Information Flows as a Permission Mechanism

Feng Shen, Namita Vishnubhotla, Chirag Todarka, Mohit Arora,
Babu Dhandapani, Eric John Lehner, Steven Y. Ko, Lukasz Ziarek
University at Buffalo, The State University of New York
{fengshen, namitavi, chiragto, marora, babupras, ericj, 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 per-app information flows within an application (*e.g.*, a permission for reading the phone number and sending it over the network) as well as cross-app 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. In order to support Flow Permissions on Android, we have developed a static analysis engine that detects flows within an Android application. We have also modified Android's existing permission mechanism and installation procedure to support Flow Permissions. We evaluate our prototype with 2,992 popular applications and 1,047 malicious applications and show that our design is practical and effective in deriving Flow Permissions. We validate our cross-app flow generation and installation procedure on a Galaxy Nexus smartphone.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers*; C.5.3 [**Computer System Implementation**]: Microcomputers—*portable devices(e.g., laptops, personal digital assistants)*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*

Keywords

Android; Permissions; Information Flows

1. INTRODUCTION

Modern mobile OSes such as iOS and Android provide permission mechanisms, allowing users to review how an application ("app") accesses the resources on a mobile device. Android, in particular, has a comprehensive permission mechanism; at development time, an app writer needs to explicitly request permissions by statically declaring them in an app configuration file (`AndroidManifest.xml`). During installation, a user needs to review the permissions that an app requests and explicitly grant them.

Currently, there are over 130 permissions which Android apps can request in API level 17. Generally, an app 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 app wants to use APIs that control the camera, it needs to request `android.permission.CAMERA`.

Although considered to be robust, the current permission mechanism of Android provides little contextual information on how permissions of an app 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 also communicate with one another via Android's IPC mechanisms to effectively gain permissions they were not explicitly given, thereby bypassing the current permission mechanism [18].

To address these issues, we propose a new permission mechanism, called *Flow Permissions*, that extends the existing Android permission mechanism with *information flows* between permission domains (*e.g.*, reading from the SIM card and sending over the network). Our Flow Permissions identify single-app flows, *i.e.*, information flows within an app, as well as cross-app flows, *i.e.*, information flows across apps via IPC mechanisms. In order to synthesize single-app flows, we develop an automated static analysis engine that detects information flows within an Android app. To synthesize cross-app flows, we modify Android to perform cross-app permission analysis when installing a new app. This cross-app permission analysis compares information flows within the new app to those of already-installed apps and derives new Flow Permissions. This combination of static and installation-time analysis comprises BlueSeal, our Flow Permission synthesis system.

More specifically, this paper makes the following contributions:

- **Flow Permissions:** We propose a new permission mechanism based on information flows between permission domains within an app, as well as across multiple apps. Cross-app flow detection is leveraged at

ASE'14, September 15-19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2643018.

installation time to alert the user of possible interactions between apps.

- **BlueSeal:** We develop a holistic system, called BlueSeal, for automatically generating Flow Permissions and building them into Android. The design of BlueSeal shows a primer on how to modify classic program analyses to statically analyze Android specific constructs. BlueSeal statically generates per-app Flow Permissions, and at installation time, it generates cross-app Flow Permissions. We augment Android's package installer to perform cross-app permission analysis. We validate the feasibility and effectiveness of our implementation by showing experimental results on a Galaxy Nexus smartphone.

- **Case Studies:** Detailed performance analysis including a comparison study with state-of-the-art tools, validation against 96 known benchmark apps, manual introspection, as well as a large validation across 2,992 popular and 1,047 malicious apps.

This paper is an extended version of our previous 6-page ASE 2013 new idea paper [22]. Our previous paper introduces Flow Permissions as a new permission mechanism and shows preliminary results of BlueSeal. We extend our previous paper with a more thorough discussion of BlueSeal, an augmented package installer, and new case studies, and performance results.

2. OVERVIEW

In our previous new idea paper [22], we have discussed the shortcomings of the current permission mechanism of Android and proposed our new Flow Permission mechanism. This section summarizes our previous paper.

2.1 Limitations of the Current Mechanism

In order to install an app on Android, a user needs to explicitly grant one or more permissions requested by the app. For example, Gmail, a popular mail client on Android, requests many permissions such as `INTERNET` and `STORAGE`. The `INTERNET` permission, once granted, allows the app to send and receive packets over the network. The `STORAGE` permission allows the app to read from and write to the phone's storage. There are other kinds of permissions such as `PHONE STATE`, which allow sensitive, personal data to be read and written, *e.g.*, the phone number and the device ID.

This means that if a user grants a permission to read the phone's device ID and as well as access to the Internet, the user is also implicitly granting permission to transmit the device ID over the Internet to an external entity. In general, 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. The problem is that the current permission mechanism offers no insight to make the connection between different permissions.

In addition, it is known that multiple apps can gain permissions *implicitly* without requiring users to explicitly grant them [18]. This is possible since Android's IPC mechanisms do not necessarily check app permissions. For example, suppose that app_1 has permission to access the network, and app_2 has permission to read the device ID. In Android, app_1 can expose its network access capability through an IPC mechanism, allowing app_2 to leverage app_1 to send the device ID over the network without explicitly requesting the network permission.

2.2 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* → *sink*. For example, if an app has a data flow between the `PHONE STATE` source and the `INTERNET` sink, then the corresponding Flow Permission is `PHONE STATE` → `INTERNET`. This means that the app can potentially read the phone state (*e.g.*, the phone number) and subsequently exports through the use of the 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 device ID to the network as this provides the app a mechanism to uniquely identify the device for analytics. Regardless, some users may not be comfortable providing such information to the app developer, as other mechanisms (*e.g.* manual login) can be used without exposing such data.

