Multi-App Security Analysis with FUSE: Statically Detecting Android App Collusion[*]

Tristan Ravitch
tristan@galois.com
Galois, Inc

E. Rogan Creswick
creswick@galois.com
Galois, Inc

Aaron Tomb
atomb@galois.com
Galois, Inc

Adam Foltzer
acfoltzer@galois.com
Galois, Inc

Trevor Elliott
trevor@galois.com
Galois, Inc

Ledah Casburn
lcasburn@galois.com
Galois, Inc

ABSTRACT

Android's popularity has given rise to myriad application analysis techniques to improve the security and robustness of mobile applications, motivated by the evolving adversarial landscape. These techniques have focused on identifying undesirable behaviors in individual applications, either due to malicious intent or programmer error. We present a collection of tools that provide a static information flow analysis across a set of applications, showing a holistic view of all the applications destined for a particular device. The techniques we present include a static binary single-app analysis, a security lint tool to mitigate the limits of static binary analysis, a multi-app information flow analysis, and an evaluation engine to detect information flows that violate specified security policies.

We show that our single-app analysis is comparable with the leading approaches on the DroidBench benchmark suite; we present a brief listing of lint-like heuristics used to show the limits of the single-app analysis in the context of an application; we present a multi-app analysis, and demonstrate information flows that cannot be detected by single-app analyses; and we present a policy evaluation engine to automatically detect violations in collections of Android apps.

General Terms

Security

Keywords

Security, Android, Java, Static Analysis

1. INTRODUCTION

The Android operating system powers a wide variety of mobile devices and provides a comprehensive set of services supporting a plethora of applications. Android's capabilities have enabled a large number of software authors to contribute to the application ecosystem, often with little professional training. However, there is little assurance that these applications behave as advertised.

[*]**Distribution Statement A: Approved for Public Release; Distribution is Unlimited.**

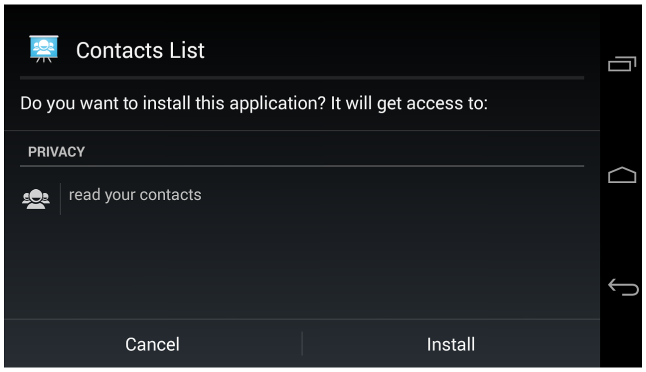

Contacts List
Do you want to install this application? It will get access to:
PRIVACY
read your contacts
Cancel
Install

Figure 1: Users must agree to grant this app access to their contacts list in order to install the app; however, this screen does not depict the full set of capabilities due to potential collusion.

Even well-intentioned application authors may not fully understand the complete behavior of their own applications–due to the inherent complexity of software development, or as is often the case, due to their dependency on third party libraries. Many such dependencies are only available in binary form, such as libraries that enable ad-supported revenue[1].

Android does enforce a security model based on *permissions* that is intended to prevent applications from accessing capabilities that the user has not approved. Each application must statically declare the set of permissions required, and the runtime system only allows the application access to behaviors protected by those declared permissions. The runtime throws exceptions if the application attempts to access any other permissions, preventing access.

Android presents a summary of the requested permissions to the user through the user interface depicted in Figure 1.

[1]Many free applications use advertising as a primary revenue stream, but the libraries to distribute advertisements frequently receive executable code that describes the way the content is displayed, and how the user can interact with that ad.

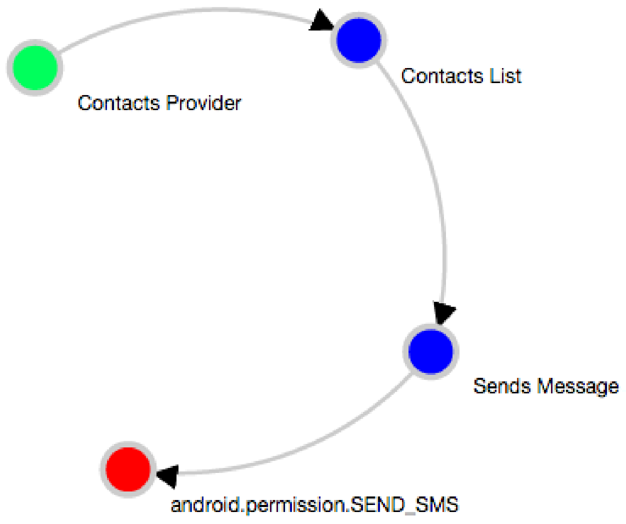


Figure 2: Two example applications colluding; Sends Message provides an API that effectively elevates the permissions of Contacts List, allowing access to the SEND_SMS permission.

