecisely inferred static analysis values, we used our probabilistic model to triage ICC links. The model is easy to train, requiring no manual labeling of the static analysis results. We have shown that our triage system is effective at ranking Intent values, with over 95.1% of 636 million ICC links being likely false positives. In future work, we will apply PRIMO in large scale ICC analyses. We will also refine our probabilistic model in order to improve ranking accuracy. In particular, we will study the influence of each field on the accuracy of Intent matching. By doing so, we hope to help the community's efforts in understanding the implications of ICC. Finally, we will apply the concepts developed in this paper to areas outside ICC link triage.

Acknowledgements

This material is based upon work supported by National Science Foundation Grants No. CNS-1064900, CNS-1228700, CNS-1228620 and CNS-1219495. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National

Science Foundation. This research was also supported by a Google Faculty Research Award, the Fonds National de la Recherche (FNR), Luxembourg, under project AndroMap C13/IS/5921289, by the DFG's Priority Program 1496 "Reliably Secure Software Systems", and by the INTERFLOW project.

A. Proofs

A.1 Proof of Theorem 2

In order to prove this theorem, we need the following lemma.

Lemma 1. *Let $i = (i_1, \ldots, i_n)$ be an Intent value and define a matching Intent Filter as $f = (f_1, \ldots, f_n)$. For any $m \in \{1, \ldots, n\}$, if $\hat{I_m} \in C$, then*

$$P\left(I_m \subseteq f_m | \bigwedge_{k=1}^{n} \hat{I_k} = i_k \wedge \bigwedge_{k=m+1}^{n} I_k \subseteq i_k\right) = 1.$$

Proof (Lemma 1). Given a field value for which the actual value (that is, as intended by the application developer) is `st`, we may only observe either `st` or a regular expression $regex$ such that $\texttt{st} \in L(regex)$ as a result of the static analysis. As a result, if we observe `st`, then it implies that the actual value is also `st`. Thus, $\hat{I_m} = i_m \in C$ implies that $I_m = i_m$ and since the Intent matches the Filter, $I_m \subseteq f_m$.

For any events A, B, C, if $B \Rightarrow A$ (that is, the occurrence of B implies that A is occurring as well) , then $P(A|B \wedge C) = 1$. This can be seen by considering that $P(B \wedge C) = P(A \wedge B \wedge C) + P(\bar{A} \wedge B \wedge C) = P(A \wedge B \wedge C)$, since $\bar{A}$ and B cannot occur simultaneously. Then we have

$$P(A|B \wedge C) = \frac{P(A \wedge B \wedge C)}{P(B \wedge C)} = \frac{P(B \wedge C)}{P(B \wedge C)} = 1.$$

Using this and the fact that $\hat{I_m} \in C \Rightarrow I_m \subseteq f_m$, we have:

$$P\left(I_m \subseteq f_m | \bigwedge_{k=1}^{n} \hat{I_k} = i_k \wedge \bigwedge_{k=m+1}^{n} I_k \subseteq i_k\right) = 1.$$

□

Proof (Theorem 2). The case where all fields are known trivially results in $P_{i,f} = 1$. For $1 < l < n$, we have:

$$\begin{aligned} P_{i,f} &= P\left(\bigwedge_{k=1}^{n} I_k \subseteq f_k | \bigwedge_{k=1}^{n} \hat{I_k} = i_k\right) \\ &= \prod_{m=1}^{l} P\left(I_m \subseteq f_m | \bigwedge_{k=1}^{n} \hat{I_k} = i_k \wedge \bigwedge_{k=m+1}^{n} I_k \subseteq i_k\right) \\ &\times P\left(\bigwedge_{k=l+1}^{n} I_k \subseteq f_k | \bigwedge_{k=1}^{n} \hat{I_k} = i_k\right), \end{aligned}$$

by using the chain rule. Since we assume that for $m \in \{1, \ldots, l\}$, $\hat{I_m} \in C$, using Lemma 1 we have:

$$\begin{aligned} P_{i,f} &= P\left(\bigwedge_{k=l+1}^{n} I_k \subseteq f_k | \bigwedge_{k=1}^{n} \hat{I_k} = i_k\right) \\ &= P\left(\bigwedge_{k=l+1}^{n} I_k \subseteq f_k | \bigwedge_{k=1}^{l} \hat{I_k} = i_k \wedge \bigwedge_{k=l+1}^{n} \hat{I_k} = i_k\right), \end{aligned}$$

Let us assume that the observed values for fields $l+1$ through n and the corresponding actual values are conditionally independent given the observed values of fields 1 through l (Assumption (A1)). As a result, we have:

$$P_{i,f} = P\left(\bigwedge_{k=l+1}^{n} I_k \subseteq f_k | \bigwedge_{k=1}^{l} \hat{I_k} = i_k\right).$$