2.3 Flow Permission Mechanism

Flow Permissions are an extension to the Android permission mechanism that characterizes the *implicit* interactions between data and APIs protected by standard permissions. As described in the rest of the paper, we derive fine-grained information flows using static analysis. We also derive cross-app flows by combining per-app flows across different apps. This is done by matching one app's sinks to another app's sources.

We display these flows to users at installation time so that users can examine the flows present in an app. Since it is possible that an app has many flows, we categorize sources and sinks into *domains* to reduce the number of flows that need to be shown to the users. We currently have thirteen source domains: `SMS`, `STORAGE`, `HISTORY BOOKMARKS`, `USE DICTIONARY`, `FINE LOCATION`, `COARSE LOCATION`, `CALENDAR`, `ACCOUNTS`, `PHONE STATE`, `CONTACTS`, `CALL LOG`, `VOICEMAIL`, and `LOG`. Similarly, we use five sink domains: `NETWORK`, `LOG`, `MMS`, `STORAGE`, and `INTENT`. Our previous paper describes how we determine these domains [22].

3. BLUESEAL DESIGN

Our system, BlueSeal, depicted in Fig. 1, is comprised of two main components, the static analysis engine—which performs per app analysis offline, and the cross-app analysis engine—which performs cross-app analysis at installation time on the phone itself. Our static analysis engine is built on top of the Soot Java Optimization Framework [33, 34]. Since Soot was originally developed for analyzing Java bytecode, it was extended recently to transform DEX bytecode into Soot's own intermediate representation (Jimple), using Dexpler [7]. We extend Soot to leverage the PScout Permission Map [5]; abstractly, a permission map is a mapping between Android API calls and the permissions required to enact those calls. The PScout Permission Map was gener-

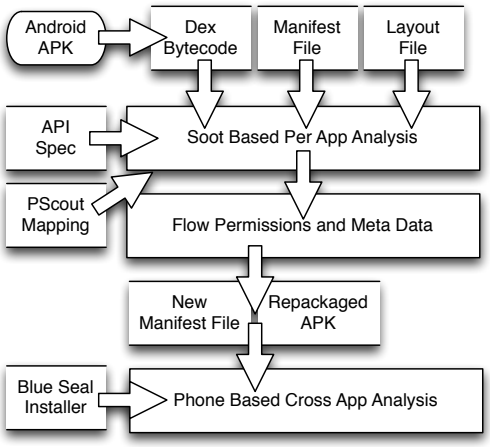

Figure 1: The BlueSeal Android app analysis framework architecture.

ated 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, BlueSeal leverages classic forward and backward intraprocedural dataflow analysis as well as interprocedural dataflow analysis based on graph reachability. BlueSeal 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, and (5) interprocedural permission flow analysis. We augment Android's package installer to perform the 6th pass (cross-app permission flow analysis) as described in Section 3.6. Abstractly, BlueSeal uses analyses (2), (3), and (4) to disambiguate Android specific constructs and to identify source and sink points, prior to tracking flows between sources and sinks in analysis (5). BlueSeal implements Stowaway's unused permission analysis [15] to remove unnecessary permissions. Since BlueSeal is built from classic analysis techniques, we tailor our discussion on how to support Android specific linguistic constructs, libraries, and IPC mechanisms in standard analyses. Currently, BlueSeal is not path or context sensitive, and is subject to the precision of the analyses from which it is constructed.

Since our previous new idea paper [22] mainly discusses analysis steps (1) and (5) (*i.e.*, entry point and interprocedural permission flow analyses), we omit the details here. Briefly, our entry point discovery deals with the event-driven nature of Android programming; we discover UI callbacks (*e.g.*, a button-click callback); we detect standard framework components of Android such as `Activity`, `Service`, `BroadcastReceiver`, and `ContentProvider`, that essentially replace traditional `main()`. We have also crawled the online API documentation of Android and discovered 1,738 callback methods that can serve as entry points (for API 17).

Our interprocedural flow analysis uses a fixed point algorithm, leveraging the standard work list model and method summaries. 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. Once

```
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) {
            ...
        }
    }
}
```

Figure 2: A code snippet illustrating the methods that comprise the control flow of an `AsyncTask` in Android and the implicit flow of arguments provided by the Android framework.

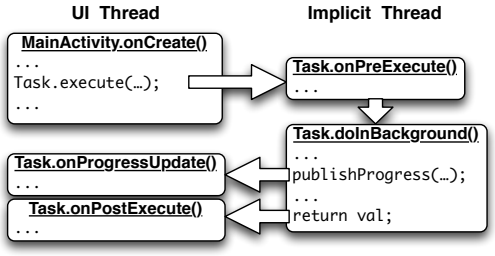

Figure 3: The execution flow of `AsyncTask` methods in their respective threads at runtime.

method summary construction reaches a fix point, we synthesize a global flow graph from the per-method summaries, giving us all potential flows across methods. Per-app 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).

3.1 Call Graph Restructuring

The Android framework is responsible for 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 dicuss this in detail below.

3.1.1 AsyncTask

`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 `onCanceled`, 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 compu-

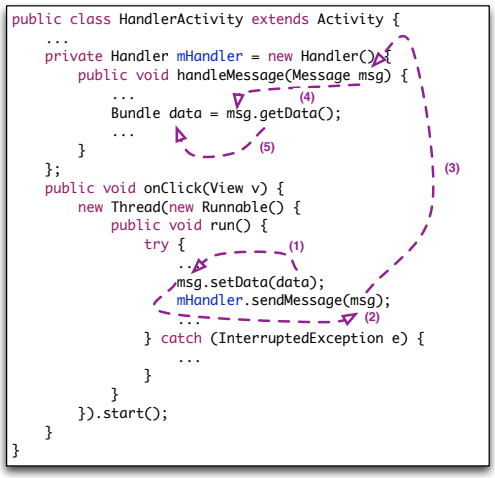

Figure 4: Flows based on pairing message sends to the appropriate message handlers.

tation for the `AsyncTask`. The methods `onPreExecute` and `onPostExecute` run before and after `doInBackground` and typically include pre- and post-processing. The `onCanceled` method is called when the `AsyncTask` is canceled by another thread. Notice that `onPreExecute` will execute in the implicitly created thread backing the `AsyncTask`, but the `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 `AsyncTask` 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.