Users who do not accept the requirements cannot install the application.

Even assuming that users understand this information before approving installation, the permissions that an application requests are not sufficient to characterize the information and resources that an application can actually access or manipulate. For example, an application could retrieve contact details directly by requesting just one permission as shown on line 6 of listing 1. That application can then send those details to another application through standard inter-application communication channels, as on line 17. Another application, such as the one shown in listing 2, could then leak the information (via SMS–line 15) that it would otherwise not have been able to access. Figure 2 depicts this information flow, using the multi-app visualization that is generated by our tools. Circles indicate applications, content providers, or permissions, and arcs indicate potential information flow.

The goal of our work is to help identify undesirable or suspicious communication patterns within curated collections of applications, also called appkits. In our model, the contents of an appkit are chosen by some curator who will use an analysis like ours to ensure that the appkit contains no undesirable data flows. We have scripts to pull all of the applications off of a device and upload them as an appkit; this makes appkit assembly nearly as easy as installing applications on a phone. Each appkit can be thought of as a communication graph with applications as nodes and information flows between applications as edges. Our analysis tools infer the communication graph and present it to analysts graphically. We have also developed a policy language and evaluation engine with which analysts can express information flow policies over the communication graph. The primary class of vulnerabilities or attacks we are concerned with are *multiple application collusion* information disclosures. In this type of information disclosure, an application with no or limited permissions can obtain sensitive information and then leak it to the outside

```
1  public class ReadsContactApp extends Activity {
2    public void onCreate(Bundle b) {
3      super.onCreate(b);
4      Cursor cur =
5        managedQuery(People.CONTENT_URI, ...);
6      String nameNumber = extractQueryData(cur);
7      sendData(encode(nameNumber));
8    }
9
10   private void sendData(String d) {
11     Intent i = new Intent();
12     i.setAction(Intent.ACTION_MAIN);
13     i.addCategory(Intent.CATEGORY_LAUNCHER);
14     i.setClassName("SendsMessageApp",
15        "SendsMessageApp");
16     i.putExtra("Example.CovertDataChannel", d);
17     startActivity(i);
18   }
19
20   private String extractQueryData(Cursor cur){
21     String name = "empty contacts";
22     String phoneNumber = "empty contacts";
23     if (cur.moveToFirst()) {
24       int nameCol =
25         cur.getColumnIndex(People.NAME);
26       int phoneCol =
27         cur.getColumnIndex(People.NUMBER);
28       name = cur.getString(nameCol);
29       phoneNumber = cur.getString(phoneCol);
30     }
31     return (name + ":" + phoneNumber);
32   }
33 }
```

Listing 1: The code for Contacts List in Figure 2

```
1  public class SendsMessageApp extends Activity {
2    private static final String CHANNEL =
3      "Example.CovertDataChannel";
4
5    public void onCreate(Bundle b) {
6      super.onCreate(b);
7      String d = extractExampleCovertData();
8      if (d == null) return;
9      sendSMS("xxx-xxxx", d);
10     finish();
11   }
12
13   private void sendSMS(String num, String msg) {
14     SmsManager sms = SmsManager.getDefault();
15     sms.sendTextMessage(num, null, msg, null, null);

16   }
17
18   private String extractExampleCovertData() {
19     return getIntent().getStringExtra(CHANNEL);
20   }
21 }
```

Listing 2: The code for Sends Message in Figure 2

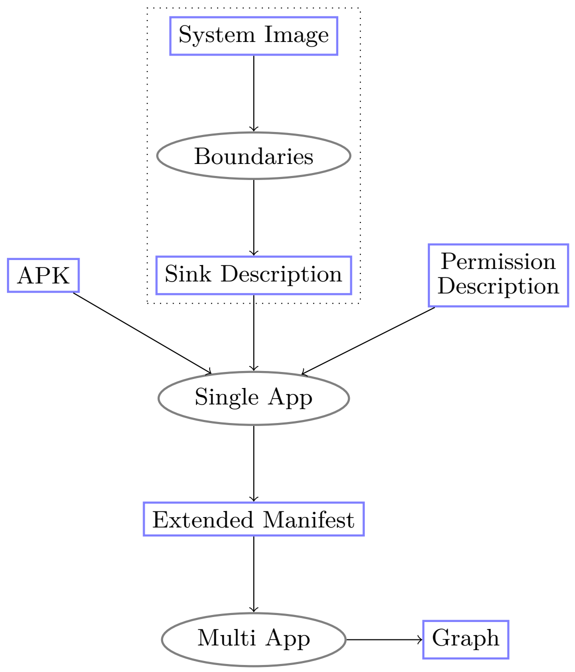


Figure 3: Architecture diagram for FUSE

world simply by communicating with other applications that have more privileges and are benign in isolation.

