Effective Inter-Component Communication Mapping in Android with *Epicc*: An Essential Step Towards Holistic Security Analysis

Damien Octeau[1], Patrick McDaniel[1], Somesh Jha[2], Alexandre Bartel[3], Eric Bodden[4], Jacques Klein[3], and Yves Le Traon[3]

[1]*Department of Computer Science and Engineering, Pennsylvania State University*
[2]*Computer Sciences Department, University of Wisconsin,*
[3]*Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg*
[4]*EC SPRIDE, Technische Universität Darmstadt*

{octeau,mcdaniel}@cse.psu.edu, jha@cs.wisc.edu, {alexandre.bartel,jacques.klein,yves.letraon}@uni.lu, eric.bodden@ec-spride.de

Abstract

Many threats present in smartphones are the result of interactions between application components, not just artifacts of single components. However, current techniques for identifying inter-application communication are ad hoc and do not scale to large numbers of applications. In this paper, we reduce the discovery of inter-component communication (ICC) in smartphones to an instance of the Interprocedural Distributive Environment (IDE) problem, and develop a sound static analysis technique targeted to the Android platform. We apply this analysis to 1,200 applications selected from the Play store and characterize the locations and substance of their ICC. Experiments show that full specifications for ICC can be identified for over 93% of ICC locations for the applications studied. Further the analysis scales well; analysis of each application took on average 113 seconds to complete. Epicc, the resulting tool, finds ICC vulnerabilities with far fewer false positives than the next best tool. In this way, we develop a scalable vehicle to extend current security analysis to entire collections of applications as well as the interfaces they export.

1 Introduction

The rapid rise of smartphone has led to new applications and modes of communication [1]. The scale of the new software markets is breathtaking; Google's Play Store has served billions of application downloads [31] within a few years. Such advances have also come with a dark side. Users are subjected to privacy violations [11, 12] and malicious behaviors [33] from the very applications they have come to depend on. Unfortunately, for many reasons, application markets cannot provide security assurances on the applications they serve [26], and previous attempts at doing so have had limited success [27].

Past analyses of Android applications [12, 14, 15, 17, 19, 36] have largely focused on analyzing application components in isolation. Recent works have attempted to expose and analyze the interfaces provided by components to interact [6, 12], but have done so in ad hoc and imprecise ways. Conversely, this paper attempts to formally recast Inter-Component Communication (ICC) analysis to infer the locations and substance of all inter- and intra-application communication available for a target environment. This approach provides a high-fidelity means to study how components interact, which is a necessary step for a comprehensive security analysis. For example, our analysis can also be used to perform information flow analysis between application components and to identify new types of attacks, such as application collusion [5, 8], where two applications work together to compromise the privacy of the user. In general, most vulnerability analysis techniques for Android need to analyze ICC, and thus can benefit from our analysis.

Android application components interact through ICC objects – mainly *Intents*. Components can also communicate across applications, allowing developers to reuse functionality. The proposed approach identifies a *specification* for every ICC source and sink. This includes the location of the ICC entry point or exit point, the ICC Intent action, data type and category, as well as the ICC Intent key/value types and the target component name. Note that where ICC values are not fixed we infer all possible ICC values, thereby building a complete specification of the possible ways ICC can be used. The specifications are recorded in a database in flows detected by matching compatible specifications. The structure of the specifications ensures that ICC matching is efficient.

We make the following contributions in this work:

- We show how to reduce the analysis of Intent ICC to an Interprocedural Distributive Environment (IDE) problem. Such a problem can be solved efficiently using existing algorithms [32].
- We develop Epicc, a working analysis tool built on top of an existing IDE framework [3] within the Soot [34] suite, which we have made available

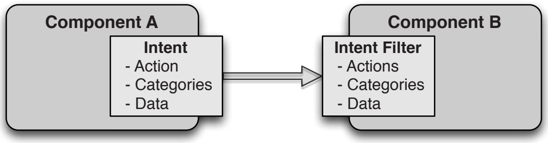


Figure 1: Implicit Intent ICC

at `http://siis.cse.psu.edu/epicc/`.

- We perform a study of ICC vulnerabilities and compare it to ComDroid [6], the current state-of-the-art. Our ICC vulnerability detection shows significantly increased precision, with ComDroid flagging 32% more code locations. While we use our tool to perform a study of some ICC vulnerabilities, our analysis can be used to address a wider variety of ICC-related vulnerabilities.
- We perform a study of ICC in 1,200 representative applications from the free section of the Google Play Store. We found that the majority of specifications were relatively narrow, most ICC objects having a single possible type. Also, key/value pairs are widely used to communicate data over ICC. Lastly, our analysis scales well, with an average analysis time of 113 seconds per application.

1.1 Android ICC

Android applications are developed in Java and compiled to a platform-specific Dalvik bytecode, and are composed of four types of components:

- An *Activity* represents a user screen. The user interface is defined through Activities.
- A *Service* allows developers to specify processing that should take place in the background.
- A *Content Provider* allows sharing of structured data within and across applications.
- A *Broadcast Receiver* is a component that receives broadcast communication objects, called *Intents*.

Intents are the primary vehicle for ICC. For example, a developer might want to start a component to display the user's current location on a map. She can create an Intent containing the user's location and send it to a component that renders the map. Developers can specify an Intent's target component (or target components) in two ways, (a) explicitly, by specifying the target's application package and class name, and (b) implicitly by setting the Intent's *action*, *category* or *data* fields.

In order for a component to be able to receive implicit Intents, *Intent Filters* have to be specified for it in the application's manifest file. Illustrated in Figure 1, Intent Filters describe the action, category or data fields of the Intents that should be delivered by the operating system to a given application component.

ICC can occur both within a single application and between different applications. In order for a component to be accessible to other applications, its *exported* attribute has to be set to `true` in the manifest file. If the *exported* attribute of a component is not defined, the OS makes the component available to other applications if an Intent Filter has been declared for it.

Intents can carry extra data in the form of key-value mappings. This data is contained in a Bundle object associated with the Intent. Intents can also carry data in the form of URIs with context-specific references to external resources or data.

Developers can restrict access to components using permissions. Permissions are generally declared in the manifest file. A component protected by a permission can only be addressed by applications that have obtained that permission. Permission requests by applications are granted by users at install time and enforced by the OS at runtime.

2 Android ICC Analysis

As highlighted above, the goal of the analysis presented in this paper is to infer specifications for each ICC source and sink in the targeted applications. These specifications detail the type, form, and data associated with the communication. We consider communication with Content Providers to be out of scope. Our analysis has the following goals:

Soundness - The analysis should generate *all* specifications for ICC that may appear at runtime. Informally, we want to guarantee that no ICC will go undetected. Our analysis was designed to be sound under the assumption that the applications use no reflection or native calls, and that the components' life cycle is modeled completely.

Precision - The previous goal implies that some generated ICC specifications may not happen at runtime ("false positives"). Precision means that we want to limit the number of cases where two components are detected as connected, even though they are not in practice. Our analysis currently does not handle URIs[1]. Since the data contained in Intents in the form of URIs is used to match Intents to target components, not using URIs as a matching criterion potentially implies more false positives. Other possible sources of imprecision include the points-to and string analyses. We empirically demonstrate analysis precision in Section 6.1.

Note that, since we do not handle URIs yet, this implies that Content Providers are out of the scope of this paper and will be handled in future work.

[1] Extending the analysis to include URIs is a straightforward exercise using the same approaches defined in the following sections. We have a working prototype and defer reporting on it to future work.

```java
private OnClickListener mMyListener =
    new OnClickListener() {
  public void onClick(View v) {
    Intent intent = new Intent();
    intent.setAction("a.b.ACTION");
    intent.addCategory("a.b.CATEGORY");
    startActivity(intent); } };
```

Figure 2: Example of implicit Intent communication

2.1 Applications

Although Android applications are developed in Java, existing Java analyses cannot handle the Android-specific ICC mechanisms. The analysis presented in this paper deals with ICC and can be used as the basis for numerous important analyses, for example:

Finding ICC vulnerabilities - Android ICC APIs are complex to use, which causes developers to commonly leave their applications vulnerable [6, 12]. Examples of ICC vulnerabilities include sending an Intent that may be intercepted by a malicious component, or exposing components to be launched by a malicious Intent. The first application of our work is in finding these vulnerabilities. We present a study of ICC vulnerabilities in Section 6.4.

Finding attacks on ICC vulnerabilities - Our analysis can go beyond ICC vulnerability detection and can be used for a holistic attack detection process. For each app. we compute entry points and exit points and systematically match them with entry and exit points of previously processed applications. Therefore, our analysis can detect applications that may exploit a given vulnerability.

Inter-component information flow analysis - We compute which data sent at an exit point can potentially be used at a receiving entry point. An information flow analysis using our ICC analysis find flows between a source in a component and a sink in a different component (possibly in a different application).

