Flow Permissions for Android

Shashank Holavanalli, Don Manuel, Vishwas Nanjundaswamy, Brian Rosenberg, Feng Shen,
Steven Y. Ko, Lukasz Ziarek
SUNY Buffalo
{sholavan, donmanue, vishwasg, bjr24, fengshen, stevko, lziarek}@buffalo.edu

***Abstract*—This paper proposes Flow Permissions, an extension to the Android permission mechanism. Unlike the existing permission mechanism our permission mechanism contains semantic information based on information flows. Flow Permissions allow users to examine and grant explicit information flows within an application (e.g., a permission for reading the phone number and sending it over the network) as well as implicit information flows across multiple applications (e.g., a permission for reading the phone number and sending it to another application already installed on the user's phone). Our goal with Flow Permissions is to provide visibility into the holistic behavior of the applications installed on a user's phone. Our evaluation compares our approach to dynamic flow tracking techniques; our results with 600 popular applications and 1,200 malicious applications show that our approach is practical and effective in deriving Flow Permissions statically.**

I. Introduction

Android is a popular platform for mobile devices. Applications for Android are written mainly in Java and referred to as 'apps.' Unlike other mobile OSes, Android has a unique permission mechanism. At development time, an app writer needs to explicitly request permissions by including them in an app configuration file (`AndroidManifest.xml`). We refer to this configuration file simply as the 'manifest' in the remainder of the paper. During installation, each user needs to review the permissions that the app requests and explicitly grant them for the duration the app is installed.

Currently, there are over 130 permissions which Android apps can request in API level 17. Generally, an application can ask for permissions to use protected APIs for phone resources (e.g, storage, NFC, WiFi, etc.) or information available on the phone (e.g., contacts, location, call logs, etc.). For example, if an application wants to use APIs that control the camera, it needs to request the `android.permission.CAMERA` permission. For perspecuity we will use the shorthand `PERM` when referring to a specific permission of the form `android.permission.PERM`.

Although considered to be robust, the current Android permission mechanism has a number of deficiencies. There is no validation that the permissions requested by an app are actually needed by the app at runtime (*i.e.* the app is over privileged with respect to APIs it uses). The burden of deciding to grant permissions is placed on the user, but the permissions themselves provide little contextual information on how sensitive APIs are leveraged by the app. For example, it is unclear if an app with the permission to access the internet, as well as the phone's SIM card, exposes the private telephony data stored on the SIM card to the outside world. Apps can collude with one another to effectively gain permission they were not explicitly given (a danger that is compounded if apps are over privileged), through the many inter-process communication mechanisms Android provides.

To address these issues, we propose a new permission mechanism, called *Flow Permissions*, that extends the existing Android permission mechanism with information on information *flows* between permission domains (*e.g.* reading from the SIM card and sending over the network). We also introduce cross-app Flow Permissions that identify how apps can interact explicitly through IPC mechanisms, and *deployment permissions* – implicit Flow Permissions granted when installing an application based on indirect interactions possible between apps installed on a phone (*i.e.* a deployment) and a newly installed app. To help developers as well as users, we provide an automated tool, called Blue Seal, for synthesizing Flow Permissions. Blue Seal integrates techniques based on automated checkers [1] to remove unnecessary permissions from over privileged apps and then synthesizes Flow Permissions for the app. Blue Seal includes a lightweight cross app analysis that can analyze multiple apps to discover cross app flows or can be leveraged at installation time to detect implicit deployment permissions.

In this paper, we make the following contributions:

- **Flow Permissions:** A new permission mechanism based on information flows between permission domains within an app, as well as across multiple apps. Cross app flow detection can be leveraged at installation time to alert the user to implicit deployment permissions.
- **Blue Seal:** A tool, called Blue Seal, for automatically generating Flow Permissions, as well as a primer on how to modify classic program analyses to analyze Android specific constructs statically. Blue Seal generates Flow Permissions for an app statically in order to display the Flow Permissions before a user installs the app.
- **Case studies:** Detailed performance analysis including a comparison study with state-of-the-art tools as well as a large validation across 600 popular and 1,200 malicious apps. We present results from a user survey as a preliminary assessment of the utility of Flow Permissions.

The remainder of the paper is organized as follows: we first present a series of motivating examples showing the current problems with the Android permission mechanism in Section II. Our Flow Permission extension to the Android

TABLE I
TABLE LISTING ANDROID APPS AND THEIR REQUESTED PERMISSION EXAMPLES.

Android App	Category	Permissions Requested
MyCalendar	Productivity	STORAGE LOCATION NETWORK PHONE CALLS
MySpace	Social	STORAGE NETWORK PHONE CALLS
Blackmoon File Browser	Productivity	STORAGE SYSTEM TOOLS
Gmail	Communication	NETWORK STORAGE

permission mechanism is detailed in Section III along with additional details on the Android platform that make inferring Flow Permissions difficult. Blue Seal is presented in Section IV. Case studies and comparisons to existing tools are presented in Section V. Related work and conclusions are given in Section VI and Section VII respectively.

II. MOTIVATION

To motivate the necessity of extending the current Android permission mechanism, we examine four apps in detail: MyCalendar, MySpace, Blackmoon File Browser, and Gmail. MyCalendar (com.kfactormedia.mycalendarmobile) is a third-party calendar app, MySpace (com.myspace.android) is a social networking app with multimedia support, Blackmoon File Browser (com.blackmoonit.android.FileBrowser) is a popular file manager, and Gmail (com.google.android.gm) is a well-known email app from Google. Although these apps have widely varying functionality, MyCalendar and MySpace request similar permissions.

A partial and stylized set of permissions each app requests is given in Table I. Notice that MyCalendar and MySpace both request `PHONE CALLS` and `NETWORK`. The `PHONE CALLS` permission grants the app a set of more fine grained permissions, which we omit for brevity, including permission to read the phone number, device ID, and the phone state. Similarly, the permission `NETWORK` allows the app to access the internet, either through wifi or cellular networking.

Savvy users may notice that by granting permissions to read from the phone's log and phone state as well as access to the internet, they are also implicitly granting permission to transmit data stored within the call log and phone state over the internet to an external source. Once the app has permission to read from a given piece of data stored on the phone (*i.e.* a data *source*) as well as permission to send data outside of the app (*i.e.* a data *sink*), the app also *implicitly* has permission to export the source data via the sink. Importantly, the permissions offer no insight if the apps leverage the APIs to ex-filtrate data.