Since application source is not always available, and in any case cannot be trusted to create the binary running on devices, we analyze binary Android applications at the bytecode level using an IR in SSA form [6]. At a high level, our analysis is divided into two parts. First, we analyze each application in isolation (Section 3) to recover the call graph of the application and find the reachable *sources* and *sinks* of each component. We consider sources to be any source of sensitive information, including IPC details used to start the component (the *Intent*–described in Section 2) and information obtained from permission-protected APIs. Sinks are places that sensitive information can flow to, and typically take the form of inter-process communication performed by a component (instances where the subject application sends information to another application or external source such as network destinations via SMS, IP, or the local file system, for example). After we have identified all of the sources and sinks in a collection of applications, we construct a graph representing the communication patterns of the collection (Section 4). Analysts can then interactively explore the graph to identify problematic information flows between applications. The full architecture of our system is depicted in Figure 3.

The Android ecosystem has developed a number of inherent complexities in the pursuit of simplified software development; unfortunately, some of these features and common practices pose significant difficulties for static analysis. To combat that, we have also implemented a separate lint-like tool for Android applications (Section 6) to notify analysts of points where an application uses techniques that are outside of the scope of our single-app analysis. For example, native code and runtime systems—such as JavaScript evaluation—operate outside of our single-app information flow analysis, but these features can often be *detected* by the linter, helping to identify the limits of our static analysis in the context of the specific application under analysis. Despite these mitigations, our approach shares many of the same limits to soundness and completeness that plague static binary analysis tools in general. We therefore cannot claim that the results generated are either sound or complete, but the combination of tools can provide a reasonably full picture of an application's behavior (and thus, the behavior of an entire appkit). Section 7 describes an evaluation of our single-app analysis to provide a more quantitative view of its capabilities.

1.1 Attack Model and Assumptions

We assume that an adversary can provide one or more applications containing arbitrary valid Dalvik bytecode. Furthermore, we assume that the dynamic permission checks in the Android system are correct and that applications cannot circumvent them through exploits to access permission-protected resources not specified in their manifests. Our approach also assumes that no information flows are enabled by native code.

2. STRUCTURE OF ANDROID APPLICATIONS

Android applications are distributed as binaries in a regular format based on zip files. While Android applications are written in Java, they run on a custom virtual machine called the Dalvik VM, which has its own register based bytecode format. Each application can contain:

- a manifest describing required permissions and available components (`AndroidManifest.xml`);
- a Dalvik executable (`classes.dex`), which contains the bytecode of the program;
- (optional) resources including string literals, their translations, and references to binary resources; and
- (optional) XML layouts describing user interface elements.

Each application is made up of a set of communicating *components*; each component has multiple entry points that are invoked by the Android system as callbacks in response to system events. Moreover, components can run concurrently with each other. Each application runs in its own OS level process, offering a degree of isolation between applications. While components of the same application give the impression of executing separately, they actually share a single address space by default. This allows components to communicate through both standard inter-process communication (IPC) mechanisms and through global variables. Furthermore, applications from the same developer (which is indicated by being signed with the same cryptographic key) can actually share a single address space, with one injecting code into the other. Application components are one of:

- Activities, which represent screens visible to the user;
- Services, which can perform background computation;
- Content Providers, which act as database-like data stores; and
- Broadcast Receivers, which handle notifications sent to multiple targets.

Being an event driven system, events from the user interface result in callbacks from the Android system into application code. These events come from native code and, for the most part, do not originate in Java code. While these callbacks are delivered to a standard set of interfaces by default, users can specify additional event handlers in XML layouts declaratively. Layouts are accessible at run time through `View` objects, which represent trees of user interface widgets.

Components can communicate with each other, both within the same process and between processes, through two Android specific interfaces: 1) an inter-process communication system embodied by the `Intent` and 2) content providers, which act as data stores. An `Intent` is a Java class that contains a small set of semi-structured data that determines the intended receiver. Additionally, `Intent`s can carry a data payload containing any serializable values. Each intent can be either *explicit* or *implicit*. Explicit intents specify an exact package and component as their recipient. Implicit intents specify their recipient by a general action they expect the receiver to be able to perform on their behalf; the Android system routes implicit intents to the first component that claims to be able to handle them. Each application declares the set of intents that each of its components can handle in its manifest by defining intent *filters*—simple predicates over the data contained by intents.

Android protects sensitive information and services through *permissions*. Most sensitive information sources, such as system content providers and GPS hardware, have an associated permission that an application must be granted in order to access it, as mentioned in Section 1. The required permissions are declared in the Android manifest, and are used to populate the approval dialog shown in Figure 1. Applications can also define their own permissions beyond those provided by the Android system and individual components in an application can restrict their APIs, only responding to apps that have been granted specific permissions.

3. SINGLE-APP ANALYSIS

We first individually analyze each application in an appkit to produce an *extended manifest*—a data structure that builds on the standard Android manifest included with each app. The extended manifest records the *sources* and *sinks* in an application, as well as the information flows between them. In essence, the extended manifest represents the internal information flow graph from application inputs (sources) to application outputs (sinks). Sources are:

- the inputs to each component, or
- permission protected resources.