In the case where the source and sink components belong to different applications, we can detect cases of *application collusion* [5, 8]. The unique communication primitives in Android allow for a new attack model for malicious or privacy-violating application developers. Two or more applications can work together to leak private information and go undetected. For example, application A can request access to GPS location information, while application B requests access to the network. Permissions requested by each application do not seem suspicious, therefore a user might download both applications. However, in practice it is possible for A and B to work together to leak GPS location data to the network. It is almost impossible for users to anticipate this kind of breach of privacy. However, statically detecting this attack is a simple application of our ICC analysis, whereas the current state-of-the-art requires dynamic analysis and modification of the Android platform [5].

```java
public void onClick(View v) {
  Intent i = new Intent();
  i.putExtra("Balance", this.mBalance);
  if (this.mCondition) {
    i.setClassName("a.b",
        "a.b.MyClass");
  } else {
    i.setAction("a.b.ACTION");
    i.addCategory("a.b.CATEGORY");
    i = modifyIntent(i);
  }
  startActivity(i); }

public Intent modifyIntent(Intent in) {
  Intent intent = new Intent(in);
  intent.setAction("a.b.NEW_ACTION");
  intent.addCategory("a.b.NEW_CATEGORY");
  return intent; }
```

Figure 3: Intent communication: running example

2.2 Examples

Figure 2 shows a representative example of ICC programming. It defines a field that is a click listener. When activated by a click on an element, it creates Intent *intent* and sets its action and category. Finally, the *startActivity*() call takes *intent* as an argument. It causes the OS to find an activity that accepts Intents with the given action and category. When such an activity is found, it is started by the OS. If several activities meeting the action and category requirements are found, the user is asked which activity should be started.

This first example is trivial. Let us now consider the more complex example from Figure 3, which will be used throughout this paper. Let us assume that this piece of code is in a banking application. First, Intent *intent* containing private data is created. Then, if condition *this.mCondition* is true, *intent* is made explicit by targeting a specific class. Otherwise, it is made implicit. Next, an activity is started using *startActivity*(). Note that we have made the implicit Intent branch contrived to demonstrate how function calls are handled. In this example, the safe branch is the one in which *intent* targets a specific component. The other one may leak data, since it might be intercepted by an malicious Activity. We want to be able to detect that possible information leak. In other words, we want to infer the two possible Intent values at *startActivity*(). In particular, knowing the implicit value would allow us to find which applications can intercept it and to detect possible eavesdropping.

3 Connecting Application Components: Overview

Our analysis aims at connecting components, both within single applications and between different applications. For each input application $\mathcal{A}$, it outputs the following:

1. A list of entry points for $\mathcal{A}$ that may be called by com-

ponents in $\mathcal{A}$ or in other applications.

2. A list of exit points for $\mathcal{A}$ where $\mathcal{A}$ may send an Intent to another component. That component can be in $\mathcal{A}$ or in a different application. The value of Intents at each exit point is precisely determined, which allows us to accurately determine possible targets.
3. A list of links between $\mathcal{A}$'s own components and between $\mathcal{A}$'s components and other applications' components. These links are computed using 1. and 2. as well as all the previously analyzed applications.

Let us consider the example in Figure 3, which is part of our example banking application. The `startActivity(i)` instruction is an exit point for the application. Our analysis outputs the value of *i* at this instruction as well as all the possible targets. These targets can be components of our banking application itself or components of previously analyzed applications.

Figure 4 shows an overview of our component matching process. It can be divided into three main functions:

- Finding target components that can be started by other components (i.e. "entry points") and identifying criteria for a target to be activated.
- Finding characteristics of exit points, i.e. what kind of targets can be activated at these program points.
- Matching exit points with possible targets.

Given an application, we start by parsing its manifest file to extract package information, permissions used and a list of components[2] and associated intent filters (1). These components are the potential targets of ICC. We match these possible entry points with the pool of already computed exit points (2). We then add the newly computed entry points to our database of entry points (3). This database and the exit points database grow as we analyze more applications. Then we proceed with the string analysis, which identifies key API method arguments such as action strings or component names (4). Next, the main Interprocedural Distributive Environment (IDE) analysis precisely computes the values of Intent used at ICC API calls (5). It also compute the values of Intent Filters that select Intents received by dynamically registered Broadcast Receivers. These exit points are matched with entry points from the existing pool of entry points (6). The newly computed exit points are stored in the exit point database to allow for later matching (7). The values associated with dynamically registered Broadcast Receivers are used for matching with exit points in the database (8). Finally, these values are stored in the entry point database (9).

One of the inputs to our analysis is a set of class files. These classes are in Java bytecode format, since our analysis is built on top of Soot [34], an existing Java analysis framework. Android application code is distributed in a platform-specific Dalvik bytecode format that is optimized for resource-constrained devices, such as smartphones and tablets. Therefore, we use Dare [29], an existing tool that efficiently and accurately retarget Dalvik bytecode to Java bytecode. While other tools such as dex2jar[3] and ded [28] are available, Dare is currently the only formally defined one and other tools' output is sometimes not reliable.

The manifest parsing step is trivial and we use a simple string analysis (see Section 6). Also, the matching process matches exit points with entry points. It can be made efficient if properly organized in a database. Thus, we focus our description on the main IDE analysis.

It is important to distinguish between what is computed by the string analysis and by the IDE analysis. In the example from Figure 2, the string analysis computes the values of the arguments to the API calls *setAction*() and *addCategory*(). The IDE analysis, on the other hand, uses the results from the string analysis along with a model of the Android ICC API to determine the value of the Intent. In particular, in Figure 2, it determines that, at the call to *startActivity*(), Intent *intent* has action `a.b.ACTION` and category `a.b.CATEGORY`. In Figure 3, the IDE analysis tells us that *i* has two possibles values at the call to *startActivity*() and determines exactly what the two possible values are.

Reducing the Intent ICC problem to an IDE problem [32] has important advantages. Our analysis is scalable (see Section 6). Further, it is a precise analysis, in the sense that it generates few false positives (links between two components which may not communicate in reality). Thus, security analyses using our ICC analysis will not be plagued by ICC-related false positives. This precision is due to the fact that the IDE framework is flow-sensitive, inter-procedural and context-sensitive.

The flow-sensitivity means that we can distinguish Intent values between different program points. In the example from Figure 3, if Intent *i* was used for ICC right before the call to $modifyIntent()$, we would accurately capture that this value is different from the one at *startActivity*(). The context-sensitivity means that the analysis of the call to $modifyIntent()$ is sensitive to the method's calling context. If $modifyIntent()$ is called at another location with a different argument $i2$, the analysis will precisely distinguish between the values returned by the two calls. Otherwise, in a context-insensitive analysis, the return value would summarize all possible values given all contexts in which $modifyIntent()$ is called in the program. The value of *i* computed by a context-insensitive analysis would be influenced by the value of $i2$, which is not the case in reality. That would be significantly less precise, resulting in more false positives.

[2] Broadcast Receivers can be registered either statically in the manifest file or dynamically using the *registerReceiver*() methods.

[3] Available at `http://code.google.com/p/dex2jar/`.

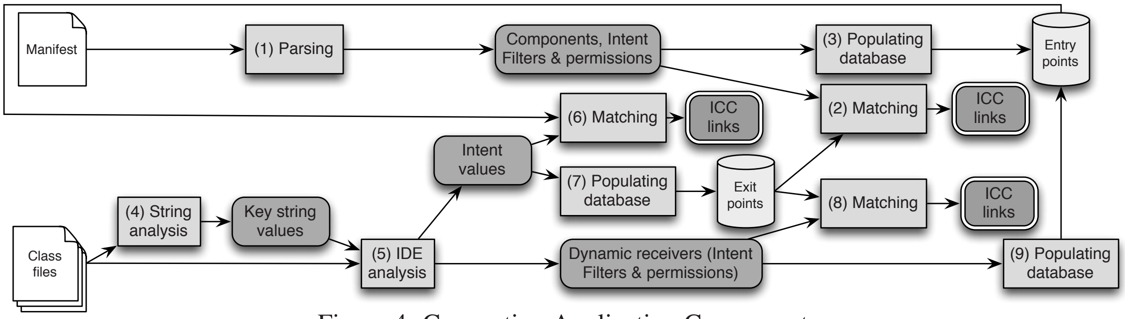


Figure 4: Connecting Application Components

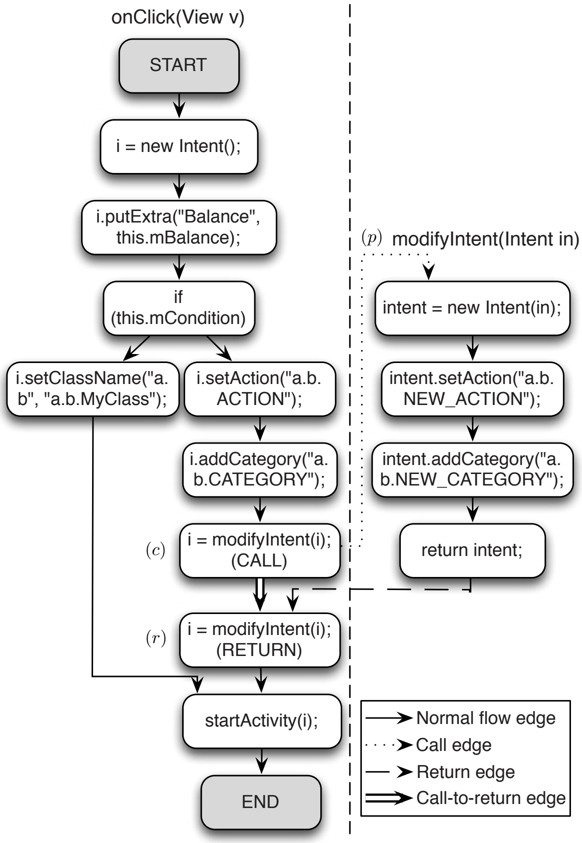

onClick(View v)
START
i = new Intent();
i.putExtra("Balance", this.mBalance);
if (this.mCondition)
i.setClassName("a.b", "a.b.MyClass");
i.setAction("a.b.ACTION");
i.addCategory("a.b.CATEGORY");
(c)
i = modifyIntent(i); (CALL)
(r)
i = modifyIntent(i); (RETURN)
startActivity(i);
END
(p)
modifyIntent(Intent in)
intent = new Intent(in);
intent.setAction("a.b.NEW_ACTION");
intent.addCategory("a.b.NEW_CATEGORY");
return intent;
Normal flow edge
Call edge
Return edge
Call-to-return edge

Figure 5: Supergraph G^* for the program from Figure 3

4 The IDE Framework: Background

The main part of our analysis is based on the IDE framework [32]. In this section, we summarize the main ideas and notations of the IDE framework. A complete description is available in [32]. The IDE framework solves a class of interprocedural data flow analysis problems. In these problems, an environment contains information at each program point. For each program idiom, environment transformers are defined and modify the environment according to semantics. The solution to this class of problems can be found efficiently.