A. Flows as Permissions

The goal of the Flow Permission mechanism is to show whether or not an app contains a *flow* between a source and a sink. The general structure of a Flow Permission is of: *source* $\rightarrow$ *sink*. From Table II, we can see that, even though MyCalendar and MySpace are granted the same permissions (`PHONE CALLS` and `NETWORK`), MyCalendar is augmented by our tool to contain the Flow Permission: `PHONE NUMBER` $\rightarrow$ `NETWORK`. This Flow Permission indicates that data read from the stored phone number was subsequently exported through the use of the network. Additionally, we can deduce that MySpace does not contain such a flow as it does not report such a permission. The MySpace app does, however, transmit the International Mobile Equipment Identity (IMEI) number of the device, which is indicated by the `IMEI NUMBER` $\rightarrow$ `NETWORK` Flow Permission.

TABLE II
TABLE LISTING ANDROID APPS AND THEIR REQUESTED PERMISSIONS ALONG WITH OUR PROPOSED FLOW PERMISSION EXTENSIONS.

Android App	Flow Permissions
MyCalendar	PHONE NUMBER $\rightarrow$ NETWORK
MySpace	IMEI NUMBER $\rightarrow$ NETWORK
Blackmoon File Browser	STORAGE $\rightarrow$ NETWORK

In this manner, Flow Permissions provide the user additional context on how the standard Android permissions and the resources / data they protect are leveraged by the apps. Nevertheless, it is up to the user to decide if these behaviors should be allowed or not. The existence of a flow does *not* indicate that the app is necessarily malicious. For example, a social networking app might be expected to contain a flow from the IMEI number to the network as this provides the app a mechanism to uniquely identify the device for analytics. However, some users may not be comfortable providing such information to the app developer, as other mechanisms (*e.g.* manual login screens) can be used without exposing such data. In contrast, a calendar app should not have such a flow. We do note, that certain Flow Permissions should never be granted, namely exposure of the user's International Mobile Subscriber Identity [1] (IMSI) number from the SIM card.

B. Interaction Between Apps

Consider a more complicated case that highlights how multiple apps can expose data sources and sinks to one another, thereby acquiring additional implicit permissions [2]. The Blackmoon File Browser app includes functionality to send a file as an email attachment. However, the app cannot access the network to send an email as it does not have the `NETWORK` permission. Instead, the Blackmoon File Browser leverages Gmail's public interface to send files over the network. In other words, the Blackmoon File Browser is implicitly granted permission, if Gmail is also installed, to use the network without overtly requesting such a permission. Flow Permissions, on the other hand, highlight the flow between the Blackmoon

[1]This number is used to uniquely identify the user, phone, and subscription plan. Networks use this to establish roaming policies and charges associated with non local network usage.

File Browser and the network, accomplished through the RPC mechanism leveraged to transmit the file, as shown in Table II.

III. Flow Permissions

Flow Permissions are an extension to the Android permission mechanism that characterizes the *implicit* interactions between data and APIs protected by standard permissions. This interaction is determined by the existence of an information *flow* between the permission *domains*. Although there maybe multiple Android permissions dealing with a domain (*i.e.* `READ_SMS`, `WRITE_SMS`, `RECEIVE_SMS`, and `SEND_SMS`, *etc.*), we only consider the domains themselves (*e.g.* SMS). Domains are split into three categories: source domains, which can be viewed as sources of data; sink domains, which can be viewed as data export mechanisms; and cross-app domains, domains which act as inputs or outputs between apps.

A. Permission Domain Types

Out of over 130 Android permissions, we have identified thirteen canonical source domains: `NETWORK`, `EMAIL`, `IME`, `SMS`, `MIC`, `CALENDAR`, `ACCOUNTS`, `SDCARD`, `CONTACTS`, `CAMERA`, `CALL LOG`, `SIMCARD` [2], and `LOCATION`. Similarly, we have identified five canonical sink domains: `NETWORK`, `EMAIL`, `SMS`, `SDCARD`, and `LOG`. The third type of domain (cross-app) consists of Android's IPC mechanisms that allow apps to share data or provide services to one another. For example, Gmail exposes its email service to other apps via an IPC mechanism as mentioned in Section II-B. These IPC mechanisms can bridge a source domain in one app to a sink domain in another app.

B. Permission Mechanism

Flow Permissions can be viewed as relations between the three types of permission domains. There are four potential types of Flow Permissions:

- `Source` → `Sink`: A flow from a source domain to a sink domain.
- `IPC` → `Sink`: A flow from an IPC source domain to a sink domain.
- `Source` → `IPC`: A flow from a source domain an IPC sink domain.
- `IPC` → `IPC`: A flow from an IPC source domain to an IPC sink domain.

Of the four types of Flow Permissions, those which deal with flows to and from `IPC`, are not reported directly to the user by default [3]. Instead, these Flow Permissions, along with meta-data to disambiguate the IPC, are leveraged at installation time or during cross-app analysis to synthesize cross-app and deployment flows. Abstractly, cross-app and deployment flows are characterized by one app having a Flow Permission of the form: `Source` → `IPC`, another app having a Flow Permission of the form: `IPC` → `Sink`, and any number of apps having Flow Permissions of the form: `IPC` → `IPC`.

[2]We observe that the SIM Card stores two important numbers: IMSI and ICCID and our Flow Permissions distinguish these two cases. Examples and explanation are given in Sec. V-D1.

[3]Our tool can be configured to emit these as well.

C. Statically Deriving Flow Permissions

In order to present Flow Permissions at installation time, we statically analyze Android apps to derive them. In doing so, we overcome a set of challenges unique to Android.

1) Android Programming: As with other GUI frameworks, Android's programming model is highly event-driven. Many of the Android APIs are essentially event handler interfaces that an app needs to implement to handle various events via callbacks. The Android framework calls these event handlers not only for user-generated events such as a button click, but also for framework events such as app start, stop, and pause. At the minimum, an app is required to extend one *framework component* [4] class that defines handlers for framework events. In addition, Android has introduced many new constructs, including new thread types (e.g., `AsyncTask`), messages and message handlers (e.g., `Intent` and `Handler`), and IPC mechanisms (e.g., `Binder`). We detail the usage scenarios of these constructs further in Section IV.

2) Challenges for Static Analysis: Android's unique programming model and constructs present the following set of challenges for static analysis.

- All entry points for an app must be identified to leverage standard analyses like deadcode elimination. The Android standard library has over 1,700 possible entry points.
- Methods registered as callbacks and listeners for various external and internal events must be identified, and their invocation points tracked. Android allows callbacks and listeners to be registered not only in the app code, but also in configuration files. Thus, configuration files must also be analyzed.
- Android provides new classes and methods for inter-thread communication in a message passing style, necessitating the pairing of possible send and receive points of messages.
- The Android IPC mechanisms require disambiguation to distinguish which apps communicate to other apps and through which mechanisms.
- Android manages the execution of asynchronous tasks, implicitly invoking methods during specific points its lifetime. These methods must be handled explicitly and their implicit invocation sites discovered.