3.1.2 *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 [1] of methods. Similar to `AsyncTask`, BlueSeal effectively replaces a call to `sendMessage*` with a call to `handleMessage` to restructure the call graph.

3.2 Content Provider Resolution Analysis

After restructuring the call graph, BlueSeal performs additional analyses to identify permission sinks and sources.

[1]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*`).

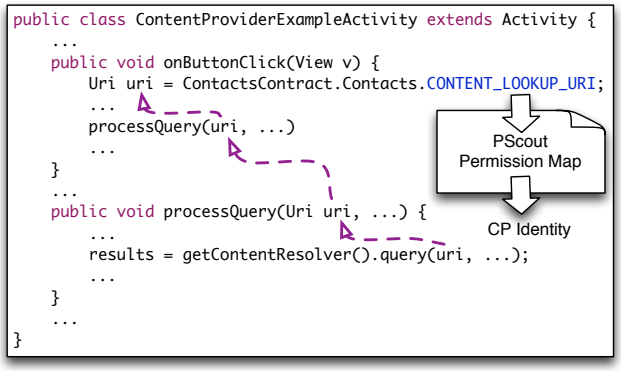

Figure 5: The data flow of an URI object that identifies which content provider is being utilized. Dashed arrows indicate information derived from dataflow analyses and block arrows how that information is used to disambiguate the content provider.

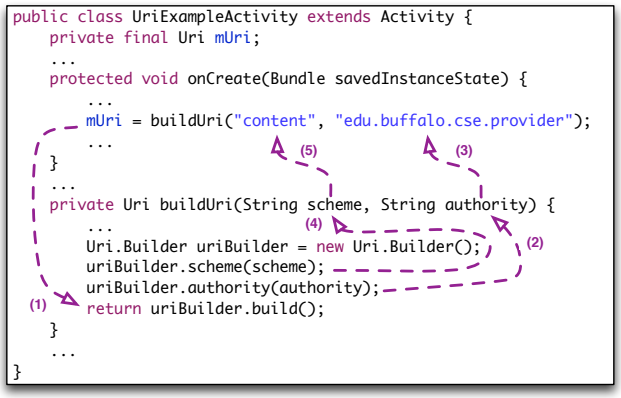

Figure 6: The data flow of an URI object initialization that is resolvable statically.

One mechanism for interaction between apps is `ContentProvider`. An app can provide content to itself or other apps can consume content hosted by a content provider, or both. Content providers are uniquely identified by a URI object and to correctly pair uses of content providers, these objects must be tracked and disambiguated to the extent possible by static analysis. To identify uses of content providers, we track the content provider API calls as well as the URI Objects (as shown in Fig. 5). Our Content Provider Resolution Analysis (CPRA) is based on an interprocedural dataflow analysis that leverages a backward intraprocedural data flow analysis. Abstractly, we track backward flows from uses of the content provider mechanism to the definitions of URI objects and from the definitions of URI objects to the strings that uniquely identify them.

Content providers 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

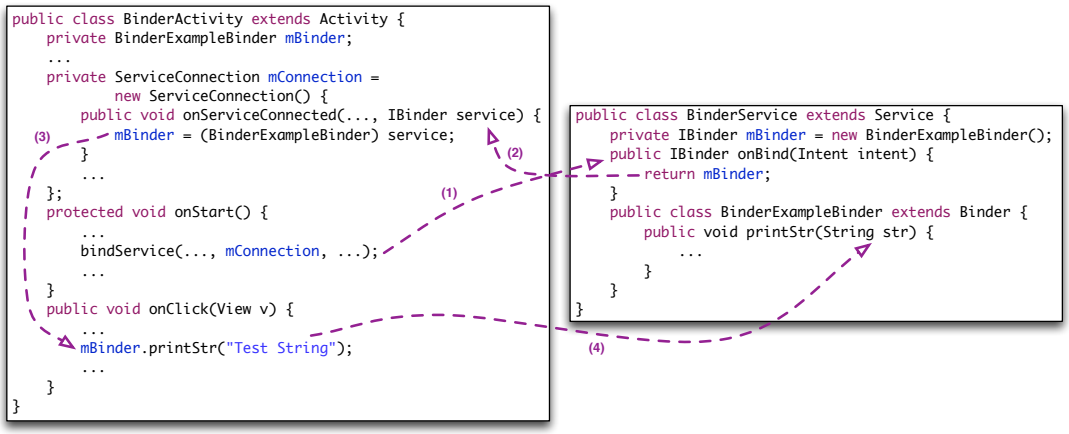

Figure 7: Data and control flow relations between a binder activity and service.

identifying URI string is hidden. For precision, our analysis leverages PScout, which 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 content provider identifier where `content` is the scheme and `edu.buffalo.cse.provider` is the authority. After the scheme and the authority are set, the actual URI object is returned by `Uri.Builder.build`. Thus, our analysis tracks calls to `scheme` and `authority` and the arguments passed to them as shown in Fig. 6.

3.3 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. 7 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 3.1.2 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`).

3.4 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 without declaring it in its manifest file.

BlueSeal performs Intent Resolution Analysis (IRA) in much the same way as CPRA. Namely, it identifies possible sources and sinks related to the intents by examining relevant API calls such as `put*Extra`, `setData*`, `get*`, `send*Broadcast*`, `startActivity*`, *etc.*, that are capable of reading from, writing to, sending, and receiving intents. In general, handling intents more precisely is a much broader problem and Octeau *et al.* [29] provide a mechanism for improved precision. We are currently exploring how to integrate their techniques into BlueSeal.