4.1 Supergraphs

A program is represented using a *supergraph* G^*. G^* is composed of the control flow graphs of the procedures in the program. Each procedure call site is represented by two nodes, one call node representing control right before the callee is entered and one return-site node to which control flows right after exiting the callee. Figure 5 shows the supergraph of the program in Figure 3.

The nodes of a supergraph are program statements. There are four kinds of edges between these nodes. Given a call to procedure (p) with call node (c) and return-site (r), three kinds of edges are used to model the effects of the procedure call on the environment:

- A call edge between (c) and the first statement of (p).
- A return edge between the last statement of (p) and (r).
- A call-to-return edge between (c) and (r).

All other edges in the supergraph are normal intraprocedural flow edges. Informally, the call edge transfers symbols and associated values from the calling method to the callee when a symbol of interest is a procedure argument. The return edge transfers information from the return value of the callee to the environment in the calling procedure. Finally, the call-to-return edge propagates data flow information that is not affected by the callee, "in parallel" to the procedure call (e.g., local variables).

4.2 Environment transformers

Let D be a finite set of symbols (e.g., program variables). D contains at least a symbol Λ that represents the absence of a data flow fact. Let $L = (V, \sqcup)$ be a join semilattice with bottom element $\perp$, where V is a set of values[4]. An *environment e* is a function from D to L. The set of environments from D to L is denoted by $Env(D,L)$.

Operator $\sqcup$ is defined over $Env(D,L)$ as a natural extension of $\sqcup$ in semilattice L: for $e_1, e_2 \in Env(D,L)$, $e_1 \sqcup e_2$ is such that, for all $d \in D$, $(e_1 \sqcup e_2)(d) = e_1(d) \sqcup e_2(d)$.

An *environment transformer* is a function from $Env(D,L)$ to $Env(D,L)$. The algorithms from [32]

[4] A join semilattice is a partially ordered set in which any two elements have a least upper bound.

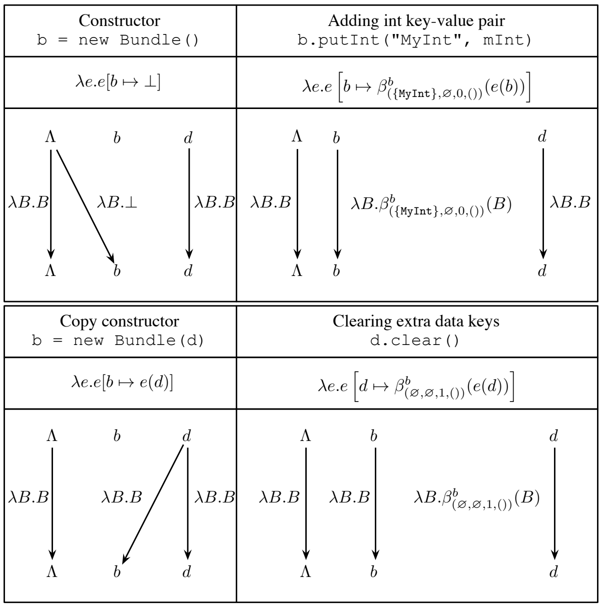

Constructor
b = new Bundle()
Adding int key-value pair
b.putInt("MyInt", mInt)
$\lambda e.e[b \mapsto \bot]$
$\lambda e.e \left[b \mapsto \beta^b_{(\{\texttt{MyInt}\},\varnothing,0,())}(e(b))\right]$
Λ
b
d
$\lambda B.B$
$\lambda B.\bot$
$\lambda B.B$
Λ
b
d
Λ
b
d
$\lambda B.B$
$\lambda B.\beta^b_{(\{\texttt{MyInt}\},\varnothing,0,())}(B)$
$\lambda B.B$
Λ
b
d
Copy constructor
b = new Bundle(d)
Clearing extra data keys
d.clear()
$\lambda e.e[b \mapsto e(d)]$
$\lambda e.e \left[d \mapsto \beta^b_{(\varnothing,\varnothing,1,())}(e(d))\right]$
Λ
b
d
$\lambda B.B$
$\lambda B.B$
$\lambda B.B$
Λ
b
d
Λ
b
d
$\lambda B.B$
$\lambda B.B$
$\lambda B.\beta^b_{(\varnothing,\varnothing,1,())}(B)$
Λ
b
d

Figure 6: Pointwise environment transformers for common Bundle operations

require that the environment transformers be *distributive*. An environment transformer t is said to be distributive if for all $e_1, e_2, \ldots \in Env(D,L)$, and $d \in D$, $(t(\sqcup_i e_i))(d) = (\sqcup_i t(e_i))(d)$. It is denoted by $t : Env(D,L) \longrightarrow_d Env(D,L)$. Environment transformers have a pointwise representation. We show an example on Figure 6. Given environment $e \in Env(D,L)$, transformer $\lambda e.e$ is the *identity*, which preserves the value of e. Given symbol $b \in D$ and value $B \in L$, $\lambda e.e[b \mapsto B]$ transforms e to an environment where all values are the same as in e, except that symbol b is associated with value B. The functions from L to L (represented next to each arrow in Figure 6) are called *micro-functions*.

The environment transformer for the copy constructor call `b = new Bundle(d)` is $\lambda e.e[b \mapsto e(d)]$. It means that the value associated with b after the instruction is the same as d's value before the instruction. In the pointwise representation, this is symbolized by an arrow between d and b with an identity function next to it.

We are trying to determine the value associated with each symbol at program points of interest, which is done by solving an Interprocedural Distributive Environment (IDE) problem. An instance IDE problem is defined as a tuple (G^*, D, L, M), where:

- $G^* = (N^*, E^*)$ is the supergraph of the application being studied.
- D is the set of symbols of interest.
- L is a join semilattice $(V, \sqcup)$ with least element $\bot$.
- M assigns distributive environment transformers to the edges of G^*, i.e. $M : E^* \longrightarrow (Env(D,L) \longrightarrow_d Env(D,L))$.

```
public ComponentName
    makeComponentName() {
  ComponentName c;
  if (this.mCondition) {
    c = new ComponentName("c.d",
        "a.b.MyClass");
  } else {
    c = new ComponentName("c.d",
        "a.b.MySecondClass"); }
  return c; }

public Bundle makeBundle(Bundle b) {
  Bundle bundle = new Bundle();
  bundle.putString("FirstName",
      this.mFirstName);
  bundle.putAll(b);
  bundle.remove("Surname");
  return bundle; }

public void onClick(View v) {
  Intent intent = new Intent();
  intent.setCompontent(makeComponentName());
  Bundle b = new Bundle();
  b.putString("Surname", this.mSurname);
  intent.putExtras(makeBundle(b));
  registerMyReceiver();
  startActivity(intent); }

public void registerMyReceiver() {
  IntentFilter f = new IntentFilter();
  f.addAction("a.b.ACTION");
  f.addCategory("a.b.CATEGORY");
  registerReceiver(new MyReceiver(),
      f, "a.b.PERMISSION", null); }
```

Figure 7: ICC objects example

Under certain conditions on the representation of micro-functions, an IDE problem can be solved in time $O(ED^3)$ [32]. For example, micro-functions should be applied in constant time. In the model we present in Section 5, we relax some of these constraints but find that the problem can still be solved efficiently in the average case. When the problem is solved, we know the value associated with each symbol at important program points.

5 Reducing Intent ICC to an IDE problem

To solve the Intent ICC problem, we need to model four different kinds of objects. First, ComponentName objects contain a package name and a class name. They can be used by explicit Intents. For example, in method *makeComponentName*() of Figure 7, a ComponentName object can take two different values depending on which branch is executed. In the first branch, it refers to class `a.b.MyClass` from application package `c.d`. In the second one, it refers to class `a.b.MySecondClass`. We want to know the possible return values of *makeComponentName*().

Second, Bundle objects store data as key-value mappings. Method *makeBundle*() of Figure 7 creates a Bundle and modifies its value. We need to find the possible return values of *makeBundle*().

Third, Intent objects are the main ICC communication objects. They contain all the data that is used to start other components. In method *onClick*() of Figure 7, the target class of *intent* is set using the return value of *makeComponentName*(). Its extra data is set to the return value of *makeBundle*(). Finally, a new Activity is started using the newly created Intent. We need to determine the value of *intent* at the `startActivity(intent)` instruction.

Fourth, IntentFilter objects are used for dynamic Broadcast Receivers. In *registerMyReceiver*() on Figure 7, an action and a category are added to IntentFilter f. Then a Broadcast Receiver of type `MyReceiver` (which we assume to be defined) is registered using method *registerReceiver*(). It receives Intents that have action `a.b.ACTION` and category `a.b.CATEGORY` and that originate from applications with permission `a.b.PERMISSION`. We want to determine the arguments to the *registerReceiver*() call. That is, we want to know that f contains action `a.b.ACTION` and category `a.b.CATEGORY`. We also want to know that the type of the Broadcast Receiver is `MyReceiver`.