When all fields are non-constant, we consider that the actual field values are independent of the observed regular expressions (given that as a hypothesis, we restrict ourselves to matching values). That is because whether a field value is inferred as a regular expression instead of the real value does not depend on the value itself but on how the program generates and processes it, as well as the specific static analysis technique. Thus:

$$P_{i,f} = P\left(\bigwedge_{k=1}^{n} I_k \subseteq f_k\right).$$

□

A.2 Proof of Theorem 3

Proof (Theorem 3). Since we have assumed that all components of a given application compatible with i_c are equally likely to be targeted by an explicit Intent (Assumption (A3)), the first case of the theorem is immediate. Let us now consider the cases where i_p is not known. In this proof, for any events A and B, we use $P(A, B) = P(A \wedge B)$.

$$\begin{aligned} P_{i,f} &= P\left(I_p \subseteq f_p, I_c \subseteq f_c | \hat{I_p} = i_p, \hat{I_c} = i_c, I_a = i_a\right) \\ &= P\left(I_p \subseteq f_p, I_c \subseteq f_c | \hat{I_c} = i_c, I_a = i_a\right), \end{aligned}$$

by conditional independence as in Theorem 2.

$$\begin{aligned} P_{i,f} &= P\left(I_p \subseteq f_p, I_c \subseteq f_c, I_n | \hat{I_c} = i_c, I_a = i_a\right) \\ &+ P\left(I_p \subseteq f_p, I_c \subseteq f_c, \bar{I_n} | \hat{I_c} = i_c, I_a = i_a\right) \end{aligned}$$

Case A: I_n is true and $f_p = i_a$.

$$\begin{aligned} P_{i,f} &= P\left(I_p \subseteq f_p, I_c \subseteq f_c, I_n | \hat{I_c} = i_c, I_a = i_a\right) \\ &= P\left(I_c \subseteq f_c | I_p \subseteq f_p, I_n, \hat{I_c} = i_c, I_a = i_a\right) \\ &\times P\left(I_p \subseteq f_p | I_n, \hat{I_c} = i_c, I_a = i_a\right) P\left(I_n | \hat{I_c} = i_c, I_a = i_a\right) \end{aligned}$$

Knowing that I_n occurs and that $I_a = i_a$ implies that $I_p \subseteq f_p$ and thus $P\left(I_p \subseteq f_p | I_n, \hat{I_c} = i_c, I_a = i_a\right) = 1$. I_n and $\hat{I_c}$ are conditionally independent given I_a, since the observed target component name does not influence whether or not the ICC is intra-application. Thus, $P\left(I_n | \hat{I_c} = i_c, I_a = i_a\right) = P\left(I_n | I_a = i_a\right)$. Further, since we have assumed that all applications are equally likely to utilize explicit inter-application ICC (Assumption (A2)), $P\left(I_n | I_a = i_a\right) = P\left(I_n\right)$. We have assumed that within an application all π components compatible with i_c are equally likely to be targeted, thus $P\left(I_c \subseteq f_c | I_p \subseteq f_p, I_n, \hat{I_c} = i_c, I_a = i_a\right) = \frac{1}{\pi}$. Therefore:

$$P_{i,f} = \frac{1}{\pi} P\left(I_n\right)$$

Case B: I_n is false and $f_p = i_{a_k}$, for some $k \in \{1, \ldots, E\}$.

$$\begin{aligned} P_{i,f} &= P\left(I_p \subseteq f_p, I_c \subseteq f_c, \bar{I_n} | \hat{I_c} = i_c, I_a = i_a\right) \\ &= P\left(I_c \subseteq f_c | I_p \subseteq f_p, \bar{I_n}, \hat{I_c} = i_c, I_a = i_a\right) \\ &\times P\left(I_p \subseteq f_p | \bar{I_n}, \hat{I_c} = i_c, I_a = i_a\right) P\left(\bar{I_n} | \hat{I_c} = i_c, I_a = i_a\right) \end{aligned}$$

We assume that when explicit inter-application occurs, all applications $i_{a_1}, \ldots, i_{a_E}$ have the same probability of being targeted (Assumption (A2)). Thus $P\left(I_p \subseteq f_p | \bar{I_n}, \hat{I_c} = i_c, I_a = i_a\right) = \frac{1}{E}$. The rest is similar to Case A. Therefore:

$$P_{i,f} = \frac{1}{E\pi_k} P\left(\bar{I_n}\right)$$

□

References

[1] A. Aiken and E.L. Wimmers. Solving systems of set constraints. In *Logic in Computer Science, 1992. LICS '92., Proceedings of the Seventh Annual IEEE Symposium on*, pages 329–340, Jun 1992.

[2] Alexander Aiken. Set constraints: Results, applications and future directions. In *Principles and Practice of Constraint Programming*, pages 326–335. Springer, 1994.

[3] Alexander Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2-3):79–111, November 1999.