IV. System Design

Our system design is built on top of the Soot Java Optimization Framework [3], [4]. Since Soot is originally developed for analyzing Java bytecode, Soot integrated the Dexpler Dex to Java bytecode translator [5] to transform Dex bytecode into Soot's own intermediate represetation (Jimple). In addition, we leverage the PScout Permission Map [6]; abstractly, a permission map is a mapping between Android API calls

[4]Android defines four framework components—activities, services, broadcast receivers, and content providers. An app extends `Activity` to handle UI events; `Service` to perform background tasks; `BroadcastReceiver` to handle broadcast events (e.g., a battery low event) from either the Android framework or apps; and `ContentProvider` to provide a custom storage with a database-like interface.

and the permissions required to enact those calls. The PScout Permission Map was generated by statically analyzing the entire Android source code and to our knowledge is the most complete among known permission maps. Our compiler leverages this precomputed mapping internally within the analyses to associate specific permission to API calls.

At its core, our Blue Seal leverages classic forward and backward intraprocedural dataflow analysis as well as interprocedural dataflow analysis based on graph reachability. As outlined in Fig. 1, Blue Seal leverages six main analysis passes to generate Flow Permissions: 1) entry point discovery, 2) call graph restructuring, 3) unused permission analysis, 4) resolution of intents, content providers, as well as uses of the binder, 5) interprocedural permission flow analysis, and 6) cross-app permission flow analysis. Abstractly, Blue Seal uses analyses 2, 3, and 4 to disambiguate Android specific constructs and identify source and sink points, prior to tracking flows between sources and sinks in analysis 5. Since Blue Seal is built from classic analysis techniques, we tailor our discussion on Android specific linguistic constructs, libraries, and IPC mechanisms and how to modify standard analyses to support them. Currently, Blue Seal is not path or context sensitive. Blue Seal implements Stowaway's unused permission analysis [1] to remove unnecessary permissions, the details are omitted for brevity.

A. Entry Point Discovery

The Android platform is event driven and almost all apps have multiple entry points. Prior to static analysis, precise entry point detection should be performed to improve precision. As we describe below, there are three ways that an app can register entry points and our entry point discovery covers all three cases. Our approach complements the entry point analysis given in CHEX [7] with support for entry points specified in layout configuration files as well as discovering potential entry points from the Android API documentation.

1) Framework Components: Any Android app is required to implement one of the four main framework components (`Activity`, `Service`, `BroadcastReceiver`, and `ContentProvider`). These components have standard entry points and are declared in the manifest. Blue Seal discovers framework components' entry points by analyzing this file.

2) UI Layout: A developer can also declare entry points handling UI events such as button clicks in layout configuration files. Discovering these entry points requires extra analysis on the layout configuration files; this is due to the fact that an app can contain multiple layout configuration files, one for each layout it uses. Android internally maintains the mappings between layouts and their configuration files by generating another configuration file at compile time. Blue Seal analyzes this internal configuration file to match classes corresponding to the layouts the app uses, each of which is identified by unique int, to handlers defined in that layout's configuration file.

3) API Callbacks: The third entry point option is implementing callbacks pre-defined in the Android APIs. In order to discover these pre-defined entry points, we have implemented a crawler that parsed the API documentation and discovered that the current Android API documentation (API 17) has 1,738 callback methods that can serve as potential entry points.

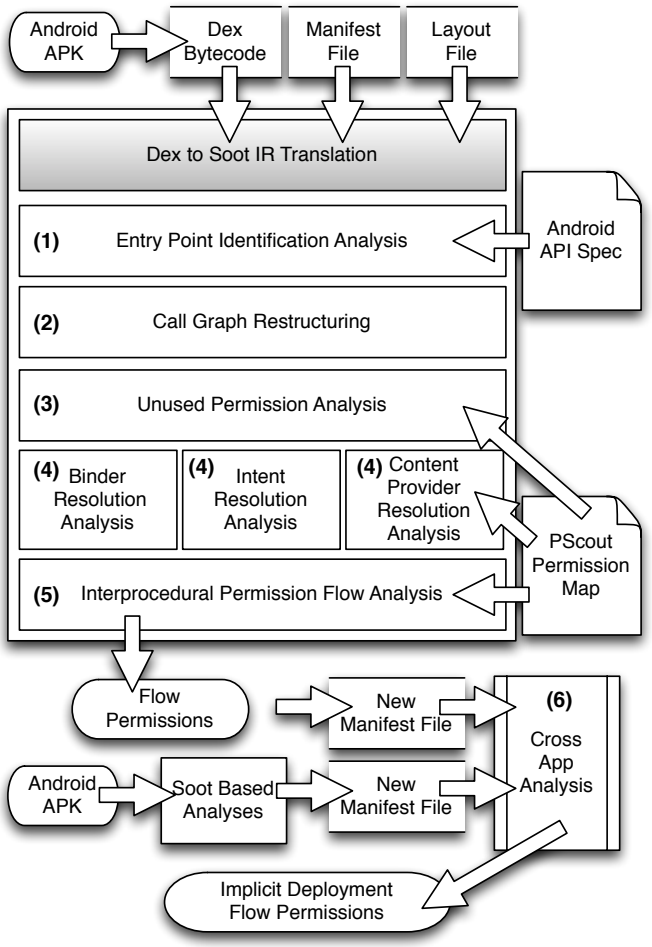

Fig. 1. The Blue Seal Android app analysis framework architecture. Shaded boxes represent components already present in Soot.

B. Call Graph Restructuring

The Android framework is responsible for implicitly invoking methods associated with many of the constructs it provides. To correctly analyze an app, we must infer the association of user-called methods to their corresponding framework-invoked methods. We discuss the two most common cases below.

1) Async Tasks: `AsyncTask` is a new threading class introduced in Android. It provides a simple way to write a short lived thread that communicates with the UI thread in an asynchronous fashion. An `AsyncTask` can implement five methods—`onPreExecute`, `doInBackground`, `onProgressUpdate`, `onPostExecute`, and `onCancelled`, which dictate the control flow of the asynchronous task. As an example consider the code snippet in Fig. 2 and the corresponding control flow given in Fig. 3.

The `doInBackground` method performs the actual computation for the async task. The methods

```
public class MainActivity extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        ...
        new Task().execute("http://www...");
        ...
    }
    ...
    private class Task extends AsyncTask<String, String, Integer> {
        ...
        protected void onPreExecute() {
            ...
        }
        protected Integer doInBackground(String... strs) {
            ...
            publishProgress("intermediate result");
            ...
            return intObj;
        }
        protected void onProgressUpdate(String...strings) {
            ...
        }

        protected void onPostExecute(Integer intObj) {
            ...
        }
    }
}
```

Fig. 2. A code snippet illustrating the methods that comprise the control flow of an async tasks in Android and the implicit flow of arguments provided by the Android framework.