3.5 Repackaging the Application

Once the Flow Permissions are derived for an app, BlueSeal appends these permissions to the list of permissions that the app requests. It does this by first extracting the manifest file (`AndroidManifest.xml`) of the app and modifying it. It then repackages the application file (`apk`) for distribution. Finally, it uses newly-added Flow Permissions for display at installation time as well as for cross-app analysis done on a phone, described next.

3.6 Augmented Package Installer

In order to display Flow Permissions and perform cross-app analysis, we have augmented Android's package installer in three ways. First, we have added all our Flow Permissions

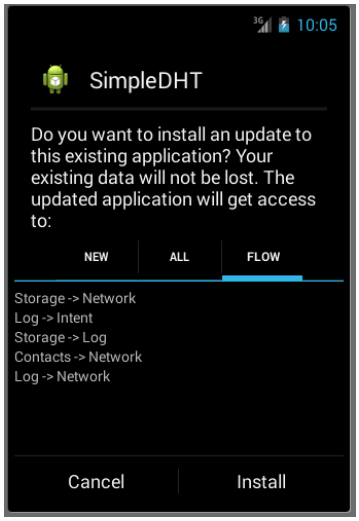

Figure 8: Flow Permission example screenshot

to the source manifest file (the framework's `AndroidManifest.xml`) that the package installer accesses during installation. This is the global list of all permissions available in the system. Second, we display our Flow Permissions by modifying the package installer. Fig. 8 shows an example of how we display Flow Permissions to users.

Lastly, we implement cross-app analysis in the package installer in its `PackageInstallerActivity`. In our implementation we synthesize cross-app permissions by performing all-to-all matching between all existing flows from already-installed apps, and flows from the app being installed. For example, while installing app_2, if app_1 (already installed) has a flow from the device ID to a file, and app_2 has a flow from the same file to a socket, a new cross-app Flow Permission, `PHONE STATE` $\rightarrow$ `NETWORK`, is created and displayed. A similar matching is done between the derived sources and sinks of already-installed apps. For this analysis, our new package installer stores all permissions from all already-installed apps in its storage. Our performance results in Section 4 demonstrate that our all-to-all comparison is still practical and feasible to run on a smartphone.

4. RESULTS AND DISCUSSION

To validate our approach, we tested BlueSeal on 2,992 of the top-rated free apps available on the Google Play Store, with 571 apps from January 2013, 2,421 apps from January 2014, and on 1,047 known malicious apps from the MalGenome Project [2] [40]. We ran BlueSeal on Amazon EC2 [1] using an 13-ECUs and 4-vCPUs node instance with 15GB of RAM. In the set of apps, there are 107 apps not analyzed because the Soot framework, which BlueSeal relies on, threw exceptions when performing intermediate representation transformation. BlueSeal, thus, was able to analyze 2,885 apps. Our full data set and results can be found at `http://blueseal.cse.buffalo.edu/data.html`.

The main purpose of our evaluation is to assess the analysis capability and usefulness of BlueSeal. We do this in four ways. First, we present aggregated as well as categorized statistics regarding information flows and Flow Permissions. Second, we present the analysis performance of BlueSeal.

[2] `http://www.malgenomeproject.org`.

Results Overview	Count
Total apps	2,885
Total number of raw flows detected	631,152
Avg number of raw flows per app	218.77
Distinct flows	2374
Total number of Flow Permissions	17,332
Avg number of Flow Permissions per app	6.01
Distinct Flow Permissions	431

Table 1: A brief overview of experiment results

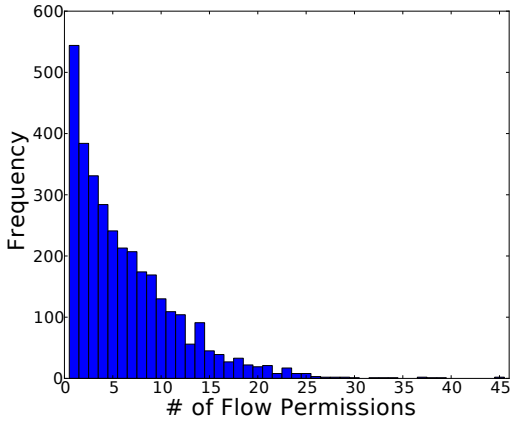

Figure 9: Distribution of Flow Permissions generated for each app. 630 apps do not generate any Flow Permission and are not shown.

Third, we discuss the limitations of BlueSeal with manual validation. Fourth, we show the usefulness of BlueSeal with a user survey.

4.1 Statistics of Flow Permissions

Table 1 shows the overview of our results. Although we detect many information flows in an app, the number of Flow Permissions is significantly smaller due to our domain categorization (described in Section 2). Fig. 9 shows the distribution of Flow Permissions generated by BlueSeal. Most of the apps contain less than 15 Flow Permissions, an amount practical for a user to examine. The app with the maximum number of Flow Permissions (45) contains a heavy usage of different content providers, currently distinguished based on their URIs. We plan on categorizing content providers in much the same way as permission domains. The app that contains the maximum number of raw flows (13,646 flows as shown in Figure 10) generates twenty Flow Permissions.