In this section, we use the notations from Sagiv *et al.* [32] summarized in Section 4. We assume that string method arguments are available. We describe the string analysis used in our implementation in Section 6.

5.1 ComponentName Model

In this section, we introduce the model we use for ComponentName objects. We introduce the notion of a branch ComponentName value. It represents the value that a ComponentName object can take on a single branch, given a single possible string argument value for each method setting the ComponentName's package and class names, and in the absence of aliasing.

Definition 1. *A* branch ComponentName value *is a tuple $c = (p,k)$, where p is a package name and k is a class name.*

In method `makeComponentName()` of Figure 7, two branch ComponentName values are constructed:

$$(\texttt{c.d}, \texttt{a.b.MyClass}) \tag{1}$$

and

$$(\texttt{c.d}, \texttt{a.b.MySecondClass}) \tag{2}$$

The next definition introduces ComponentName values, which represent the possibly multiple values that a ComponentName can have at a program point. A ComponentName can take several values in different cases:

- After traversing different branches, as in method *makeComponentName*() of Figure 7.
- When a string argument can have several values at a method call.
- When an object reference is a possible alias of another local reference or an object field.
- When an object reference is a possible array element.

In the last two cases, in order to account for the possibility of a false positive in the alias analysis, we keep track of two branch ComponentName values. One considers the influence of the call on the possible alias and the other one does not.

Definition 2. *A* ComponentName value *C is a set of branch ComponentName values: $C = \{c_1, c_2, \cdots, c_m\}$. The set of ComponentName values is denoted as V_c. We define $\bot = \varnothing$ and $\top$ as the ComponentName value that is the set of all possible branch ComponentName values in the program. The operators $\cup$ and $\subseteq$ are defined as traditional set union and comparison operators: for $C_1, C_2 \in V_c$, $C_1 \subseteq C_2$ iff $C_1 \cup C_2 = C_2$. $L_c = (V_c, \cup)$ is a join semilattice.*

Note that given the definitions of $\bot$ and $\top$ as specific sets, $\cup$ and $\subseteq$ naturally apply to them. For example, for all $C \in V_c$, $\top \cup C = \top$.

In method *makeComponentName*() from Figure 7, the value of c at the return statement is

$$\begin{aligned}\{&(\texttt{c.d}, \texttt{a.b.MyClass}),\\ &(\texttt{c.d}, \texttt{a.b.MySecondClass})\}.\end{aligned} \tag{3}$$

It simply combines the values of c created in the two branches, given by Equations (1) and (2).

We define transformers from V_c to V_c that represent the influence of a statement or a sequence of statements on a ComponentName value. A pointwise branch ComponentName transformer represents the influence of a single branch, whereas a pointwise ComponentName transformer represents the influence of possibly multiple branches.

Definition 3. *A* pointwise branch ComponentName transformer *is a function $\delta^c_{(\pi,\chi)} : V_c \to V_c$, where π is a package name and χ is a class name. It is such that, for each $C \in V_c$,*

$$\delta^c_{(\pi,\chi)}(C) = \{(\pi,\chi)\}$$

Note that $\delta^c_{(\pi,\chi)}(C)$ is independent of C, because API methods for ComponentName objects systematically replace existing values for package and class names. In the example from Figure 7, the pointwise branch ComponentName transformer corresponding to the first branch is

$$\delta^c_{(\texttt{c.d}, \texttt{a.b.MyClass})}, \tag{4}$$

and the one for the second branch is

$$\delta^c_{(\texttt{c.d}, \texttt{a.b.MySecondClass})}. \tag{5}$$

Definition 4. *A* pointwise ComponentName transformer *is a function $\delta^c_{\{(\pi_1,\chi_1),\cdots,(\pi_n,\chi_n)\}} : V_c \to V_c$ such that, for each $C \in V_c$,*

$$\delta^c_{\{(\pi_1,\chi_1),\cdots,(\pi_n,\chi_n)\}}(C) = \{(\pi_1,\chi_1),\cdots,(\pi_n,\chi_n)\}$$

A pointwise ComponentName transformer summarizes the effect of multiple branches (or a single branch with multiple possible string arguments, or with possible aliasing) on a ComponentName value. That is, given the value C of a ComponentName right after statement s_i and given transformer $\delta^c_{\{(\pi_1,\chi_1),\cdots,(\pi_n,\chi_n)\}}$ that summarizes the influence of statements $s_{i+1},\cdots,s_k$ on C, $\delta^c_{\{(\pi_1,\chi_1),\cdots,(\pi_n,\chi_n)\}}(C)$ represents all the possible values of C right after s_k. In method *makeComponentName*() of Figure 7, the pointwise ComponentName transformer that models the two branches is

$$\delta^c_{\{(\texttt{c.d},\texttt{a.b.MyClass}),(\texttt{c.d},\texttt{a.b.MySecondClass})\}}. \tag{6}$$

It combines the transformers given by Equations (4) and (5). In order to understand how this transformer is applied in practice, we should mention that the algorithm to solve IDE problems initially sets values to $\perp$ [32]. Therefore, in method *makeComponentName*(), the value associated with c is initially $\perp = \varnothing$. Using Definition 4, we can easily see that if we apply the transformer given by Equation (6), we get the value given by Equation (3). This confirms that the transformer models the influence of the two branches:

$$\begin{aligned}\delta^c_{\{(\texttt{c.d},\texttt{a.b.MyClass}),(\texttt{c.d},\texttt{a.b.MySecondClass})\}}&(\perp)\\ = \{(\texttt{c.d},\texttt{a.b.MyClass})&,\\ (\texttt{c.d},\texttt{a.b.MySecondClass})&\}\end{aligned}$$

5.2 Bundle Model

The model of Bundle objects is defined similarly to the model of ComponentName objects. An additional difficulty is introduced. The data in a Bundle can be modified by adding the data in another Bundle to it, as shown in method *makeBundle*() of Figure 7. In this example, the data in Bundle b is added to the data in Bundle *bundle*. Bundle *bundle* is later modified by removing the key-value pair with key `Surname`. The issue is that when the data flow problem is being tackled, the value of b is not known. Therefore, the influence of the call to `remove("Surname")` is not known: if a key-value pair with key `Surname` is part of b, then the call removes it from *bundle*. Otherwise, it has no influence.

Our approach to deal with this object composition problem is to perform two successive analyses. In Analysis I, we use placeholders for Bundles such as b in instruction `bundle.putAll(b)`. We also record all subsequent method calls affecting *bundle*. After the problem is solved, b's key-value pairs at the `putAll(b)` method call are known, as well as the subsequent method calls. We then perform Analysis II, in which b's key-value pairs are added to *bundle*'s. The influence of the subsequent method call is precisely evaluated and finally the value of *bundle* at the return statement can be known.

5.2.1 Analysis I

In the first analysis, we consider intermediate values that contain "placeholders" for Bundle values that are not known when the problem is being solved.

Definition 5. *An* intermediate branch Bundle value *is a tuple $b_i = (E,O)$, where:*

- *E is a set of keys describing extra data.*
- *O is a tuple of two types of elements. O contains references to particular Bundle symbols at instructions where putAll() calls occur. O also contains functions from V^i_b to V^i_b, where V^i_b is the set of intermediate Bundle values defined below. These functions represent a sequence of method calls affecting a Bundle.*

The difference with previous definitions is the introduction of O, which models calls to *putAll*() as well as subsequent calls affecting the same Bundle. In method *makeBundle*() of Figure 7, at the return statement, the intermediate branch Bundle value associated with *bundle* is (E,O), where

$$E = \{\texttt{FirstName}\} \tag{7}$$

$$O = ((b,\texttt{bundle.putAll(b)}),\beta^b_{(\varnothing,\{\texttt{Surname}\},0,())}) \tag{8}$$

In O, $(\text{b},\texttt{bundle.putAll(b)})$ is a reference to variable b at instruction `bundle.putAll(b)`. $\beta^b_{(\varnothing,\texttt{Surname},0,())}$ models the *remove*() method call. It is defined below.

We just defined intermediate branch Bundle values. As we did before, we need to consider multiple branches and related issues (e.g., several possible string values):

Definition 6. *An* intermediate Bundle value *B_i is a set of intermediate branch Bundle values: $B_i = \{b_{i_1},\cdots,b_{i_m}\}$. The set of intermediate Bundle values is V^i_b. We define $\perp = \varnothing$ and $\top$ as the intermediate Bundle value that is the set of all possible intermediate branch Bundle values in the program. We define $\subseteq$ and $\cup$ as natural set comparison and union operators. They are such that, for $B_{i_1},B_{i_2} \in V^i_b$, $B_{i_1} \subseteq B_{i_2}$ iff $B_{i_1} \cup B_{i_2} = B_{i_2}$. $L^i_b = (V^i_b,\cup)$ is a join semilattice.*

In method *makeBundle*() from Figure 7, since there is only a single branch, the intermediate Bundle value associated with *bundle* at the return statement is $\{(E,O)\}$, where E and O are given by Equations (7) and (8).

Pointwise transformers are defined from V^i_b to V^i_b. Similarly to the ComponentName model, we first introduce pointwise branch Bundle transformers before defining pointwise Bundle transformers. In the definitions below, we use the $\backslash$ notation for set difference, and $\cup$ is naturally extended to tuples such that $(a_1,\cdots,a_k)\cup(a_{k+1},\cdots,a_l) = (a_1,\cdots,a_k,a_{k+1},\cdots,a_l)$.