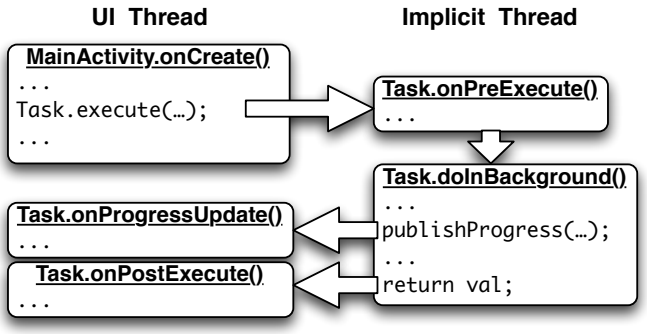

Fig. 3. The execution flow of async task methods in their respective threads at runtime.

```
public class HandlerActivity extends Activity {
    ...
    private Handler mHandler = new Handler() {
        public void handleMessage(Message msg) {
            ...                                    (4)
            Bundle data = msg.getData();
            ...                                    (5)
        }
    };                                                    (3)
    public void onClick(View v) {
        new Thread(new Runnable() {
            public void run() {
                try {                              (1)
                    ...
                    msg.setData(data);
                    mHandler.sendMessage(msg);
                    ...                            (2)
                } catch (InterruptedException e) {
                    ...
                }
            }
        }).start();
    }
}
```

Fig. 4. Flows based on pairing message sends to the appropriate message handlers.

`onPreExecute` and `onPostExecute` run before and after `doInBackground` and typically include pre- and post-processing. The `onCancelled` method is called when the async task is cancelled by another thread. Notice that `onPreExecute` will execute in the implicitly created thread backing the asynchronous task, but `onPostExecute` callback will be executed by the UI thread. Similarly, `onProgressUpdate` gets executed as a callback in the UI thread after there is a call to `publishProgress` within `doInBackground`. An app writer can call `AsyncTask`'s `execute` and `executeOnExecutor` to start an `AsyncTask`. Obviously, a typical call graph generation process does not understand this execution flow; hence, we identify all `AsyncTask` instances and augment the call graph to include edges corresponding to the async task control flow. We do this by effectively replacing the invoke of `execute` with invoke calls to `onPreExecute`, `doInBackground`, and `onPostExecute`. Similarly, a call to `publishProgress` is replaced with a `onProgressUpdate` call. Notice that `doInBackground` implicitly passes its return value as an argument to `onPostExecute`. `publishProgress` also passes its arguments as arguments to `onProgressUpdate`. The call graph and method bodies are updated accordingly.

2) Handler: Android also provides a message mechanism for communicating between threads within an app, called `Handler` (depicted in Fig. 4). Threads can communicate through a shared Handler object. Receiving threads implement the `handleMessage` method to process received messages and sending threads communicate through the `sendMessage*` family of methods [5]. Similar to async tasks, Blue Seal effectively replaces a call to `sendMessage*` with a call to `handleMessage` to restructure the call graph.

C. Content Provider Resolution Analysis

After restructuring the call graph, Blue Seal performs additional analyses to identify permission domains discussed in Section III. One mechanism for interaction between apps is the Content Provider (CP). An app can provide content to itself or other apps, can consume content hosted by a CP, or both. CPs are uniquely identified by an URI object (`android.net.Uri`) and to correctly pair uses of CPs these objects must be tracked and disambiguated to the extent possible by static analysis. To identify uses of CPs we track the Content Provider API calls as well as the URI Objects (as shown in Fig. 5). Our CP Resolution Analysis (CPRA) is based on an interprocedural dataflow analysis that leverages a backward intraprocedural data flow analysis. Abstractly, we

[5]By method family we mean any methods of similar form defined by the same class (*e.g.* `setData` and `setDataAndType` belong to the method family `setData*`).

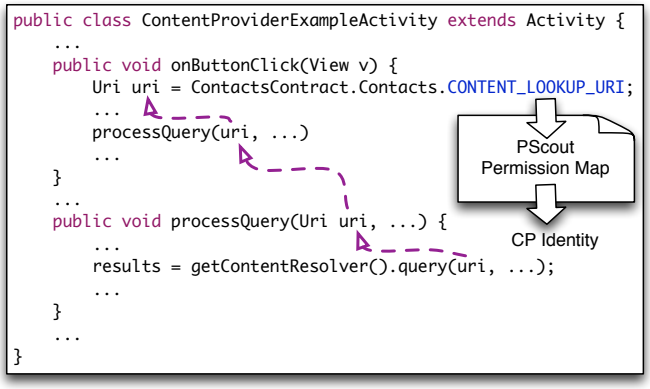

Fig. 5. The data flow of an URI object that identifies which CP is being utilized. Dashed arrows indicate information derived from dataflow analyses and block arrows how that information is used to disambiguate the CP.

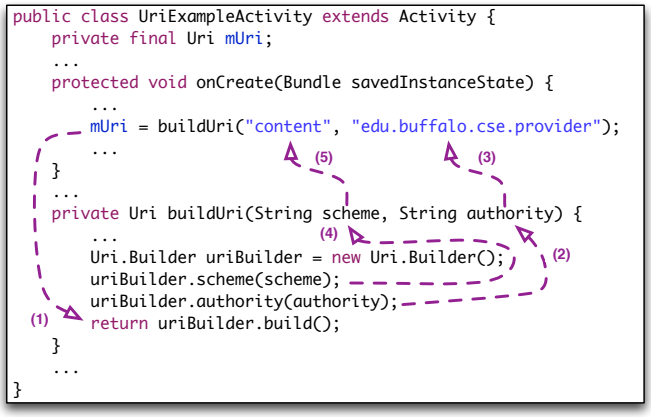

Fig. 6. The data flow of an URI object initialization that is resolvable statically.

track backward flows from uses of the CP mechanism to the definitions of URI objects and from the definitions of URI objects to the strings that uniquely identify them.

CP are accessed through two separate classes in Android: `ContentResolver` and `ContentProviderClient`. Within these classes the methods from which we begin tracking flows are: `insert`, `query`, and `update`. Each of these methods takes an URI object as an argument. Our analysis identifies the creation points of the URI objects passed into these methods. URI objects can be created in one of two ways: they can be provided by the Android libraries or they can be constructed by the app itself. In the former, the identifying URI string is hidden. For precision, our analyses leverages PScout, which also provides a mapping between framework provided URI objects and their URI strings as shown in Fig. 5. For app created URI objects we attempt to discover this information in the compiler. Once the app created URI object is identified, the analysis tracks the construction of this object. There are two ways to construct an URI object. One is to use `Uri.parse` and the other is to use `Uri.Builder`. The first case is simple as the argument to the `parse` method is the URI string. If `Uri.Builder` is used, then `Uri.Builder.scheme` is used to set the scheme and `Uri.Builder.authority` is used to set the authority. For example, 'content://edu.buffalo.cse.provider', is a valid CP identifier where the authority is 'edu.buffalo.cse.provider' and 'content' is the scheme. After the scheme and the authority are set, `Uri.Builder.build` returns the actual URI object. Thus, our analysis tracks calls to `scheme` and `authority` and the arguments passed to them as shown in Fig. 6.