Table 2 shows the ten most common flows observed from normal apps and Table 3 shows the ten most common flows observed in malicious apps. From these two tables, we make four observations. First, normal apps are more concerned about users' input data since most normal apps require a user's login information and many provide social communication functionality that requires a lot of user input. On the other hand, malicious apps are heavily interested in the phone's unique identifier, *DeviceId*. Second, both normal and malicious apps read system content providers often. The most commonly accessed content provider in both normal and malicious apps is the contacts. Third, normal apps access the phones' location data more frequently than

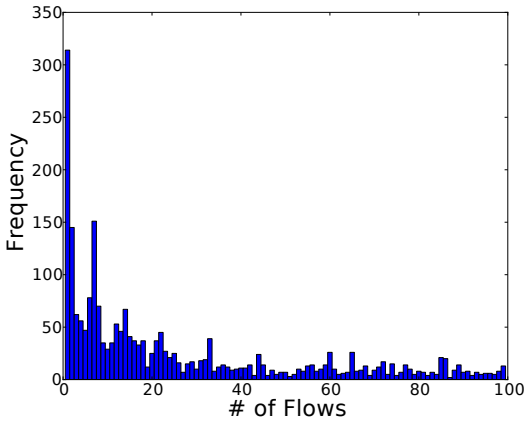

Figure 10: Distribution of raw flows generated for each app. (1046 apps range from 100 to 13,646 and 630 apps with 0 flow are omitted.)

Count	Raw Flow
100379	EditText:getText→ Intent:putExtra
49910	ContentResolver:query→ Intent:putExtra
18512	EditText:getText→ Log:e
16937	Location:getLatitude→ Intent:putExtra
16756	Location:getLongitude→ Intent:putExtra
15135	EditText:getText→ Log:d
10660	ContentResolver:query→ Log:e
9191	Location:getLongitude → DataOutputStream:writeShort
9173	Location:getLatitude → DataOutputStream:writeShort
9053	EditText:getText→ PrintStream:println

Table 2: Ten most common flows in normal apps.

malicious apps. Fourth, most of the flows in normal apps indicate that data is often used inside the app, while in malicious apps data mainly flows to the network and storage. This observation suggests that normal apps leverage the sensitive data for debugging purposes while malicious apps may store or send the data. Last, the two location flows in Table 2 show that location is often sent within intents; this is in fact a common design pattern that many developers use. An app gets a location update, wraps it in an intent, and sends the intent to itself to display the update. BlueSeal currently does not distinguish whether or not an app sends intents to itself; it just detects that there is a flow to an intent that gets sent out. To improve BlueSeal's precision, we can leverage existing techniques such as the ones implemented in Epicc [29].

In Tables 4 and 5, we collect top ten sources used by normal and malicious apps, respectively. As shown in these two tables, there are three main categories of sources frequently accessed by both normal and malicious apps—*system content providers*, *phone identifier*, and *location*. Specifically, normal apps use the users' location data while malicious apps read the phone identifier and data stored in system content provider, which is often private (*e.g.*, contacts).

Table 6 and Table 7 show the top ten sinks present in normal and malicious apps, respectively. We observe that there are mainly three categories of sinks in normal apps: log, storage, and intent. In malicious apps, the top sinks are log, intent, storage, and network. Logging, which is used for debugging purposes, is the most frequently used sink in both normal and malicious apps.

Count	Raw Flow
2757	TelephonyManager:getDeviceId→ Log:d
2577	TelephonyManager:getSubscriberId→ Log:d
739	TelephonyManager:getDeviceId→ Log:e
588	TelephonyManager:getDeviceId → HttpClient:execute
584	TelephonyManager:getDeviceId → ByteArrayOutputStream:write
569	ContentResolver:query→ Log:i
561	ContentResolver:query→ Log:d
507	TelephonyManager:getSubscriberId→ Log:e
476	TelephonyManager:getDeviceId→ Intent:putExtra
448	COntentResolver:query→ FileOutputStream:write

Table 3: Ten most common flows in malicious apps.

Count	Source Type
205706	EditText:getText
123870	ContentResolver:query
77009	Location:getLatitude
76105	Location:getLongitude
43911	LocationManager:getLastKnownLocation
37561	TelephonyManager:getDeviceId
13820	FileInputStream:read
5816	AccountManager:getAccountsByType
4296	AutoCompleteTextView:getText
3815	TelephonyManager:getLine1Number

Table 4: Top-10 sources in normal apps.

4.2 Uses of the Statistics

Using these statistics, we can make observations about the difference between malicious apps and normal apps in terms of information flows. For this purpose, we have classified malicious apps into different categories according to their malware type, *e.g.*, AnserverBot, BeanBot, DroidKungFu3, DroidKungFu4, GoldDream, *etc.* We then examined what common or distinct flows exist across the categories. A visualization of this data can be found for all categories at `http://blueseal.cse.buffalo.edu/flows.png`.

With this flow analysis, we have discovered that there are a few flows that highlight the difference between malicious apps and normal apps. Table 8 shows the number of apps in malicious and normal categories. It summarizes the flow analysis result for one specific flow—the flow from `TelephonyManager:getLine1Number` to `(network)OutputStream:write`. This flow is used to ex-filtrate the user's phone number using Java's Socket IO. We observe that this flow is only used in 11 out of the 2198 normal apps which have more than one flow. This flow exists with a much greater frequency in the DroidKungFu3 and DroidKungFu4 categories. Further, we have found out that 3 out of the 11 *normal* apps with this flow have been removed from the Play Store since we originally downloaded them; these apps are `com.SuperQiang.SexyGirlWallpaper.apk`, `com.SuperQiang.SexyGirlWallpaper3.apk`, and `Muli.touch.Sex10009.apk`.

These individual flows can also be grouped to make a more specific match. Table 9 shows the number of apps in each malicious category and among all normal apps for the flow pair from `TelephonyManager:getLine1Number` to `(network)OutputStream:write` and the flow from `TelephonyManager:getDeviceId` to `HttpClient:execute`. The former flow is the same flow that we used in Table 8. The

Count	Source Type
6697	TelephonyManager:getDeviceId
4159	TelephonyManager:getSubscriberId
3939	ContentResolver:query
1190	Location:getLatitude
1188	Location:getLongitude
1169	LocationManager:getLastKnownLocation
1114	TelephonyManager:getLine1Number
1106	FileInputStream:read
883	EditText:getText
446	TelephonyManager:getCellLocation

Table 5: Top-10 sources in malicious apps.