Definition 7. *A* pointwise branch Bundle transformer *is a function $\beta^b_{(\eta^+,\eta^-,cl,\Theta)} : V^i_b \to V^i_b$, where:*

- *η^+ is a set of string keys describing extra data. It models calls to putExtra() methods.*

- *η^- is a set of string keys describing removed extra data. It represents the influence of calls to the removeExtra() method.*
- *cl takes value 1 if the Bundle data has been cleared with the clear() method and 0 otherwise.*
- *Θ is a tuple of two types of elements. It contains references to particular Bundle symbols at instructions where putAll() calls occur. It also contains functions from V_b^i to V_b^i. These functions represent a sequence of method calls affecting a Bundle.*

It is such that

$$\beta^b_{(\eta^+,\eta^-,cl,\Theta)}(\bot) = \{(\eta^+ \backslash \eta^-, \Theta)\}$$

and, for $B_i = \{(E_1, O_1), \cdots, (E_m, O_m)\}$ ($B_i \neq \bot$),

$$\beta^b_{(\eta^+,\eta^-,cl,\Theta)}(B_i) = \{(E'_1, O'_1), \cdots, (E'_m, O'_m)\}$$

where, for each j from 1 to m:

$$E'_j = \begin{cases} \eta^+ \backslash \eta^- & \textit{if } cl = 1 \\ (E_j \cup \eta^+) \backslash \eta^- & \textit{if } cl = 0 \textit{ and } O_j = \varnothing \\ E_j & \textit{otherwise} \end{cases}$$

$$O'_j = \begin{cases} \Theta & \textit{if } cl = 1 \textit{ or } O_j = \varnothing \\ O_j \cup \left(\beta^b_{(\eta^+,\eta^-,0,\Theta)}\right) & \textit{otherwise} \end{cases}$$

The definition of E'_j accounts for several possible cases:

- If the Bundle data has been cleared (i.e., $cl = 1$), then we discard any data contained in E_j. This leads to value $\eta^+ \backslash \eta^-$ for E'_j: we only keep the values η^+ that were added to the Bundle data and remove the values η^- that were removed from it.
- If the Bundle has not been cleared, then there are two possible cases: either no reference to another Bundle has been previously recorded (i.e., $O_j = \varnothing$), or such a reference has been recorded to model a call to *putAll*(). In the first case, we simply take the union of the original set E_j and the set η^+ of added values, and subtract the set η^- of removed values. This explain the $(E_j \cup \eta^+) \backslash \eta^-$ value. In the second case, a call to *putAll*() has been detected, which means that any further method call adding or removing data has to be added to set O_j instead of E_j. Therefore in this case $E'_j = E_j$.

The definition of O'_j considers several cases:

- If the Bundle data has been cleared, then the previous value of O_j is irrelevant and we set $O'_j = \Theta$. Also, if O_j is empty, then we can also just set O'_j to Θ (which may or may not be empty).
- Otherwise, the Bundle data has not been cleared ($cl = 0$) and a call to *putAll*() has been detected ($O_j \neq \varnothing$). Then it means that the current function models method calls that happened after a call to *putAll*(). Therefore we need to record $\beta^b_{(\eta^+,\eta^-,0,\Theta)}$ in O'_j, which explains the definition $O'_j = O_j \cup (\beta^b_{(\eta^+,\eta^-,0,\Theta)})$.

For example, the pointwise branch Bundle transformer that models the influence of the method *makeBundle*() from Figure 7 is $\beta^b_{(\eta^+,\varnothing,0,\Theta)}$, where

$$\eta^+ = \{\texttt{FirstName}\} \tag{9}$$

$$\Theta = \Big((b, \texttt{bundle.putAll(b)}), \beta^b_{(\varnothing,\{\texttt{Surname}\},0,())}\Big) \tag{10}$$

Pointwise branch Bundle transformers model the influence of a single branch. In order to account for multiple branches or issues such as possible aliasing false positive, we define pointwise Bundle transformers.

Definition 8. *A* pointwise Bundle transformer *is a function*

$$\beta^b_{\{(\eta_1^+ +,\eta_1^-,cl_1,\Theta_1),\cdots,(\eta_n^+,\eta_n^-,cl_n,\Theta_n)\}} : V_b^i \to V_b^i$$

such that, for each $B_i \in V_b^i$,

$$\beta^b_{\{(\eta_1^+,\eta_1^-,cl_1,\Theta_1),\cdots,(\eta_n^+,\eta_n^-,cl_n,\Theta_n)\}}(B_i) = \beta^b_{(\eta_1^+,\eta_1^-,cl_1,\Theta_1)}(B_i) \cup \cdots \cup \beta^b_{(\eta_n^+,\eta_n^-,cl_n,\Theta_n)}(B_i)$$

For example, method *makeBundle*() from Figure 7 only has a single branch, thus the pointwise Bundle transformer that models it is simply $\beta^b_{\{(\eta^+,\varnothing,0,\Theta)\}}$, where η^+ and Θ are given in Equations (9) and (10). As we did for the ComponentName value example, we can confirm using Definitions 7 and 8 that $\beta^b_{\{(\eta^+,\varnothing,0,\Theta)\}}(\bot) = \{(E, O)\}$, where E and O are given by Equations (7) and (8).

5.2.2 Analysis II

After Analysis I has been performed, the values of the Bundles used in placeholders in intermediate Bundle values are known. Ultimately, we want to obtain branch Bundle values and finally Bundle values:

Definition 9. *A* branch Bundle value *b is a set E of string keys describing extra data.*

Definition 10. *A* Bundle value *B is a set of branch Bundle values: $B = \{b_1, \cdots, b_m\}$.*

Since the values of the referenced Bundles are known, we can integrate them into the Bundle values referring to them. Then the influence of the subsequent method calls that have been recorded can precisely be known.

Let us consider the example of *makeBundle*() from Figure 7. After Analysis I has been performed, we know that the intermediate value of *bundle* at the return statement is $\{(E, O)\}$, where

$$E = \{\texttt{FirstName}\}$$

$$O = \Big((b, \texttt{bundle.putAll(b)}), \beta^b_{(\varnothing,\{\texttt{Surname}\},0,())}\Big)$$

We consider all elements of O in order. As the first element of O is `(b,bundle.putAll(b))`, we integrate b's value into *bundle*. From Analysis I, we know that the value of b at instruction `bundle.putAll(b)` is $\{\{\texttt{Surname}\},\varnothing\}$. Thus, E becomes $\{\texttt{FirstName}, \texttt{Surname}\}$. The next element of O is $\beta^{b}_{(\varnothing,\{\texttt{Surname}\},0,())}$. This means that we have to remove key `Surname` from E. The final value of E is therefore $\{\texttt{FirstName}\}$. Thus, the Bundle value associated with *bundle* at the return statement is $\{\{\texttt{FirstName}\}\}$.

Note that the referenced Bundle can also make references to other Bundles. In that case, we perform the resolution for the referenced Bundles first. There can be an arbitrary number of levels of indirection. Analysis II is iterated until a fix-point is reached.

5.3 Intent and IntentFilter Models

The Intent model is defined similarly to the Bundle model, which includes object composition. In method $onClick()$ of Figure 7, the target of Intent *intent* is set using a ComponentName object and its extra data is set with a Bundle. Because of this object composition, finding the Intent value also involves two analyses similar to the ones performed for Bundles. First, intermediate Intent values with placeholders for referenced ComponentName and Bundle objects are found. Second, the referenced objects' values are integrated into *intent*'s value.

Similarly to the Bundle model, we define intermediate branch Intent values and intermediate Intent values. The set of intermediate Intent values is V_i^i and we define a lattice $L_i^i = (V_i^i, \cup)$ as we did for L_b^i. We also define pointwise branch Intent transformers and pointwise Intent transformers. For example, in method $onClick()$ of Figure 7, the final intermediate value for *intent* simply has placeholders for a ComponentName and a Bundle value. Other fields, such as action and categories, are empty. The ComponentName and Bundle values are computed using the models presented in Sections 5.1 and 5.2. Finally, we define branch Intent values and Intent values, which are output by the second analysis. The final value for *intent* after the second analysis precisely contains the two possible targets (`a.b.MyClass` and `a.b.MySecondClass` in package `c.d`) and extra data key `FirstName`. For conciseness, and given the strong similarities with the Bundle model, we do not include a full description of the Intent model here.

In order to analyze dynamic Broadcast Receivers, we model IntentFilter objects. Modeling IntentFilters is similar to modeling Intents, except that IntentFilters do not involve object composition. That is because IntentFilters do not have methods taking other IntentFilters as argument, except for a copy constructor. Thus, their analysis is simpler and involves a single step. Similarly to what we did for other ICC models, we define branch IntentFilter values, IntentFilter values, pointwise branch IntentFilter transformers and pointwise IntentFilter transformers. In particular, we define lattice $L_f = (V_f, \cup)$, where V_f is the set of IntentFilter values. In method $onClick()$ from Figure 7, the final value of f contains action `a.b.ACTION` and category `a.b.CATEGORY`. Given the similarity of the IntentFilter model with previous models, we do not include a complete description.

5.4 Casting as an IDE Problem

These definitions allow us to define environment transformers for our problem. Given environment $e \in Env(D,L)$, environment transformer $\lambda e.e$ is the *identity*, which does not change the value of e. Given Intent i and Intent value I, $\lambda e.e[i \mapsto I]$ transforms e to an environment where all values are the same as in e, except that Intent i is associated with value I.

We define an environment transformer for each API method call. Each of these environment transformers uses the pointwise environment transformers defined in Sections 5.1, 5.2 and 5.3. It precisely describes the influence of a method call on the value associated with each of the symbols in D.