D. Intent Resolution Analysis

Intents are message objects that can be used to send data between components within a single app as well as across different apps. An app can receive intents in two ways, either statically or dynamically. Static intents are declared in the app's manifest file on a per component basis. An app can also register itself to receive intents dynamically at run time

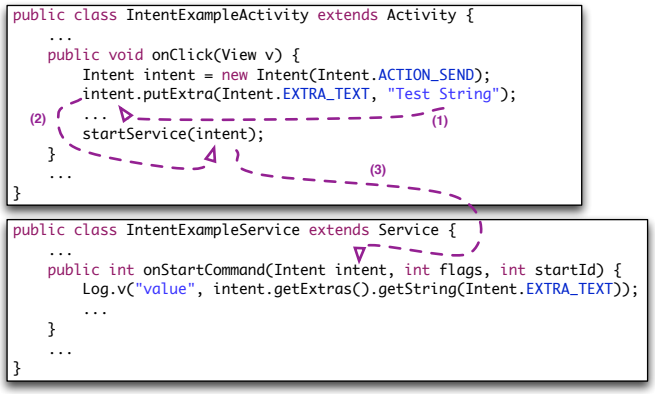

Fig. 7. The data flow of an Intent passed from one app to another.

without declaring it in its manifest file.

To resolve dynamic intents on the receiver side, our Intent Resolution Analysis (IRA) must first discover all classes that are registered to receive the intent via `Context.registerReceiver`. The call to `registerReceiver` requires an intent filter that identifies which intents the class is able to receive. Intents that are to be received by the filter can be specified at initialization time via the intent filter constructor or dynamically via the `add` method. In much the same was as CPRA, IRA also performs an inter-procedural backwards flow analysis to disambiguate between intents by tracking strings. An example is shown in Fig. 7.

Once the intents are disambiguated, the analysis must identify possible sources and sinks related to the intents. Intent sinks are identified by any API call that inserts data into the intent. In Android, this can be done through the intent constructors `Intent(...)` as well as the method call families `put*Extra` and `setData*`. Intent sources are identified by any API call that retrieves data from the intent, namely the method family `get*`. Intents themselves can be sent between apps. Sinks related to the intents

used in this manner are the methods that can send out intents to other apps and include the following method families: `send*Broadcast*`, `startActivity*`, `startIntentSender`, `startService`, and `stopService`. All the methods listed above require an Intent object. Similar to intent filter, there are two main ways to specify an action string that uniquely identifies the intent. First is at the initialization time via the constructor. The second is by using `setAction`. IRA does not track what data flows through or from intents, it only serves to identify and disambiguate how the app leverages intents. Effectively, IRA computes points in the intermediate representation that act as sources and sinks for the subsequent compiler passes.

E. Binder Resolution Analysis

`Binder`/`IBinder`, commonly referred to as just *Binder*, is the default IPC mechanism on Android. It can be used for inter-component communication within the same app (e.g., activity-to-service communication) as well as inter-process communication between different apps. Android provides multiple ways to use the Binder mechanism, such as simply extending the base `Binder` class or using AIDL (Android Interface Definition Language) to define a customized interface. Regardless of which method is used, a Binder server (i.e., an IPC callee) implements all the IPC methods in the `Binder` class. A Binder client (i.e., an IPC caller) uses an `IBinder` object which is the proxy for the server-side `Binder`. Fig. 8 shows an example.

Although Binder calls are mostly identical to local calls, there are two cases to handle for correctness of our analysis. First, for inter-component communication, we need to match each call with an `IBinder` object to the corresponding `Binder` implementation. Second, for inter-process communication, each client-side `IBinder` call is a potential sink, which might result in a server-side `Binder` call which then becomes a potential source.

A variation of `Binder` is `Messenger`, which allows a process to send a message to another process. It relies on `Binder`/`IBinder` to implement its functionalities underneath, but is simpler to use from the programmer's point of view. In order to receive a message, a server needs to create a `Messenger` object; it also needs to implement a `Handler` as described in Section IV-B2 and pass it to the `Messenger` object. In order to send a message, a client can use `Messenger`'s `send` method. We handle these implicit calls by matching calls to `send` with `Handler`'s `handleMessage`. If matches cannot be enumerated we treat them as a potential sink (for `send`) or a source (for `handleMessage`).

F. Interprocedural Permission Flow Analysis

To synthesize Flow Permissions we leverage an interprocedural forward flow analysis to track flows between sources and sinks. Our analysis is fixed point based, leveraging the standard work list model and method summaries. The flow analysis is parameterized by a listing of sources and sinks. Sources and sinks are specified directly from API calls via the PScout Permission Map or synthesized by CPRA, IRA, and BRA. Sources and sinks synthesized by CPRA, IRA, and BRA correspond to uses of IPC. The goal of this analysis is to track data flows originating at sources and terminating at sinks.

1) Computing and Applying Method Summaries: The intraprocedural forward flow analysis, leveraged by our interprocedural analysis, builds a method summary for each reachable method. The intraprocedural analysis is standard and builds in-flow and out-flow sets for each statement in the method body. The method summary constructed during this analysis is a flow graph representing the flows between sources and sinks within the method itself as well as arguments, returns, and class variables the method reads or writes. We add nodes to the graph for every argument, return statement, statement containing a class variable read/write, and statements identified as sources or sinks. Edges between nodes are added when a flow is determined by the intraprocedural forward flow analysis. Argument nodes and source nodes can have only outgoing edges. Sinks and return nodes can have only incoming edges. Nodes which represent class variable reads/writes can have both incoming and outgoing edges. Thus, there are four types of possible flows contained within the flow graph comprising the method summary: 1) generative flows: flows from a source to a return or class variable, 2) terminating flows: flows from an argument or class variables to a sink, 3) local flows: flows from a sources to a sink, and 4) transitive flows: flows from arguments or class variables to other class variables or returns. Orphan nodes, nodes with no incoming or outgoing edges, are pruned.

At a method call site the analysis applies the summary for that method. If the method summary contains transitive flows, we add the arguments supplied at the call site to the out-flow set for the call. For both generative and terminating flows, we add a place holder node into the method summary. This place holder node represents potentially multiple sources and/or sinks, one for each generative and terminating flow. The place holder nodes will be used to synthesize a global flow graph once all method summaries have been computed and the interprocedural analysis reaches a fixed point. Edges between nodes in the flow graph and the place holder node are added as if the place holder node was a source and/or sink node.