Sinks represent the means by which a component can send (possibly sensitive) information to another component or to the outside world, including:

- sending broadcasts,
- starting activities,
- communicating with a service (start, stop, or bind),
- reading from, or writing to, a content provider,
- reading from, or writing to, internal (non-private) storage,
- sending data to a permission protected method, or
- writing data to the system log.

Each sink records its type, from the list above, and the list of sensitive information sources that can flow to it.

```java
class MyTitle implements CharSequence {
  private Context context;
  MyTitle(Context ctx) {
    context = ctx;
  }

  public String toString() {
    context.startActivity(...);
  }
}

class MyActivity extends Activity {
  void onCreate(Bundle b) {
    CharSequence title = new MyTitle(this);
    DownloadManager m =
      new DownloadManager(null, "");
    Request r = new Request(...);
    r.setTitle(title);
    m.enqueue(r);
  }
}
```

Listing 3: A code sample requiring framework analysis

```java
class DownloadManager {
  class Request {
    private CharSequence mTitle;
    public Request setTitle(CharSequence title) {
      mTitle = title;
    }

    ContentValues toContentValues(String pkg) {
      ContentValues values = new ContentValues();
      values.put(Downloads.Impl.COLUMN_TITLE,
          mTitle.toString());
    }
  }

  public long enqueue(Request request) {
    ContentValues values =
      request.toContentValues(...);
  }
}
```

Listing 4: Relevant framework code referenced by listing 3

The core of our single-app analysis is a pointer analysis with which we compute a call graph and determine the reachable methods in each application. We use a version of Andersen's analysis [2], which is field-sensitive but flow- and context-insensitive. Our implementation provides on-the-fly call graph construction and uses type filters [8], which keep points-to set sizes manageable. We provide summaries of the core Java collection classes to improve scalability and precision: the summaries enable us to treat collections context-sensitively. Aside from these summaries, we analyze each application with the complete Android framework and core Java libraries. We then employ a taint analysis to determine where sensitive information in each application can flow. We introduce taint labels at each *source* in the application. Any taint labels that reach *sinks* reveal the information flows that the application enables. We record these *flows* as a list of sources connected to each sink in the extended manifest.

We analyze application code along with the full framework because some information flows can be hidden through framework code, as demonstrated in listing 3. In this example, `MyActivity` is an Android `Activity` implementing the standard `Activity` entry point `onCreate`. The most important feature of this code is that the class `MyTitle` has an implementation of `toString` on line 7 that starts an `Activity`. Furthermore, the `Activity` never explicitly calls this `toString` method. An instance of `MyTitle` is passed to the Android framework on line 18; even at this point, there are no adverse effects. However, by consulting the relevant framework code in listing 4, it is apparent that the subsequent call to `enqueue` indirectly calls the duplicitous `toString` method, thus launching an `Activity`. Analyzing the application code without the framework code that it depends on is insufficient to determine what methods defined in the application are really called, and summaries are not sufficient to construct the complete call graph. Treating all application code as reachable is a safe approximation, but it sacrifices precision when applications include large libraries while only using a small portion of the included functionality. Furthermore, naively including all of the code in an application might uncover all IPC calls, but again sacrifices precision in determining what information can flow to the affected sinks. Conservatively assuming that all information can flow to a sink is normally too pessimistic to report to users because such an approach generates an overwhelming number of false positives.

3.1 Android Lifecycle Model

Unlike most applications, Android apps do not begin executing from a single `main` method. Instead, Android applications are composed of multiple components, each of which has multiple entry points (called the Android Lifecycle Methods). Some examples are `Activity.onCreate` and `Service.onBind`. User interface event handlers, including those specified in layouts, are additional entry points called by the Android system at unpredictable times.

We model an application by creating a synthetic `main` method that, for each active component `C` from the manifest:

1. allocates an object `o` of type `C`,
2. calls the default constructor for `C` on `o`,
3. sets the `Context` of `o` to a unique instance of `ContextImpl`, and
4. calls each of the lifecycle methods of `o`.

We manually set the `Context` of component objects because it is not populated by the object constructor. Instead, the Android framework explicitly assigns one shortly after reflectively creating the object. Because our analysis is flow insensitive, we do not need to impose any specific ordering on the calls to the lifecycle methods, nor do we need to model the interleaved execution of components. We similarly instantiate layouts defined in application resources and set up calls from the synthetic `main` method to: 1) `setContentView` for the components that instantiate the layout, 2) the standard user interface callback methods, and 3) any extra callbacks specified in application layouts. Our pointer analysis begins exploring the call graph from the synthetic `main` method.