Figure 6 shows some environment transformers and their pointwise representation. The first one is a constructor invocation, which sets the value corresponding to b to $\bot$. The second one adds an integer to the key-value pairs in Bundle b's extra data, which is represented by environment transformer

$$\lambda e.e\left[b \mapsto \beta^{b}_{(\{\texttt{MyInt}\},\varnothing,0,())}\left(e(b)\right)\right].$$

It means that the environment stays the same, except that the value associated with b becomes $\beta^{b}_{(\{\texttt{MyInt}\},\varnothing,0,())}(e(b))$, with $e(b)$ being the value previously associated with b in environment e. The pointwise transformer for b is

$$\beta^{b}_{(\{\texttt{MyInt}\},\varnothing,0,())},$$

which we denote by

$$\lambda B.\beta^{b}_{(\{\texttt{MyInt}\},\varnothing,0,())}(B)$$

on Figure 6 for consistency with the other pointwise transformers. It simply adds key `MyInt` to the set of data keys. The next transformer is for a copy constructor, where the value associated with d is assigned to the value associated with b. The last transformer clears the data keys associated with d.

Trivially, these environment transformers are distributive. Therefore, the following proposition holds.

Proposition 1. *Let G^* be the supergraph of an Android application. Let D_c, D_b D_i and D_f be the sets of ComponentName, Bundle and Intent variables, respectively, to which we add the special symbol Λ*[5]*. Let L_c, L_b^i, L_i^i and L_f be the lattices defined above.*

[5] Recall from Section 4.2 that Λ symbolizes the absence of a data flow fact.

Let M_c, M_b, M_i and M_f be the corresponding assignments of distributive environment transformers. Then (G^*, D_c, L_c, M_c), (G^*, D_b, L_b^i, M_b^i), (G^*, D_b, L_i^i, M_i^i) and (G^*, D_i, L_f, M_f) are IDE problems.

It follows from this proposition that we can use the algorithm from [32] to solve the Intent ICC problem.

The original IDE framework [32] requires that the micro-function be represented efficiently in order to achieve the time complexity of $O(ED^3)$. Our model does not meet these requirements: in particular, applying, composing, joining micro-function or testing for equality of micro-functions cannot be done in constant time. Indeed, the size of micro-functions grows with the number of branches, aliases and possible string arguments (see Equation 6 for an example with two branches). However, in practice we can find solutions to our IDE problem instances in reasonable time, as we show in Section 6.

6 Evaluation

This section describes an evaluation of the approach presented in the preceding sections, and briefly characterizes the use of ICC in Android applications. We also present a study of potential ICC vulnerabilities. Our implementation is called Epicc (Efficient and Precise ICC) and is available at `http://siis.cse.psu.edu/epicc/`. It is built on Heros [3], an IDE framework within Soot [34]. We also provide the version of Soot that we modified to handle pathological cases encountered with retargeted code.

In order to compute string arguments, we use a simple analysis traversing the interprocedural control flow graph of the application. The traversal starts at the call site and looks for constant assignments to the call arguments. If a string argument cannot be determined, we conservatively assume that the argument can be any string. As we show in Section 6.1, in many cases we are able to find precise string arguments. More complex analyses can be used if more precision is desired [7].

For points-to analysis and call graph construction, we use Spark [24], which is part of Soot. It performs a flow-sensitive, context-insensitive analysis. We approximate the call graph in components with multiple entry points. In order to generate a call graph of an Android application, we currently use a "wrapper" as an entry point. This wrapper calls each class entry point once, which may under-approximate what happens at runtime. This impacts a specification only if an ICC field (e.g., Intent) is modified in a way that depends on the runtime execution order of class entry points. Generally, if we assume that our model of components' life cycle is complete and if the application does not use native calls or reflection, then our results are sound.

The analysis presented in this section is performed on two datasets. The first *random sample* dataset contains 350 applications, 348 of which were successfully analyzed after retargeting. They were extracted from the Google Play store[6] between September 2012 and January 2013. The applications were selected at random from over 200,000 applications in our corpus. The second *popular application* dataset contains the top 25 most popular free applications from each of the 34 application categories in the Play store. The 850 selected applications were downloaded from that application store on January 30, 2013. Of those 850 applications, 838 could be retargeted and processed and were used in the experiments below. The 14 applications which were not analyzed were pathological cases where retargeting yielded code which could not be analyzed (e.g., in some cases the Dare tool generated offsets with integer overflow errors due to excessive method sizes), or where applications could not be processed by Soot (e.g., character encoding problems).

6.1 Precision of ICC Specifications

The first set of tests evaluates the technique's *precision* with our datasets. We define the precision metric to be the percentage of source and sink locations for which a specification is identified without ambiguity. Ambiguity occurs when an ICC API method argument cannot be determined. These arguments are mainly strings of characters, which may be generated at runtime. In some cases, runtime context determines string values, which implies that our analysis cannot statically find them.

Recall the various forms of ICC. Explicit ICC identifies the communication sink by specifying the target's package and class name. Conversely, implicit ICC identifies the sink through action, category, and/or data fields. Further, a mixed ICC occurs when a source or sink can take on explicit or implicit ICC values depending on the runtime context. Finally, the dynamic receiver ICC occurs when a sink binds to an ICC type through runtime context (e.g., Broadcast Receivers which identify the Intent Filter types when being registered). We seek to determine precise ICC specifications, where all fields of Intents or Intent Filters are known without ambiguity.

As shown in Table 1, with respect to the random sample corpus, we were able to provide unambiguous specifications for over 91% of the 7,835 ICC locations in the 348 applications. Explicit ICC was precisely analyzed more frequently (≈98%) than implicit ICC (≈88%). The remaining 7% of ICC containing mixed and dynamic receivers proved to be more difficult, where the precision rates are much lower than others. This is likely due to the fact that dynamic receivers involve finding more data

[6] Available at `https://play.google.com/store/apps`.

Random Sample					
	Precise	**%**	**Imprecise**	**%**	**Total**
Explicit	3,571	97.65%	86	2.35%	3,657
Implicit	3,225	88.45%	421	11.55%	3,646
Mixed	28	59.57%	19	40.43%	47
Dyn. Rec.	357	73.61%	128	26.39%	485
Total	7,181	91.65%	654	8.35%	7,835

Popular					
	Precise	**%**	**Imprecise**	**%**	**Total**
Explicit	27,753	94.43%	1,637	5.57%	29,390
Implicit	23,133	93.82%	1,525	6.18%	24,658
Mixed	509	85.12%	89	14.88%	598
Dyn. Rec.	4,161	95.81%	182	4.19%	4,343
Total	55,556	94.18%	3,433	5.82%	58,989

Table 1: Precision metrics

than Intents: Intent Filters limiting access to dynamic receivers can define several actions, and receivers can be protected by a permission (which we attempt to recover).

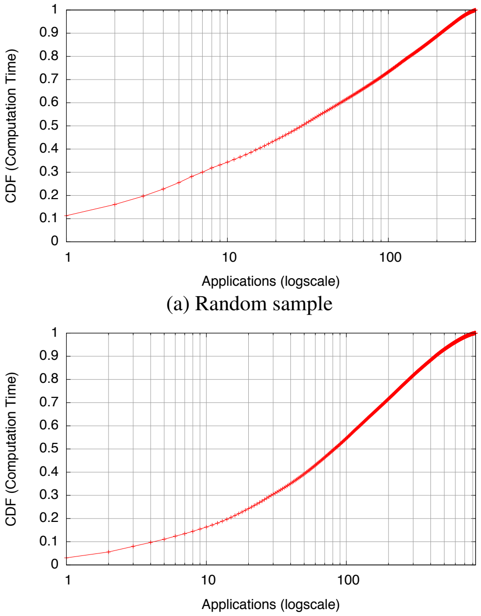
In the popular applications, we obtain a precise specification in over 94% of the 58,989 ICC locations in the 838 apps. Explicit ICC was slightly more precisely analyzed than implicit ICC. Mixed ICC is again hard to recover. This is not surprising, as mixed ICC involves different Intent values on two or more branches, which is indicative of a method more complex than most others.

A facet of the analysis not shown in the table is the number of applications for which we could identify unambiguous specifications for all ICC – called 100% precision. In the random sample, 56% of the applications could be analyzed with 100% precision, 80% of the applications with 90% precision, and 91% of the applications with 80% precision. In the popular applications, 23% could be analyzed with 100% precision, 82% could be analyzed with 90% precision and 94% with 80% precision. Note that a less-than-100% precision does not mean that the analysis failed. Rather, these are cases where runtime context determines string arguments, and thus *any* static analysis technique would fail.

6.2 Computation Costs

A second set of tests sought to ascertain the computational costs of performing the IDE analysis using Epicc. For this task we collected measurements at each stage of the analysis and computed simple statistics characterizing the costs of each task on the random sample and the popular applications.

Experiment results show that ICC analysis in this model is feasible for applications in the Google Play store. We were able to perform analysis of all 348 applications in the random sample in about 3.69 hours of compute time. On average, it took just over 38 seconds to perform analysis for a single application, with a standard deviation of 99 seconds. There was high variance in the analysis run times. A CDF (cumulative distribution function) of the analysis computation time for all 348 applications is presented in Figure 8(a). It is clear from the figure that costs were dominated by a handful of applications; the top application consumed over 11% of the time, the top 5 consumed over 25% of the total time, and the top 29 consumed over 50% of the total time. These applications are large with a high number of entry points.