2) Synthesizing a Global Flow Graph: Once the interprocedural analysis reaches a fix point, we synthesize a global flow graph from the per method summaries. Place holder nodes that were inserted when method summaries were applied and class variable nodes serve as merge points for combining method summaries. Once all method summaries are merged, paths that do not originate from a source and terminate in a sink are pruned. Flow Permissions can be generated from the graph by enumerating all paths and removing duplicates (*e.g.* an app may send contact data over the network in multiple code blocks). Lastly, we remove any Flow Permissions that correspond to permissions the app does not request. This

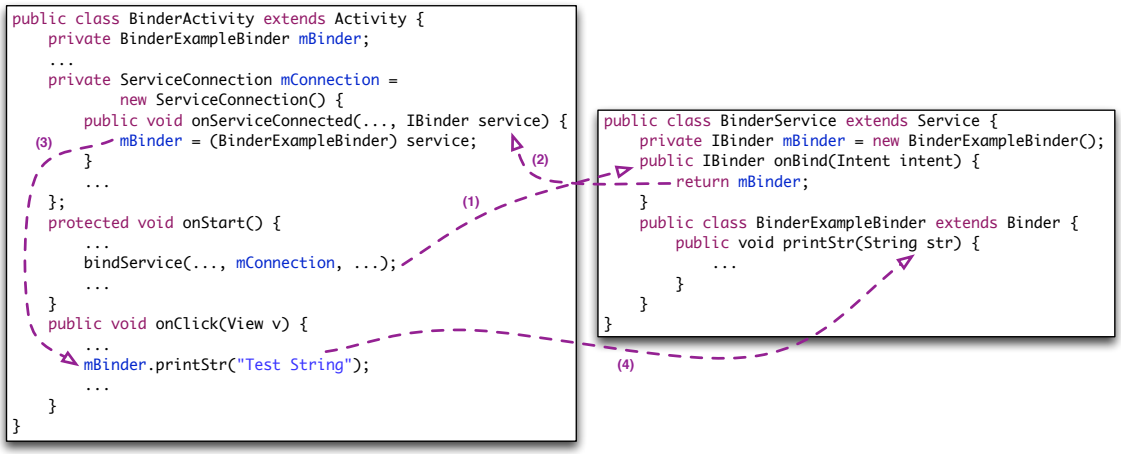

Fig. 8. Data and control flow relations between a binder activity and service.

step is necessary because ad libraries [8] check to see which permissions an app has been granted and perform computation based on these permissions. Thus, an app may contain code that contains flows, but will never be executed at runtime. In general, any flow that requires a permission the app has not been granted cannot be executed at runtime. The Flow Permissions are then added to the manifest file for the app.

Special consideration must be given to apps that leverage Android's shared user ID mechanism. This mechanism allows for multiple apps to execute as a single process. This process is granted the union of the permissions requested by the apps. In this case, we do *not* remove any Flow Permissions, regardless if the app requests the necessary permissions or not.

G. Cross App Permission Flows Analysis

Cross-app permission flow analysis simply enumerates a permutations of pairs of Flow Permissions between apps such that one app has a source to IPC flow and another has a IPC flow to a sink, and the IPC mechanism is the same. In the case where an IPC is not disambiguated statically in either app, our analysis is conservative and assumes *any* IPC mechanism of the same type can be potentially utilized. For file reads/writes to external storage we track the file name(s) if they are deducible statically. Notice that when an app is installed on a phone, its Flow Permissions and associated meta-data (*e.g.* types and identifiers of IPCs) can be compared to the Flow Permissions of installed apps to synthesize implicit deployment Flow Permissions.

V. Results and Discussion

To test the validity of our approach, we tested Blue Seal on 600 of the top rated free apps available on the Google Play Store, as of January 2013, and on 1,200 know malicious apps identified by the MalGenomeProject [6] [9]. We ran Blue Seal on the Amazon EC2 [10] using their 8-core node instance with 7GB of ram.

[6]The full dataset is publicly available at http://www.malgenomeproject.org.

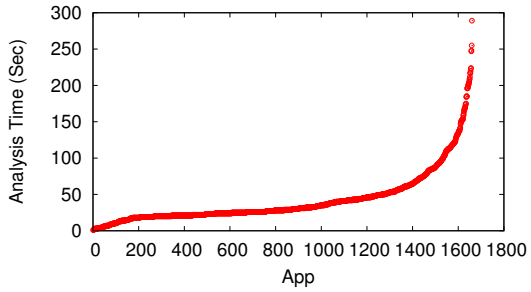

Fig. 9. Scatter plot showing the time taken to analyze all apps in seconds.

A. Performance

Blue Seal is able to analyze and synthesize Flow Permissions for all but the largest apps in under two minutes. Only 79 apps require an analysis time greater than two minutes. Full performance results are given in Fig. 9. However, Soot's front-end Dex bytecode parser, Dexpler, has limitations and generates incorrect intermediate representations for 199 of the apps. Of these apps, 169 are form the Google Play store and 30 are from the MalGenomeProject. We used an alternative tool, dex2jar [11], to translate these 199 apps into Java bytecode. Of the 199 apps, we were able to analyze 127 apps using the dex2jar translation in Blue Seal. We are currently investigating the causes of the mis-translation of the remaining 72 apps.

Cross-app analysis is executed over the Flow Permissions generated by Soot. Effectively, this analysis generates permutations of matches between Flow Permissions that share a common IPC. As the majority of apps have less than 10 Flow Permissions (8.34 Flow Permissions on average), we can calculate implicitly granted deployment permission on a phone that has 400 installed apps in less than 5 seconds. Although these experiments were executed in EC2, these preliminary numbers indicate that with optimizations this analysis is viable to be performed on an actual phone.

B. Flow Permission compared to Dynamic Taint Analysis

To test the correctness of our implementation we compared against TaintDroid [12]—a custom Android OS that performs a dynamic taint analysis for identifying malicious flows. We have manually compared Blue Seal's generated Flow Permissions to TaintDroid's dynamically discovered taints on over thirty apps. Each app was executed for 15 minutes and fed random key-presses. For every taint reported by TaintDroid while using the app, we checked that a corresponding Flow Permissions was synthesized by Blue Seal. Unsurprisingly, the most common taints reported by TaintDroid mirrored our own findings and that of prior work, namely: `IMEI NUMBER` $\rightarrow$ `NETWORK` and `LOCATION` $\rightarrow$ `NETWORK`. Thus far we have not discovered any taints reported by TaintDroid for which Blue Seal does not generate a corresponding Flow Permission.