[4] Alexander Aiken, Dexter Kozen, and Ed Wimmers. Decidability of systems of set constraints with negative constraints. *Information and Computation*, 122, 1995.

[5] AppBrain. Number of available android applications. Available from `http://www.appbrain.com/stats/number-of-android-apps`.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th Conference on Programming Language Design and Implementation (PLDI)*, June 2014.

[7] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 13–24, New York, NY, USA, 2013. ACM.

[8] Patrick P.F. Chan, Lucas C.K. Hui, and S. M. Yiu. Droidchecker: Analyzing android applications for capability leak. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 125–136, New York, NY, USA, 2012. ACM.

[9] Witold Charatonik and Leszek Pacholski. Set constraints with projections. *J. ACM*, 57(4):23:1–23:37, May 2010.

[10] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.

[11] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[12] Patrick Cousot and Michael Monerau. Probabilistic abstract interpretation. In Helmut Seidl, editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*, pages 169–193. Springer Berlin Heidelberg, 2012.

[13] Matthew Dering and Patrick McDaniel. Android market reconstruction and analysis. In *Proceedings of the Military Communications Conference (MILCOM)*, 10 2014.

[14] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[15] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[16] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '14)*, 2014.

[17] A. Filieri, C.S. Pasareanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 622–631, May 2013.

[18] Wouter Gelade and Frank Neven. Succinctness of the complement and intersection of regular expressions. *ACM Transactions on Computer Logic*, 13(1):4, 2012.

[19] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 166–176, New York, NY, USA, 2012. ACM.

[20] Leo A. Goodman and William H. Kruskal. Measures of association for cross classifications. *Journal of the American Statistical Association*, 49(268):pp. 732–764, 1954.

[21] Google. Intents and intent filters. Available from `https://developer.android.com/guide/components/intents-filters.html`.

[22] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS '12*, 2012.

[23] Andrew Hinton, Marta Kwiatkowska, Gethin Norman, and David Parker. Prism: A tool for automatic verification of probabilistic systems. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer Berlin Heidelberg, 2006.

[24] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1036–1046, New York, NY, USA, 2014. ACM.

[25] B. Johnson, Yoonki Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681, May 2013.

[26] 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*, SOAP '14, pages 1–6, New York, NY, USA, 2014. ACM.

[27] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of the 10th International Symposium on Static Analysis*, SAS'03, pages 295–315, Berlin, Heidelberg, 2003. Springer-Verlag.

[28] 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 (ICSE 2015)*, 2015.

[29] Li Li, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Automatically exploiting potential component leaks in android applications. In *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2014)*. IEEE, 2014.

[30] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: Specification inference for explicit information flow problems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 75–86, New York, NY, USA, 2009. ACM.

[31] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[32] 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*, CCS '12, pages 229–240. ACM, 2012.

[33] Amiya Kumar Maji, Fahad A Arshad, Saurabh Bagchi, and Jan S Rellermeyer. An empirical study of the robustness of inter-component communication in android. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[34] Piotr Mardziel, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. Dynamic enforcement of knowledge-based security policies. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations*

Symposium, CSF '11, pages 114–128, Washington, DC, USA, 2011. IEEE Computer Society.

[35] David Monniaux. An abstract monte-carlo method for the analysis of probabilistic programs. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 93–101, New York, NY, USA, 2001. ACM.

[36] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting android applications to java bytecode. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering*, November 2012. Available from `http://siis.cse.psu.edu/dare/`.

[37] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, May 2015.

[38] 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*, 2013.

[39] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015. ACM.

[40] Ben Fox Rubin. Amazon appstore nears 400k apps on 'huge progress'. Available from `http://www.cnet.com/news/amazon-appstore-nears-400k-apps-on-huge-progress/`.

[41] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 447–458, New York, NY, USA, 2013. ACM.

[42] 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 *To appear in Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE '14)*, 2014.

[43] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 762–774, New York, NY, USA, 2014. ACM.

[44] Saksham Varma, Robert J. Walls, Brian Lynn, and Brian Neil Levine. Efficient smart phone forensics based on relevance feedback. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, pages 81–91, New York, NY, USA, 2014. ACM.

[45] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 221–233, New York, NY, USA, 2014. ACM.

[46] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 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*, CCS '14, pages 1329–1341, New York, NY, USA, 2014. ACM.

[47] Daoyuan Wu, Xiapu Luo, and Rocky KC Chang. A sink-driven approach to detecting exposed component vulnerabilities in android apps. *arXiv preprint arXiv:1405.6282*, 2014.

[48] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 623–634, New York, NY, USA, 2013. ACM.

[49] Mu Zhang and Heng Yin. Appsealer: Automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS '14)*, 2014.

[50] 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.

[51] Yajin Zhou and Xuxian Jiang. Detecting passive content leaks and pollution in android applications. In *Proceedings of the 20th Annual Symposium on Network and Distributed System Security*, 2013.