Count	Sink Type
221831	Intent:putExtra
56584	Log:e
48814	Log:d
41747	DataOutputStream:writeShort
30808	DataOutputStream:writeUTF
20801	PrintStream:println
20065	OutputStream:write
19590	DataOutputStream:write
15576	Log:i
14523	Log:w

Table 6: Top-10 sinks in normal apps.

Count	Sink Type
6965	Log:d
2793	Log:e
1809	Log:i
1330	Intent:putExtra
1200	HttpClient:execute
1197	ContentResolver:insert
1176	ByteArrayOutputStream:write
973	OutputStream:write
939	Log:v
897	FileOutputStream:write

Table 7: Top-10 sinks in malicious apps.

Category	Distinct Count	Total Count
AnserverBot	1	172
BeanBot	7	7
DroidKungFu3	189	236
DroidKungFu4	73	82
GoldDream	3	40
Normal	11	2198

Table 8: Distinct malicious and normal apps using a flow, `TelephonyManager:getLine1Number` to `(network)OutputStream:write`

latter flow sends the device ID over to the network with an HTTP API. By flow pair we mean two flows that are held within the same app. In this case, each app counted contains a flow that is capable of sending the phone number off the phone using the Java Socket IO methodology and is capable of sending the phone's device ID over the network using Android's HttpClient. We observe that this flow pair is found in 6 out of 2198 normal apps which have any flow. Three of these apps were the same three that had been removed from the Play Store since our initial download. Of the remaining three, one is the free version of a popular guitar simulator app. The other two apps, `com.applock1` and `com.mm.security.androidhider1`, require root access and AVG Threat Labs indicate that they are malware. In the DroidKungFu4 category, the flow pair was found with the greatest frequency. More flows, in increasingly larger groups, can be used to narrow down the likelihood of an app being malicious, and possibly what malicious category the app would pertain to.

Certain flows can be even more telling of malicious activity. In many of the categories, flows with a source of `SmsManager:sendTextMessage` were found to be malicious, which can be seen in Table 10. One app in our normal group that used this as a source stood out as well, `com.bluecode.photo.space.effects.fx.apk`. This app is stated to add space effects to one's photos, and this flow may be construed as a way to send photos to friends via text messages. This assumption would be false though, as photos need to be sent via MMS using a built-in MMS app's API. While we wanted to test this app further, it has since been removed from the Play Store, with AVG's Threat Labs [2] reporting that this app contained adware.

4.3 Performance

BlueSeal is able to analyze and synthesize per-app Flow Permissions for all but the largest apps in under ten minutes. Only 163 apps require an analysis time greater than ten minutes (all of these 163 apps finish under 30 minutes). Fig. 11 shows the full performance results. However, Soot's front-end Dex bytecode parser, Dexpler, has limitations and generates incorrect intermediate representations for 107 of the apps. These apps are all from the Google Play store. We are currently investigating the causes of the mis-translation of the remaining 107 apps.

To measure the performance of our cross-app analysis, we have tested the installation performance of BlueSeal's augmented package installer with 44 random apps. In our experiment, we have installed each app on a Galaxy Nexus phone, one app at a time, without uninstalling previously-installed apps. We have measured the cross-app analysis time as well as the total time until the installer displays the installation screen. As Fig. 12 shows, it takes less than 2.1 seconds for all the apps to analyze cross-app permissions and display the installation screen. Synthesizing cross-app flows does not exceed 0.25 seconds for any of the app installs.

4.4 Manual Validation

False positives and false negatives are well-known limitations of static analysis, which also apply to BlueSeal. Thus, we manually validate BlueSeal in three ways to understand its limitations. First, we have compared against TaintDroid [13]—a custom Android OS that performs a dynamic taint analysis for identifying malicious flows. We have manually compared BlueSeal's generated Flow Permissions to TaintDroid's dynamically discovered taints on thirty apps. Each app was manually executed for 15 minutes and fed random key-presses. Unsurprisingly, the most common taints reported by TaintDroid mirrored our own findings and that of prior work. We have not discovered any taints reported by TaintDroid for which BlueSeal does not generate a corresponding Flow Permission.

Second, we have randomly chosen and inspected 100 apps that have exactly one BlueSeal-reported flow. We have examined each app's intermediate representation of the source

Category	Distinct Count	Total Count
AnserverBot	1	172
BeanBot	0	0
GoldDream	3	40
Normal	6	2198
DroidKungFu4	72	82
DroidKungFu3	188	236

Table 9: Distinct malicious and normal apps using a flow pair compared to distinct apps within a category

Category	Distinct Count	Total App Count
Bgserv	1	1
CoinPirate	1	1
DogWars	1	1
DroidKungFu3	3	236
Endofday	1	1
Geinimi	4	25
GamblerSMS	1	1
GPSSMSSpy	6	6
NickyBot	1	1
Normal	20	2198
SMSReplicator	1	1
Walkinwat	1	1

Table 10: Distinct malicious and normal apps using the SmsManager:sendTextMessage sink

and verified that BlueSeal detects actual flows in all 100 apps.

Lastly, we have vetted BlueSeal against DroidBench, an Android benchmark suite with 64 apps provided by FlowDroid [4]. BlueSeal can detect all the flows in DroidBench except implicit flows, since BlueSeal currently does not handle implicit flows. In addition, there are several apps for which BlueSeal reports false positives. This occurs for one of three reasons—flows in dead code, lack of context sensitivity, and flows in complex data structures.