(a) Random sample

(b) Popular applications

Figure 8: CDF of computation time

Analyzing the 838 popular applications took 33.58 hours, that is, 144 seconds per application. The standard deviation was 277 seconds. The average processing time is significantly higher than for the random sample. However, this is expected, as the average application size is almost 1,500 classes, which is significantly higher than the random sample (less than 400 classes per application). This is likely related to the popularity bias: one can expect frequently downloaded applications to have fully developed features as well as more complex/numerous features, which implies a larger code base. A CDF of the computation time for all 838 applications is presented in Figure 8(b). Once again, analysis time is dominated by a few applications. The top 5 consumed over 11% of the analysis time and the top 83 (less than 10% of the sample) consumed over 50% of the analysis time.

Processing was dominated by the standard Soot processing (e.g., translating classes to an intermediate representation, performing type inference and points-to analysis, building a call graph). It consumed 75% of the processing time in the random sample and 86% in the popular applications. It was itself dominated by the translation to Soot's internal representation and by the call graph construction. The second most time-consuming task was the IDE analysis (which also includes the string analysis in our implementation). It took 15% of the processing time with the random sample and 7% with the popular one. Finally, I/O operations accounted for most of the remainder of the processing time. Loading classes took 7% of the time in the random sample and 3% in the popular one. Database operations accounted for 2% of processing for the random sample and 3% for the popular applications. Other operations (e.g., parsing manifest files) took less than 1% of processing time.

6.3 Entry/Exit Point Analysis

This section briefly characterizes the exit (source) and entry (sink) points of Android applications in our data sets. Note that this analysis is preliminary and will be extended in future work.

An exit point is a location that serves as a source for ICC; i.e., the sending of an Intent. In the random sample, our analysis found 7,350 exit points which can transmit 10,035 unique Intent values. About 92% of these exit points had a single Intent specification, with the remaining exit points being able to take on 2 or more values. In two pathological cases, we noted an exit point that could have 640 different Intent values (most likely the result of contrived control flow or multiple aliasing for an Intent value). The popular applications had 48,756 exit points, associated with 316,419 Intent values. Single Intent specifications were found in 90% of exit points. We found 10 pathological cases where an exit point was associated with 512 Intent values or more. The use of key value data was more prevalent than we initially expected, in about 36% of exit points in the random sample. Key-value data was present in Intents in 46% of exit points in the popular applications.

Our study of entry points focused on the sinks of ICC that were either dynamically registered broadcast receivers or component interfaces (exported or not) identified in the application manifest. In the random sample, we were able to identify 3,863 such entry points associated with 1,222 unique intent filters. The popular applications comprised 25,291 entry points with 11,375 Intent Filters. 1,174 components were exported (and thus available to other applications) in the random sample, 7,392 in the popular applications. Of those, only 6% (67) of the exported components were protected by a permission in the random sample and 5% (382) were protected in the popular applications. This is concerning, since the presence of unprotected components in privileged applications can lead to confused deputy [21] attacks [17].

Oddly, we also found 23 components that were exported without any Intent Filter in the random sample and 220 in the popular sample. Conversely, we found 32 cases where a component had an Intent Filter but was not exported in the random sample and 412 in the popular one. The latter indicates that developers sometimes use implicit Intents to address components within an application, which is a potential security concern, since these Intents may also be intercepted by other components. Lastly, application entry points were relatively narrow (with respect to intent types). Over 97% of the entry points received one Intent type in the random sample. Single Intent Filters were found in 94% of components protected by Intent Filters in the popular applications.

6.4 ICC Vulnerability Study

In this section, we perform a study of ICC vulnerabilities in our samples using Epicc and compare our results with ComDroid [6]. We look for the same seven vulnerabilities as in [6]. Activity and Service hijacking can occur when an Intent is sent to start an Activity or a Service without a specific target. Broadcast thefts can happen when an Intent is Broadcast without being protected by a signature or signatureOrSystem permission[7]. In all three cases, the Intent may be received by a malicious component, along with its potentially sensitive data.

Malicious Activity or Service launch and Broadcast injection are Intent spoofing vulnerabilities. They indicate that a public component is not protected with a signature or signatureOrSystem permission. It may be started by malicious components. These vulnerabilities can lead to permission leakage [17, 19, 25].

Finally, some Intent Broadcasts can only be sent by the operating system, as indicated by their action field. Broadcast Receivers can register to receive them by specifying Intent Filters with the appropriate action. However, these public components can still be addressed directly by explicit Intents. That is why the target Receivers should check the action field of the received Intent to make sure that it was sent by the system.

Table 2 shows the results of the study for the random and the popular samples. The first line shows the number of vulnerabilities identically detected by both analyses, the second line shows vulnerabilities detected by ComDroid only and the third line shows vulnerabilities detected by Epicc only. The last two lines show the total number of vulnerabilities found by each tool. For the three unauthorized Intent receipt vulnerabilities (first three columns), both ComDroid and Epicc indicate whether the sent Intent has extra data in the form of key-value pairs, and whether the Intent has the `FLAG_GRANT_READ_URI_PERMISSION` or the `FLAG_GRANT_WRITE_URI_PERMISSION`. These flags are used in Intents which refer to Content Provider data and may allow the recipient to read or write the data [6].

[7] The signature permission protection level only allows access to a component from an application signed by the same developer. The signatureOrSystem protection level additionally allows the operating system to start the component.

Vulnerability	Activity Hijacking		Service Hijacking		Broadcast Theft		Activity Launch		Service Launch		Broadcast Injection		System Broadcast w/o action check		Total vulnerabilities	
Sample	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R	P
Identical	2,591	15,214	78	1,200	503	4,825	179	1,731	23	263	273	3,503	30	126	3,677	26,862
ComDroid only	916	7,717	78	535	218	2,854	12	169	2	18	104	1,684	3	20	1,333	12,997
Epicc only	181	2,079	3	151	23	297	4	20	0	1	4	43	77	580	292	3,171
Total ComDroid	3,507	22,931	156	1,735	721	7,679	191	1,900	25	281	377	5,187	33	146	5,010	39,859
Total Epicc	2,772	17,293	81	1,351	526	5,122	183	1,751	23	264	277	3,546	107	706	3,969	30,033

Table 2: ICC vulnerability study results for the random sample (R) and the popular applications (P)

For the presence of flags and the detection of extra data, Epicc can precisely indicate when the value of an Intent depends on the execution path. On the other hand, a ComDroid specification does not make this distinction. When Epicc and ComDroid differ for a code location, we include flags in both the “ComDroid only” and “Epicc only” rows of Table 2.

The Activity hijacking vulnerabilities found by both ComDroid and Epicc are unsurprisingly common: they represent all cases where implicit Intents are used to start Activities. Service hijacking vulnerabilities are much less prevalent, which is correlated with the fact that Services are used less often than Activities. Broadcast theft vulnerabilities are quite common as well. As previously described in Section 6.3, few exported components are protected by permissions. Therefore, the high number of malicious Activity or Service launch as well as Broadcast injection vulnerabilities is not surprising. Note the discrepancy between the number of components without permissions and the total number of these vulnerabilities. A large portion of the components not protected by permissions are Activities with the `android.intent.action.MAIN` action and the `android.intent.category.LAUNCHER` category, which indicate that these components cannot be started without direct user intervention. They are therefore not counted as potential vulnerabilities.

If we consider the first three vulnerabilities (unauthorized Intent receipt), we can see that ComDroid flags a high number of locations where Epicc differs. A manual examination of a random subset of applications shows that these differences are either false positives detected by ComDroid or cases where Epicc gives a more precise vulnerability specification. We observed that a number of code locations are detected as vulnerable by ComDroid, whereas Soot does not find them to be reachable. Epicc takes advantage from the sound and precise Soot call graph construction to output fewer false positives. Additionally, the IDE model used by Epicc can accurately keep track of differences between branches (e.g., explicit/implicit Intent or URI flags), whereas ComDroid cannot. Note that when an Intent is implicit on one branch and explicit on another, ComDroid detects it as explicit, which is a false negative. On the other hand, the IDE model correctly keeps track of the possibilities.

With a few exceptions, the ComDroid and Epicc analyses detect the same possible malicious Activity and Service launches. That is expected, since both are detected by simply parsing the manifest file. The few differences can be explained by minor implementation differences or bugs in pathological cases. The Broadcast injection vulnerability shows stronger differences, with ComDroid detecting 377 cases for the random sample and 5,187 for the popular one, whereas Epicc only finds 277 and 3,546, respectively. Some of the Broadcast injections detected by ComDroid involved dynamically registered Broadcast Receivers found in unreachable code. Once again, the call graph used by Epicc proves to be an advantage. Many other cases involve Receivers listening to protected system Broadcasts (i.e., they are protected by Intent Filters that only receive Intents sent by the system). The list of protected Broadcasts used by ComDroid is outdated, hence the false positives.

Finally, there is a significant difference in the detection of the system Broadcasts without action check, with Epicc detecting 107 vulnerabilities in the random sample and 706 in the popular one, whereas ComDroid only detects 33 and 146, respectively. The first reason for that difference is that the ComDroid list of protected Broadcasts is outdated. Another reason is an edge case, where the Soot type inference determines Receivers registered using a $registerReceiver()$ method as having type `android.content.BroadcastReceiver` (i.e., the abstract superclass of all Receivers). It occurs when several types of Receivers can reach the call to $registerReceiver()$. Since no Receiver code can be inspected, even though there may be a vulnerability, our analysis conservatively flags it as a vulnerability.

Overall, Epicc detects 34,002 potential vulnerabilities. On the other hand, ComDroid detects 44,869 potential security issues, that is, 32% more than Epicc. As detailed above, the extra flags found by ComDroid that we checked were all false positives. Further, the potential causes of unsoundness in Epicc (i.e., JNI, reflection and entry point handling) are also handled unsoundly in ComDroid. Thus, we do not expect the locations flagged by ComDroid but not by Epicc to be false negatives. The precision gain over ComDroid is significant and will help further analyses. Note that it is possible that both tools have false negatives in the presence of JNI, reflection, or when the life cycle is not properly approximated. In particular, we found that 776 out of the 838 popular applications and 237 out of 348 applications in the random sample make reflective calls. Future work will seek to quantify how often these cause false negatives in practice. We will also attempt to determine if the locations flagged by Epicc are true positives.