3.2 Permissions

We use the summaries produced by the PScout tool of Au et al. [4] to determine what permissions are used by an application. Permission-protected methods can be either information sources or information sinks; however, PScout does not differentiate. We make the following assumptions about the PScout summaries:

- permission protected methods that return a reference type are sources, and
- other permission protected methods serve as sinks for their arguments.

Furthermore, we treat a number of interfaces related to permission protected resources specially to capture their semantics as sources of sensitive information. For example, we have a summary of the `onLocationChanged` method of the `android.location.LocationListener` interface that marks location information as sensitive for any type implementing the interface. While we rely on the PScout summaries, FUSE can be easily adapted to any suitable descriptions of Android system permissions.

3.3 Taint Analysis

We introduce a taint label for each source in an application in order to track information flows through an application from sources to sinks. Permission protected source methods are annotated with the permission required to invoke that method. For example, calls to the `getDeviceId` method of the `TelephonyManager` class are labeled with the `android.permission.READ_PHONE_STATE` permission. Parameters to methods of permission protected interfaces (e.g., `android.location.LocationChanged`) receive taint labels in a similar fashion. We also label each `Intent` that starts a component with the class name of the component it started. Inputs to components must be tainted so that we can track their flow to sinks. If a component passes its inputs on to other components through IPC, it can enable subtle flows from unprivileged applications to more privileged applications—possibly allowing the unprivileged application to control the execution of code in a privileged application. We formulate our taint analysis as a system of set constraints [1] with taint labels as values in sets and program points and heap locations as set variables. We use the results of our pointer analysis to propagate taint labels through heap locations.

3.4 Finding IPC Sinks

With a call graph in hand, our single-app analysis next finds IPC sinks, which are method calls that send information to other components (possibly other applications) in the system. To find sinks, we rely on sink descriptions that we extract from the Android platform. Each new version of the Android platform introduces new APIs for IPC, requiring limited updates. Our tools are designed to track Android development while attempting to minimize the manual annotation burden for each new Android release.

We manually annotated the basic set of IPC methods in the `IActivityManager` and `IContentProvider`[2] interfaces, which are implemented by or used by all other current IPC mechanisms. These annotated interfaces are generally not used directly by Android applications; rather, applications invoke methods that eventually delegate to these interfaces. We have a separate tool that propagates these IPC annotations up the call graph of the Android framework code to create a list of sink methods that can be used for IPC. This process minimizes the manual annotation effort while still covering the wide API that applications use in practice.

[2]We also include `ContentProvider` and `ContentResolver`, which are also base classes for content provider IPC.

When a new version of the Android platform is released, we can re-run this discovery algorithm without having to manually annotate all of the new methods introduced in the release. The manual annotations on the `IActivityManager` and `IContentProvider` interfaces need to be updated if they change from version to version, but they change much less frequently than the rest of the platform.

We find sinks by iterating over all of the reachable call instructions c in an application. If c is a call to a sink method, we examine its arguments. Each type of sink can expose one or more of its arguments to the outside world through IPC, file access, or permission protected resources. For IPC sinks, that parameter is an `Intent`. For each `Intent` we record:

- details related to the destination of the IPC[3].
- the taint labels reaching the `Intent` being sent, and
- the name of the calling method.

We record each of these triples in the extended manifest. This is enough to tell us where each application can send sensitive information. If no taint labels reach a sink, we know that the sink cannot expose sensitive information through IPC.

3.4.1 *String Analysis*

The destinations of IPC calls are specified through strings. While these are often string literals, they are sometimes constructed dynamically at run time. We make an effort to statically approximate the relevant strings with a string analysis. Our string analysis is based on def-use chains, and is thus intraprocedural. However, it is augmented with information from the points-to analysis to resolve string values loaded from the heap.

In the cases where we cannot statically approximate a string determining the destination of an IPC call, we conservatively assume it can be sent to any component in the system.

3.4.2 *Parametric Intents*

We introduce the notion of *parametric* communication to account for inter-process messages that may themselves come from outside the application. If the `Intent` destination is an input to the component, we cannot know what the possible targets of the `Intent` are. Instead, we record that the `Intent` is *parametric* and defer resolution until we construct a graph of the communication between components in a collection of applications (as described in Section 4).

3.5 Reflection

Our analysis resolves a subset of object types instantiated through the reflection API, specifically: `Class.newInstance`. This is often used to choose between alternative implementations of interfaces based on the installed Android platform version. The vast majority of calls to `Class.newInstance` are immediately followed by a checked cast to a specific type. The casts effectively limit the type of the reflectively-created object, as long as the reference is not used on any other paths (that do not have checked casts). We use the types in the casts as an approximation (unless that type is `Object`, which would not limit anything). We currently ignore reflectively-called methods.

[3]For reasons explained in Section 3.4.2, we cannot always identify IPC destinations at this step.

A more sophisticated reflection analysis, such as the work by Livshits et al. [9], could improve our results in some cases. While a more precise analysis would reduce the number of cases that cannot be handled, there could always be uses of reflection that stymie any analysis. Ideally, once the analysis is precise enough, any uses of reflection that cannot be resolved could be flagged as suspicious.