BlueSeal reports flows in dead code since it does not perform any dead code analysis. BlueSeal simply relies on Soot to determine code reachability. Also, BlueSeal is not currently context sensitive; for example, if there is a flow from a source to a class variable, and another flow from the same class variable to a sink, BlueSeal reports that there is a flow, regardless of the relationship of the call sites in which those flows occur. Lastly, if an app has a flow from a source to a complex Java data structure such as `HashMap`, and a flow from the same data structure to a sink, BlueSeal reports that there is a flow. As a result, BlueSeal reports one flow in 12 apps in DroidBench that do not have any flow.

4.5 User Study

To test the utility of Flow Permissions, we created a user survey and tested graduate and undergraduate students taking computer science courses. These students are mixed majors (CS and non-CS). Our survey results were obtained anonymously with 540 participants. The survey procedure is as follows. (1) The survey presented a description of an anonymized app and its requested permissions. (2) Students then responded how likely they were to install the app. (3) The same question was asked including our Flow Permissions synthesized for the app. (4) At the end of the survey, the anonymized app was revealed and the students were once again asked how likely they were to install the app. We repeated the survey for two apps, Twitter and DropBox.

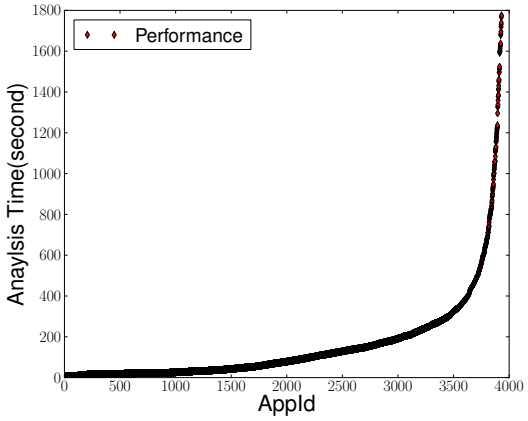

Figure 11: Scatter plot showing the time taken to analyze all apps in seconds.

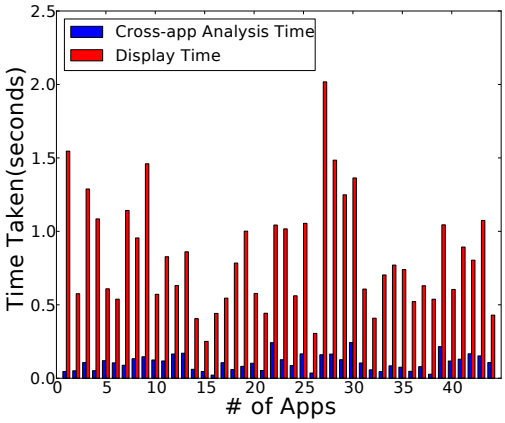

Figure 12: Performance of our cross-app analysis.

Table 11 presents the results of our survey. The percentages shown in the table show the likelihood of 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. The last column shows how the answers change once the app name is revealed.

Our results indicate that Flow Permissions can significantly impact user's decisions to install an app when the users are unbiased, *i.e.*, when users do not have any preconceived notions about the app or the developer of the app. Flow Permissions have a minor impact 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.

4.6 Threats to Validity and Discussion

We have used independent sources to obtain our testing apps, which helps us to validate BlueSeal without much selection bias. The first source is MalGenome Project that

App Name	Android Anonymized	Flow Permissions Anonymized	Android Named
Twitter	21 %	15 %	40 %
DropBox	37 %	15 %	35 %

Table 11: User survey result showing how likely the user is to install the app.

provides 1,047 malicious apps. The second source is the Google Play Store, where we downloaded 2,992 free apps. These apps include top-rated apps from all categories and randomly-selected apps. Selection bias can potentially come from MalGenome project's choice of apps as well as Google's categorization of apps. In addition, both CS and non-CS majors are represented in our user study, which also reduces the chance of selection bias.

5. RELATED WORK

The growing popularity of Android has resulted in many tools, case studies, and analysis engines. CHEX [25] provides a tool for detecting highjack enabling flows within an app. It is the first tool to tackle analysis of Android's constructs such as `AsyncTask` and `Handler`, though it uses a brute force permutation approach for matching call sites to destinations. Our call graph restructuring described in Section 3.1 can refine CHEX's approach since we identify implicit calls in Android's constructs whenever possible. Xiao *et al.* [36] propose a privacy-aware accesss control approach based on information flows. They employs the user-driven access control mechanism that allows users to choose among *real* information, *anonymized* information, or abort execution to protect users' privacy. Epicc [29] is another tool for statically analyzing Android apps. Their focus is precise analysis for intents to detect inter-component communication. They reduce the problem to an instance of the Interprocedural Distributive Environment (IDE) problem with less false positive rates. AndroidLeaks [35] 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 `AsyncTask`, `Intent`, nor `ContentProvider` and is unable to track cross-app flows. SCanDroid [20] 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 [26], but the framework has not been evaluated on real-world applications. DroidChecker [10] 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 [24] 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. FlowDroid [4] is an extendable data flow detection system for Android. Our analysis techniques are complementary to FlowDroid, as our techniques are designed to handle Android constructs specifically. Dendroid [32] is a tool used to automatically classify malware based on code structures. AppProfiler [31] developed a knowledge base with which to expand upon what applications are using the permissions for, which they call behaviors. They present informaiton on an applications behvaior to the user and received anonymous feedback on whether the behaviors were objectionable or not. A methodology for empirical analysis [6] was developed by this paper using self-organizing maps to visualize the Android permission mechanism.