7 Related Work

ComDroid [6] is the work most closely related to ours. Our work aims to formalize the notions it first captured. It is different in many aspects. First, ComDroid directly analyses Dalvik bytecode, whereas we use retargeted Java bytecode. This allows us to leverage analyses integrated with Soot (e.g., call graph). Also, unlike ComDroid, our analysis is fully interprocedural and context-sensitive. Second, our ICC model is sound and more detailed, taking multiple branches and aliasing into account. Thus, as shown in Section 6.4, our ICC vulnerability study produces fewer false positives. Finally, ComDroid seeks to find potential vulnerabilities, whereas our approach enables finding attacks for vulnerabilities in existing applications. This is done by keeping a database of analysis results and matching newly analyzed applications with applications in our database. This will allow us to identify problematic application combinations.

Several kinds of application analysis have been performed for the Android platform [10]. Permission analysis infers applications properties based on the permissions requested at install time. Kirin [13] uses permissions to flag applications with potential dangerous functionality. Other methods for permission analysis have been proposed [2, 15, 16], including analyses to detect over-privileged applications [15] or malware [36].

Dynamic analysis consists in analyzing applications while they are running. TaintDroid [11] performs dynamic taint tracking on Android. It exposes widespread leakage of personal data to third parties. An extension to TaintDroid handles implicit flows [18] by monitoring and recording control flow information. TaintDroid is also used in the AppFence system [22], which actively prevents sensitive data exfiltration from mobile devices. Alternative approaches dynamically prevent some classes of privilege escalation attack through ICC [4, 9]. Dynamic analyses such as TaintDroid are limited by the way they interact with the User Interface (UI). SmartDroid [35] tackles this issue by combining static and dynamic analyses. It is able to simulate the UI to expose hidden behavior for seven malwares. As we use static analysis we do not interact with the UI: the call graph is complete and does not depend on any runtime condition.

Static analysis consists in analyzing application code to infer useful properties without running the application. Several approaches for static analysis have already been proposed for Android applications. Enck *et al.* use decompilation [28] followed by source code analysis to characterize security properties of applications [12]. Grace *et al.* perform a study of the dangers caused by 100 ad libraries found in a sample of 100,000 applications [20] through a reachability analysis on disassembled bytecode. Several analyses have statically found permission leaks [17, 19, 25], which happen when a privileged application leaks its capabilities to unprivileged ones. These analyses focus on finding paths between exposed entry points and sensitive API calls, whereas we focus on connecting exit points to entry points. Thus, these analyses could benefit from our ICC analysis.

ScanDal [23] attempts to soundly analyze information flow. It convert Dalvik bytecode to a formally defined intermediate language. Dangerous flows are detected using abstract interpretation. Its analysis is path-insensitive and has limited context-sensitivity. It finds some actual privacy leaks, but is limited by a high number of false positives and flows that are impossible to confirm.

Saint [30] modifies the Android framework to control application interaction. Every application comes with a policy describing how it uses permissions it declares. Policy compliance verification is a possible application of our tool but is out of the scope of this paper.

8 Conclusion

In this paper we have introduced an efficient and sound technique for inferring ICC specifications, and demonstrated its feasibility on a large collection of market applications. Future work will study a range of applications and analyses that exploit the database of ICC specifications. We will also explore a range of extensions that can use this information at runtime to identify potentially malicious communication between applications. Through these activities, we aim to aid the community's efforts to gauge the security of market applications.

Acknowledgements

We thank Matthew Dering for providing our application samples. We also thank Atul Prakash, Patrick Traynor and our shepherd Ben Livshits for editorial comments during the writing of this paper. This material is based upon work supported by the National Science Foundation Grants No. CNS-1228700, CNS-0905447, CNS-1064944 and CNS-0643907. 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 is also supported by a Google Faculty Award.

References

[1] ARTHUR, C. Feature phones dwindle as android powers ahead in third quarter. *The Guardian*, Nov. 2012. Available at `http://www.guardian.co.uk/technology/2012/nov/15/smartphones-market-android-feature-phones`.

[2] BARRERA, D., KAYACIK, H. G., VAN OORSHOT, P. C., AND SOMAYAJI, A. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *Proceedings of the ACM Conference on Computer and Communications Security* (Oct. 2010).

[3] BODDEN, E. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis* (2012). Available from `http://sable.github.com/heros/`.

[4] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., AND SADEGHI, A.-R. XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Tech. Rep. TR-2011-04, Technische Universitat Darmstadt, Germany, Apr. 2011.

[5] BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., SADEGHI, A.-R., AND SHASTRY, B. Towards taming privilege-escalation attacks on Android. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium* (Feb. 2012).

[6] CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2011).

[7] CHRISTENSEN, A. S., MØLLER, A., AND SCHWARTZBACH, M. I. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium (SAS)* (June 2003), vol. 2694 of *LNCS*, Springer-Verlag, pp. 1–18. Available from `http://www.brics.dk/JSA/`.

[8] DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., AND WINANDY, M. Privilege Escalation Attacks on Android. In *Proc. of the 13th Information Security Conference (ISC)* (Oct. 2010).

[9] DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *20th USENIX Security Symposium* (2011).

[10] ENCK, W. Defending users against smartphone apps: Techniques and future directions. In *ICISS* (2011), pp. 49–70.

[11] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of the 9th USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (2010).

[12] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium* (August 2011).

[13] ENCK, W., ONGTANG, M., AND MCDANIEL, P. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)* (Nov. 2009).

[14] ENCK, W., ONGTANG, M., AND MCDANIEL, P. Understanding Android Security. *IEEE Security & Privacy Magazine 7*, 1 (January/February 2009), 50–57.

[15] FELT, A. P., CHIN, E., HANNA, S., SONG, D., AND WAGNER, D. Android Permissions Demystified. In *Proc. of the ACM Conf. on Computer and Communications Security (CCS)* (2011).

[16] FELT, A. P., GREENWOOD, K., AND WAGNER, D. The Effectiveness of Application Permissions. In *Proc. of the USENIX Conference on Web Application Development (WebApps)* (2011).

[17] FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission Re-Delegation: Attacks and Defenses. In *Proc. of the 20th USENIX Security Symp.* (August 2011).

[18] GILBERT, P., CHUN, B.-G., COX, L. P., AND JUNG, J. Vision: Automated Security Validation of Mobile Apps at App Markets. In *Proceedings of the International Workshop on Mobile Cloud Computing and Services (MCS)* (2011).

[19] GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS '12* (2012).

[20] GRACE, M. C., ZHOU, W., JIANG, X., AND SADEGHI, A.-R. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks* (2012), WISEC '12, ACM.

[21] HARDY, N. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev. 22*, 4 (Oct. 1988).

[22] HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2011).

[23] KIM, J., YOON, Y., AND YI, K. Scandal: Static analyzer for detecting privacy leaks in android applications. In *MoST 2012: Workshop on Mobile Security Technologies 2012* (2012).

[24] LHOTÁK, O., AND HENDREN, L. Scaling java points-to analysis using spark. In *Proceedings of the 12th international conference on Compiler construction* (2003), CC'03, Springer-Verlag.

[25] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proc. of the 2012 ACM conference on Computer and communications security* (2012), CCS '12, ACM, pp. 229–240.

[26] MCDANIEL, P., AND ENCK, W. Not So Great Expectations: Why Application Markets Haven't Failed Security. *IEEE Security & Privacy Magazine 8*, 5 (September/October 2010), 76–78.

[27] MLOT, S. Google's bouncer malware tool hacked. *PC Magazine*, June 2012. Available from `http://www.pcmag.com/article2/0,2817,2405358,00.asp`.

[28] OCTEAU, D., ENCK, W., AND MCDANIEL, P. The ded Decompiler. Tech. Rep. NAS-TR-0140-2010, Network and Security Research Center, Pennsylvania State University, USA, Sept. 2010. Available from `http://siis.cse.psu.edu/ded/`.

[29] OCTEAU, D., JHA, S., AND MCDANIEL, P. 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/`.

[30] ONGTANG, M., MCLAUGHLIN, S., ENCK, W., AND MCDANIEL, P. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)* (Dec. 2009), pp. 340–349.

[31] ROSENBERG, J. Google play hits 25 billion downloads. *Android - Official blog*, Sept. 2012. Available at `http://officialandroid.blogspot.com/2012/09/google-play-hits-25-billion-downloads.html`.

[32] SAGIV, M., REPS, T., AND HORWITZ, S. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci. 167*, 1-2 (Oct. 1996), 131–170.

[33] SECURITY, N. Malware controls 620,000 phones, sends costly messages. *Help Net Security*, January 2013. Available from `http://www.net-security.org/malware_news.php?id=2391`.

[34] VALLÉE-RAI, R., GAGNON, E., HENDREN, L. J., LAM, P., POMINVILLE, P., AND SUNDARESAN, V. Optimizing java bytecode using the soot framework: Is it feasible? In *Proc. of the 9th International Conf. on Compiler Construction* (2000), CC '00.

[35] ZHENG, C., ZHU, S., DAI, S., GU, G., GONG, X., HAN, X., AND ZOU, W. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices* (2012), ACM, pp. 93–104.

[36] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the Network and Distributed System Security Symposium* (Feb. 2012).