4. MULTI-APPLICATION ANALYSIS

The multi-application analysis takes as input an appkit—a collection of apps, including the extended manifest for each app in that collection. The multi-app analysis produces a graph characterizing the flow of information between applications in the appkit k. We construct the graph $G(V, E)$ with Vertexes V and Edges E in the following way (the functions `match` and `enrich` are defined in Section 4.1 and Section 4.2 below):

1. Add a vertex V for each permission p used in k.
2. For each application $a \in k$, add a vertex to V for each of $\mathit{sources}(a)$, and $\mathit{sinks}(a)$.
3. For each sink $s \in V$, for each source or permission $c \in \mathit{flows}(s)$, add an edge $c \rightarrow s$ to E.
4. For each sink $s \in V$ and each source $c \in V$, add an edge $s \rightarrow c$ to E if $\mathit{match}(s, c)$.
5. Until a fixed point is reached:
   (a) For each parametric sink $p \in V$, $\mathit{enrich}(p)$.
   (b) For each sink $s \in V$ and source $c \in V$, add edge $s \rightarrow c$ to E if $\mathit{match}(s, c)$.

After adding the permissions from the appkit, we add individual applications to the graph and model the information flow within each application. In step 4, we seed the graph with the initial set of inter-application edges representing simple inter-process communication. The set $\mathit{flows}(s)$ contains all of the sources (components and permissions) that can flow to s. Recall from Section 3.4 that the destination of a parametric IPC message depends on the inputs to a component and cannot be determined in isolation. In step 5, we resolve parametric information flows by computing a fixed point over the parametric sinks in G.

4.1 Matching Sinks to Components

The predicate $\mathit{match}(s, c)$ is true if sink s can send IPC messages to component c. We implement the same matching rules as the Android run time system[4], with one exception: IPC targets that our string analysis cannot resolve match all components, as discussed in Section 3.4.1. If s sends an implicit intent to another component, Android would route the message to the first component that could handle it, or possibly prompt the user; we conservatively route implicit intents to all matching components.

4.2 Enriching Parametric Sinks

A parametric sink, like any other sink, sends a message to another component. Unlike most sinks, however, the target of a parametric sink is determined by an input to the component hosting the sink, c. To find the target components of a parametric sink p, we must examine all sinks s' that send `Intent`s that can flow to p. $\mathit{enrich}(p)$ modifies p by merging the intents of s' into the intents sent from p. For

[4]Android Intent Resolution: `http://goo.gl/f7ISt9`

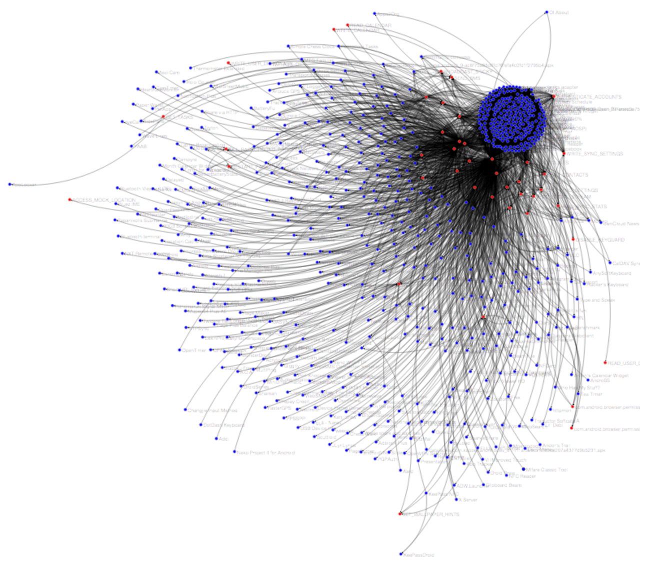


Figure 4: The multi-app graph of application interactions in F-Droid, scaled to fit on the page.

example, if p sends an explicit `Intent`, the target class names of the `Intents` sent by s' will be added to the `Intents` sent by p. The merged set of components (or other fields) are disjoined—indicating that the parametric sink could send an `Intent` to either of the named components.

5. MULTI-APP POLICY EVALUATION

FUSE generates multi-app graphs that are often overwhelming in their complexity, such as the graph of all applications in the F-Droid open-source application repository depicted in Figure 4.

The complexity of interactions between apps can make it difficult to manually locate all information flows of interest. To simplify this process, we have introduced a system of information flow assertions that allow analysts to specify information flows that are either disallowed, expected, or only allowed if a specific application is involved at an intermediate stage. For example, a security analyst may wish to ensure that no information can flow from the contact list to the SMS system. Figure 5 (a) shows the assertion that encodes this policy.

The user is alerted if an appkit violates this policy, and an example path is provided to demonstrate how the violation could occur. This path includes the full list of applications or permissions involved in the information flow from the starting node to the end.