C. Limitations

Blue Seal performs static analysis to generate Flow Permissions and thus suffers from the classic limitations of this approach. Although we analyze all of the Android specific constructs, our current implementation does not yet identify all classic Java sources and sinks. We have focused primarily on file, network, and output stream APIs. Wherever possible, Blue Seal leverages the fact that most RPCs, CPs, and Intents are known and enumerated statically by unique integers or unique strings. In most cases Blue Seal is able to disambiguate the components, but in cases where it cannot, Blue Seal necessarily needs to be conservative, leading to potentially many false positives. Our only comparison points were the comparison of synthesized Flow Permissions to TaintDroid in Sec. V-B and manual introspection of the apps. In the apps we tested, our Flow Permissions correctly identified the same flows as Taintdroid. This process is unfortunately not guaranteed to generate a flow if one exists. As such, we do not yet have a good metric to quantify false positives.

D. Case Studies: Flow Permissions in the real world

Based on our results gathered, we observe that the most common Flow Permissions generated for apps are: `IMEI NUMBER` $\rightarrow$ `NETWORK`, `IMEI NUMBER` $\rightarrow$ `LOG`, and `LOCATION` $\rightarrow$ `NETWORK`. In most cases these Flow Permissions correspond to the ad libraries leveraged by the apps. The IMEI number is typically used to uniquely identify the user being tracked. The location, established either by GPS or via networking data, is also frequently tracked. The most common cross-app Flow Permissions are those that leverage CPs, specifically CPs identified by URIs pertaining to contacts. The vast majority of the Flow Permissions generated for this CP (90%), use the CP as a source via the query method. The most common sink for these Flow Permissions is `SMS`.

1) Transmitting SIM Card and Phone Data: Currently in Android there is no easy way to anonymously identify a phone. Apps skirt this issue typically in two manners: 1) user login: the user must login to gain the benefit of leveraging their usage history within the app, and 2) Phone State or SIM Card data: the app accesses and transmits the low-level identification of the phone (IMEI number), the low-level identification of the SIM card (ICCID number), or the international mobile subscriber identity (IMSI number) of the user. Flow Permissions provide a mechanisms to distinguish between these two methods as different Flow Permissions are generated in each case. For the former, `IME` $\rightarrow$ `NETWORK`, is generated by Blue Seal and corresponds to textual data input by the user (IME) being sent across the network. In the later cases the following Flow Permissions are generated: `IMEI NUMBER` $\rightarrow$ `NETWORK`, `ICCID` $\rightarrow$ `NETWORK`, or `IMSI NUMBER` $\rightarrow$ `NETWORK`. The IMSI number should be transmitted as rarely as possible and third party apps should not typically utilize this number. An app that is not provided by the user's service provider that has this Flow Permission should be considered malicious. We note, however, that user logins are not a universal solution. As an example, consider PhoneLab at SUNY Buffalo [13], a state of the art smartphone testbed with over 200 users. Participants run the PhoneLab app in the background, which collects pertinent statistics on the data and telephony usage. Data is collected for scientific purposes and PhoneLab *must* be able to distinguish between devices.

2) Incorrect Usage of the Log: Android provides a logging service called *logcat* that apps leverage for debugging. However, we believe that no app should use logcat at run time as it is a form of permanent storage that malicious apps can potentially access [14]. Based on our Flow Permissions, we have discovered that most apps use logcat and the most commonly stored sources are location and IMEI. To address this problem, Google has recently changed its permission mechanisms to restrict the `READ_LOG` permission to vetted system apps.

E. User Study

To test the utility of Flow Permissions, we created a user survey and tested first year masters and PhD students taking an Android based distributed systems course. Our survey results were obtained anonymously over 61 participants. The survey presented a description of an anonymous app and its requested permissions. Students then responded how likely they were to install the app. The same question was asked including our Flow Permissions synthesized for the app. At the end of the survey, the anonymous app was revealed and the students were once again asked how likely they were to install the app.

Table III presents the results of our survey. The percentages shown in the table correspond to the percent of answers that corresponded to "Likely" and "Very Likely" for installation of the app. The first column presents results of the anonymized app with standard Android permissions. The second column shows results for our Flow Permission mechanisms for the anonymized app and the last column shows how the answers change once the app name is revealed.

Our results indicate that Flow Permissions can significantly impact users decisions to install an app when the users are unbiased—users do not have any preconceived notions about

TABLE III
USER SURVEY RESULT SHOWING HOW LIKELY THE USER IS TO INSTALL THE APP.

App Name	Android Anonymous	Flow Permissions Anonymous	Android Named
Twitter	24.5 %	4.8 %	37.6 %
DropBox	73.7 %	29.4 %	63.9 %

the app or the developer of the app. Flow Permissions have a minor impact (DropBox installation rate was reduced by about 10%) on biased users. Although these results are preliminary, they do give a positive indication that Flow Permissions can be useful in a real-world setting, especially when users are not familiar with an app or its developer.

VI. RELATED WORK

The growing popularity of Android has resulted in many tools, case studies, and analysis engines. The most closely related work to ours is CHEX [7], which provides a tool for detecting highjack enabling flows with an app. It is the first tool to tackle analysis of Android's constructs such as async tasks and handlers, though it uses a brute force permutation approach for disambiguation. Our call graph restructuring described in Section IV-B can refine CHEX's approach since we identify implicit calls in Android's constructs whenever possible. AndroidLeaks [15] is a static analysis tool implemented in WALA that can find leaks of sensitive information sent over the network from Android apps. It does not support analysis of async tasks, intents, nor content providers and is unable to track cross-app flows. SCanDroid [16] first proposed a methodology for analyzing intents statically, but was never tested on real-world apps. The approach also required the original Java source of the programs. Mann *et al.* created a framework to identify privacy leaks from the Android APIs [17], but the framework has not been evaluated on real-world applications. DroidChecker [18] is a static analysis tool aimed at discovering privilege escalation attacks and thus only analyzes exported interfaces and APIs that are classified as dangerous. ScanDal [19] is an abstract interpretation framework for tracking information flows within apps. Currently, their framework is able to track flows between location information, phone identifiers, camera, and microphone exported to the network and SMS.