Besides static analysis tools, there is a plethora of tools that perform dynamic analyses. Alazab *et al.* [3] provide a dynamic analysis technique that runs apps in a sandbox and can detect malicious apps. MockDroid [8] is a tool that protects users' privacy by supplying mock data instead of sensitive data. Aurasium [37] provides user-level sandboxing and policy enforcement to dynamically monitor an app for security and privacy violations. CrowDroid [9] is an offline analysis over traces that can be leveraged to identify malicious apps through examining their behavior via crowdsourcing. Moonsamy *et al.* [27] provided a thorough investigation and classification of 123 apps using static and dynamic techniques over the apps' Java source code. Grace *et al.* [21] showed that ad frameworks opportunistically scan and leverage permissions granted by the app they are called from. AdDroid [30] 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 BlueSeal takes as parameter. PiOS [11], a static analysis tool for iOS, leverages reachability analysis on control-flow graphs to detect leaks. AppIntent [38] instruments applications to provide GUI notifications indicating the events that lead to data leakage. VetDroid [39] a dynamic analysis platform designed to reconstruct permission use behaviors of Android applications, specifically for malware analysis.

Although Android has a comprehensive permission mechanism, it has limitations. Most users do not understand what each permission means and blindly grant them [12, 17]. These studies have shown that the Android permission mechanism is not effective as a *protection mechanism* [16] and suggest allowing users to grant permissions individually [28]; blocking and sanitizing sensitive data [23]; designing an app verification mechanism [14]; and analyzing apps to report over-privilege [15]. Fragkaki *et al,* [19] 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 [12, 17] 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.

6. CONCLUSIONS

In this paper, we have presented a flow-based extension to the Android permission mechanisms, called Flow Permissions. We have detailed a comprehensive primer on Android specific mechanisms and libraries in our description of BlueSeal, an automated infrastructure for synthesizing Flow Permissions. We have provided a comprehensive evaluation of Flow Permissions in a wide variety of Android apps both for single-app static analysis as well as cross-app analysis. Our evaluation shows that BlueSeal is practical to deploy and Flow Permissions provide visibility into the holistic behavior of mobile apps.

References

[1] Amazon ec2. `http://aws.amazon.com/ec2/`.

[2] Avg threat labs. `http://www.avgthreatlabs.com/android-app-reports/app/_pkg=com.bluecode.photo.space.effects.apk`.

[3] Moutaz Alazab, Veelasha Monsamy, Lynn Batten, Patrik Lantz, and Ronghua Tian. Analysis of malicious and benign android applications. In *Proceedings of the 2012 32nd International Conference on Distributed Computing Systems Workshops*, ICDCSW '12, pages 608–616, Washington, DC, USA, 2012. IEEE Computer Society.

[4] Steven Arzt, Siedfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android appstion in tcb source code. In *PLDI '14*, Edinburgh, UK, 2014.

[5] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, 2012.

[6] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 73–84, New York, NY, USA, 2010. ACM.

[7] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin 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, pages 27–38, New York, NY, USA, 2012. ACM.

[8] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 49–54, New York, NY, USA, 2011. ACM.

[9] Iker Burguera, Urko Zurutuza, and Simin 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, pages 15–26, New York, NY, USA, 2011. ACM.

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

[11] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Pios: Detecting privacy leaks in ios applications. In *NDSS*. The Internet Society, 2011.

[12] Serge Egelman, Adrienne P. Felt, and David 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.

[13] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[14] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and communications security (CCS)*, 2009.

[15] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, 2011.

[16] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the 2Nd USENIX Conference on Web Application Development*, WebApps'11, pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

[17] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the 8th Symposium on Usable Privacy and Security (SOUPS)*, 2012.

[18] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: attacks and defenses. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, 2011.

[19] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and enhancing androids permission system. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security – ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2012.

[20] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications.

[21] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza 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, pages 101–112, New York, NY, USA, 2012. ACM.

[22] Shashank Holavanalli, Don Manuel, Vishwas Nanjundaswamy, Brian Rosenberg, Feng Shen, Steven Y. Ko, and Lukasz Ziarek. Flow permissions for android. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013)*, 2013.

[23] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David 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.

[24] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. ScanDal: Static analyzer for detecting privacy leaks in android applications. In Hao Chen, Larry Koved, and Dan S. Wallach, editors, *MoST 2012: Mobile Security Technologies 2012*, Los Alamitos, CA, USA, May 2012. IEEE.

[25] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, 2012.

[26] Christopher Mann and Artem 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, pages 1457–1462, New York, NY, USA, 2012. ACM.

[27] Veelasha Moonsamy, Moutaz Alazab, and Lynn Batten. Towards an understanding of the impact of advertising on data leaks. *Int. J. Secur. Netw.*, 7(3):181–193, March 2012.

[28] Mohammad Nauman, Sohail Khan, and Xinwen 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.

[29] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, , Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security'13)*, 2013.

[30] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David 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, pages 71–72, New York, NY, USA, 2012. ACM.

[31] Sanae Rosen, Zhiyun Qian, and Z. Morely Mao. Appprofiler: A flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 221–232, New York, NY, USA, 2013. ACM.

[32] Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris-Lopez, and Jorge Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(4, Part 1):1104 – 1117, 2014.

[33] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–. IBM Press, 1999.

[34] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 18–34, London, UK, UK, 2000. Springer-Verlag.

[35] Michael S. Ware and Christopher J. Fox. Securing java code: heuristics and an evaluation of static analysis tools. In *Proceedings of the 2008 workshop on Static analysis*, SAW '08, pages 12–21, New York, NY, USA, 2008. ACM.

[36] Xusheng Xiao, Nikolai Tillmann, Manuel Fahndrich, Jonathan De Halleux, and Michal Moskal. User-aware privacy control via extended static-information-flow analysis. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 80–89, New York, NY, USA, 2012. ACM.

[37] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, 2012.

[38] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 1043–1054, New York, NY, USA, 2013. ACM.

[39] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 611–622, New York, NY, USA, 2013. ACM.

[40] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (Oakland), 2012 IEEE Symposium on*, 2012.