Alternatively, such a flow may be allowable if the sensitive information is scrubbed or encrypted before it is transfered off of a device. For example, the applications on a curated device could be structured such that all images are sent through an application that specifically strips geotags from the image metadata before passing it on for further processing. The policy depicted in Figure 5 (b) ensures that all information obtained from methods protected by the `FINE_LOCATION` permission (which protects access to the GPS) must be processed by an app (`GeoCleaner`) before the information can be sent to any methods protected by the `INTERNET` permission.

6. SECURITY LINTER

The security linter complements our single-app analysis by 1) identifying potentially dangerous uses of the Android framework and 2) highlighting code that the analysis does not target. The linter can be run quickly at reduced precision, or in a more precise configuration. The more precise configuration is essentially free when run with the single-app analysis, as they share core data structures. The security linter issues warnings for problems like:

- Component hijacking (as described by Lu et al. [10]),
- Dynamically registered broadcast receivers,
- Known use of insecure credential storage,
- Reflection that FUSE cannot resolve,
- Unused permissions requested in the manifest, and
- Writing to world-accessible files (which allows for trivial information disclosure)

The individual checks implemented in the security linter make heavy use of the SSA IR, points-to graph and callgraph used in the single-app static analysis. We are able to very quickly add additional security checks to the linter by reusing these structures that are already necessary to identify information flows through applications. This sharing reduces implementation effort as well as analysis runtime.

7. EVALUATION

While application collusion is a growing concern, there are few tools to actually detect such behaviors and no multi-app evaluation suites (that we are aware of) available for comparative analysis. Thus, the following sections focus on the evaluation of the FUSE single-app analysis, and show comparisons with Fortify and FlowDroid—two of the leading single-app static analysis tools. This analysis is inherently incomplete, as it does not include any of the capabilities of the FUSE multi-app, multi-app policy, or linter tools. However, it does demonstrate that the single-app analysis is comparable with the leading approaches, and as such, is a suitable base on which to build the higher-level tools that cannot be compared to existing approaches.

7.1 DroidBench

We have run our tools on the DroidBench test suite, an Android-specific single-app test suite developed by Arzt et al. [3] to evaluate their FlowDroid tool. Our results are shown in Table 1. Compared to FlowDroid:

- Our precision is lower due to our lack of flow sensitivity.
- Our recall is equivalent with one exception: we do not treat password fields as sensitive information; With improved summaries (perhaps even using the FlowDroid summaries from SUSI [13]), the FUSE recall would be higher than FlowDroid's current performance[5].

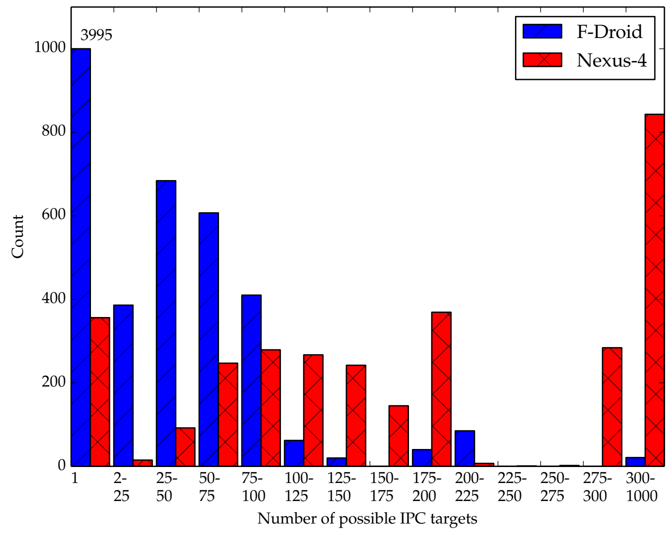
We report false positives on *FieldSensitivity4* and *ObjectSensitivity2* because FUSE is not flow sensitive. Like FlowDroid, FUSE is unable to find the leak in *IntentSink1* because we do not model the flow of information from `Activity.setResult` and `Activity.startActivityForResult`.

Our analysis reports a false negative on *ActivityLifecycle1* due to the nature of our summaries, which are based on the