Besides static analysis tools, there is a plethora of tools that perform dynamic analyses. Alazab *et al.* [20] provide a dynamic analysis technique that runs apps in a sandbox and can detect malicious apps. MockDroid [21] is a tool that protects users' privacy by supplying mock data instead of sensitive data. Aurasium [22] provides user-level sandboxing and policy enforcement to dynamically monitor an app for security and privacy violations. Notably Aurasium does not require modifications to the underlying OS. We believe that Aurasium is complementary to Blue Seal, as our Flow Permissions can provide a specification of possible malicious leaks. CrowDroid [23] is an offline analysis over traces that can be leveraged to identify malicious apps through examining their behavior via. crowdsourcing. Moonsamy *et al.* [24] provided a thorough investigation and classification of 123 apps using static and dynamic techniques over the apps' Java source code. Grace *et al.* [8] showed that ad frameworks opportunistically scan and leverage permissions granted by the app they are called from. AdDroid [25] introduced a new advertisement framework with privilege separation, accomplished through a new set of advertising APIs and permissions. We believe our tool can be extended to analyze their framework through extensions to the permission map Blue Seal takes as parameter. PiOS [26], a static analysis tool for iOS, leverages reachability analysis on control-flow graphs to detect leaks.

Porscha [27] is a tool to extend Android with policy oriented secure content handling to enforce DRM policies. Chen *et al.* [28] propose identification of malware by the temporal order in which an application uses APIs through static analysis and model checking. Panorama [29] is a malware detection system that leverages a graph based approach for dynamic taint analysis. It focuses on identifying malware based on access and processing patterns of sensitive information. We believe this approach is complimentary to the analysis for Flow Permissions and we envision extending our static analysis to approximate how much sensitive data is accessed.

Although Android has a comprehensive permission mechanism, it has limitations. Most users do not understand what each permission means [30], [31] and blindly grant them [31]. These studies have shown that the Android permission mechanism is not effective as a *protection mechanism* and suggest allowing users to grant individual permissions [32], blocking and sanitizing sensitive data [33], designing an app verification mechanism [34], and analyzing apps to report over-privilege [1]. Fragkaki *et al,* [35] propose an extension to the Android permission mechanism for disallowing of flows of the form: `disallow-flow(A,B)`, and shows how interesting policies can be built on top of such a mechanism. We believe our synthesized Flow Permissions could be leveraged to conservatively check the adherence of an app to such policies statically.

Previous case studies [31], [30] have reported that comprehension of permissions is reduced primarily due to the "presentation" of the permissions and not the mechanism itself. We believe our Flow Permissions can benefit from new presentation styles as they are developed.

VII. CONCLUSION AND FUTURE WORK

In this paper, we present a flow based extension to the Android permission mechanisms, called Flow Permissions. We detailed a comprehensive primer on Android specific mechanisms and libraries in our description of Blue Seal, an automated infrastructure for synthesizing Flow Permissions. We provided a comprehensive evaluation of Flow Permissions in a wide variety of Android apps as well as a preliminary user study indicating the utility of Flow Permissions on users' decision to install apps. For future work, we plan on leveraging Soot's advanced features to improve Blue Seal's precision, including path and context sensitivity.

REFERENCES

[1] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, 2011.

[2] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: attacks and defenses," in *Proceedings of the 20th USENIX conference on Security*, SEC'11, 2011.

[3] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pp. 13–, IBM Press, 1999.

[4] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing java bytecode using the soot framework: Is it feasible?," in *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, (London, UK, UK), pp. 18–34, Springer-Verlag, 2000.

[5] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP '12, (New York, NY, USA), pp. 27–38, ACM, 2012.

[6] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, 2012.

[7] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, 2012.

[8] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, (New York, NY, USA), pp. 101–112, ACM, 2012.

[9] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (Oakland), 2012 IEEE Symposium on*, 2012.

[10] "Amazon ec2." http://aws.amazon.com/ec2/.

[11] "dex2jar." http://code.google.com/p/dex2jar/.

[12] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.

[13] "Phonelab: A large-scale participatory smartphone testbed." http://www.phone-lab.org.

[14] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proceedings of the 20th USENIX conference on Security*, SEC'11, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2011.

[15] M. S. Ware and C. J. Fox, "Securing java code: heuristics and an evaluation of static analysis tools," in *Proceedings of the 2008 workshop on Static analysis*, SAW '08, (New York, NY, USA), pp. 12–21, ACM, 2008.

[16] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications."

[17] C. Mann and A. Starostin, "A framework for static detection of privacy leaks in android applications," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, (New York, NY, USA), pp. 1457–1462, ACM, 2012.

[18] P. P. Chan, L. C. 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, (New York, NY, USA), pp. 125–136, ACM, 2012.

[19] J. Kim, Y. Yoon, K. Yi, and J. Shin, "ScanDal: Static analyzer for detecting privacy leaks in android applications," in *MoST 2012: Mobile Security Technologies 2012* (H. Chen, L. Koved, and D. S. Wallach, eds.), (Los Alamitos, CA, USA), IEEE, May 2012.

[20] M. Alazab, V. Monsamy, L. Batten, P. Lantz, and R. Tian, "Analysis of malicious and benign android applications," in *Proceedings of the 2012 32nd International Conference on Distributed Computing Systems Workshops*, ICDCSW '12, (Washington, DC, USA), pp. 608–616, IEEE Computer Society, 2012.

[21] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, (New York, NY, USA), pp. 49–54, ACM, 2011.

[22] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: practical policy enforcement for android applications," in *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, 2012.

[23] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, (New York, NY, USA), pp. 15–26, ACM, 2011.

[24] V. Moonsamy, M. Alazab, and L. Batten, "Towards an understanding of the impact of advertising on data leaks," *Int. J. Secur. Netw.*, vol. 7, pp. 181–193, Mar. 2012.

[25] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: privilege separation for applications and advertisers in android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '12, (New York, NY, USA), pp. 71–72, ACM, 2012.

[26] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *NDSS*, The Internet Society, 2011.

[27] M. Ongtang, K. Butler, and P. McDaniel, "Porscha: policy oriented secure content handling in android," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, (New York, NY, USA), pp. 221–230, ACM, 2010.

[28] K. Z. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. X. Wu, M. Rinard, and D. Song, "Contextual policy enforcement in android applications with permission event graphs," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*, (San Diego, USA), February 2013.

[29] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, (New York, NY, USA), pp. 116–127, ACM, 2007.

[30] S. Egelman, A. P. Felt, and D. Wagner, "Choice Architecture and Smartphone Privacy: There's A Price for That," in *Proceedings of the 11th Annual Workshop on the Economics of Information Security (WEIS)*, 2012.

[31] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android Permissions: User Attention, Comprehension, and Behavior," in *Proceedings of the 8th Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[32] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[33] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications," in *Proceedings of the 18th ACM Conference on Computer and communications security (CCS)*, 2011.

[34] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," in *Proceedings of the 16th ACM Conference on Computer and communications security (CCS)*, 2009.

[35] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey, "Modeling and enhancing androids permission system," in *Computer Security ESORICS 2012* (S. Foresti, M. Yung, and F. Martinelli, eds.), vol. 7459 of *Lecture Notes in Computer Science*, pp. 1–18, Springer Berlin Heidelberg, 2012.