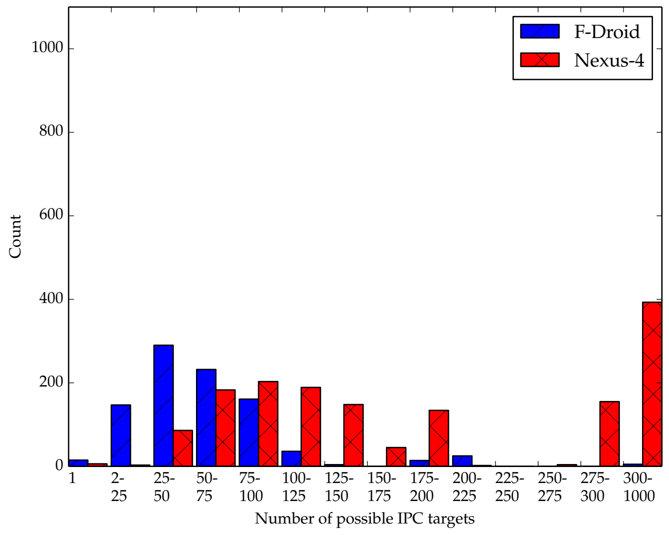
[5]Improved summaries would allow FUSE to pass both password field tests and *ActivityLifecycle1*, raising the FUSE recall to 96%.

```
(a) assert [com.android.contacts] -> [android.permission.SEND_SMS] :: never;
(b) assert [android.permission.FINE_LOCATION] -> [android.permission.INTERNET] :: *. com.example.GeoCleaner *.;
```

Figure 5: Information flow assertions in FUSE



(a) Explicit Intents



(b) Implicit intents

Figure 6: Numbers of inferred targets for intents

results of PScout. PScout only records the list of Android framework methods that are guarded by permissions. We treat PScout methods that return a reference type as *sources* of permission protected information. We consider the rest of the PScout methods to be *sinks* for sensitive information. To cut down on false positives, we assume that receiver objects do not flow to sinks, while the remaining arguments do. This assumption causes us to miss the leak in *ActivityLifecycle1*. With more precise source and sink summaries, our analysis could resolve this case.

The private data leaks (*PrivateDataLeak1* and *PrivateDataLeak2* test whether the analysis treats strings obtained from password fields in Android as sensitive data. FUSE does not treat password fields as sensitive data sources and trivially fails these tests. While it is straightforward to identify password fields accessed by their unique identifier assigned in static layout files, a number of Android `Views` can be configured dynamically to become password fields. It is in general difficult to say whether or not an arbitrary Android `View` is a password field because this information is not necessarily available statically. In the absence of robust static inference of password fields, we are faced with either reporting leaks of obvious password fields or large numbers of false positives. Without knowing precisely which `View` objects are password fields, reading a value from any text field in a layout containing a password field would taint the values in all fields of the layout.

7.2 Sinks Found

In this section, we report on the ability of our single-app analysis to resolve targets of interprocess communication. In Table 2, We report for three collections of applications:

- Nexus 4: a collection of 189 applications drawn from an end-user's Nexus 4 (including system applications), running Android 4.4.2, intended to be representative of a typical user device.
- F-Droid: The complete collection of 1124 unique apps in F-Droid as of May, 2014.
- Malware: The Malware Genome dataset [16], a collection of 1260 apks containing assorted Android malware.

For each of these application collections, we calculated: 1) the total number of sinks, 2) the number of parametric sinks, 3) the number of intents sent from sinks, and 4) the number of those intents whose targets we are able to resolve. The number of intents sent from sinks is less than the total number of sinks because not all sinks (e.g., content provider sinks and permission sinks) have an associated intent. Though our analysis is flow insensitive, our string analysis is able to resolve at least a prefix of most of the relevant fields of intents. Furthermore, a small but significant number of intents are parametric and must be resolved by the fixed point iteration in our multi-application analysis.

In Figures 6a and 6b, we look more closely at the precision of our analysis on intent-based sinks. Figure 6a is a histogram in which each entry on the X axis is a bucket representing the number of targets that our multi-application analysis identifies for an explicit intent. Since explicit intents name their expected handler exactly, we would expect most explicit intents to have exactly one target. In the F-Droid appkit, the vast majority of explicit intents have exactly one target, as expected, and most of the rest have a modest number of possible targets. This implies that our analysis is precise enough to handle the applications in the F-Droid appkit.

The results from the Nexus 4 appkit are more mixed. While a large number of these explicit intents have a single

⊛= correct warning, *= false warning, ○= missed leak

App Name	Fortify	FlowDroid	FUSE
Arrays and Lists			
ArrayAccess1		*	*
ArrayAccess2	*	*	*
ListAccess1	*	*	*
Callbacks			
AnonymousClass1	⊛	⊛	⊛
Button1	⊛	⊛	⊛
Button2	⊛○○	⊛⊛⊛*	⊛⊛⊛*
LocationLeak1	○○	⊛⊛	⊛⊛
LocationLeak2	○○	⊛⊛	⊛⊛
MethodOverride1	⊛	⊛	⊛
Field and Object Sensitivity			
FieldSensitivity1			
FieldSensitivity2			
FieldSensitivity3	⊛	⊛	⊛
FieldSensitivity4			*
InheritedObjects1	⊛	⊛	⊛
ObjectSensitivity1			
ObjectSensitivity2			*
Inter-App Communication			
IntentSink1	⊛	○	○
IntentSink2	⊛	⊛	⊛
ActivityCommunication	⊛	⊛	⊛
Lifecycle			
BroadcastReceiverLifecycle1	⊛	⊛	⊛
ActivityLifecycle1	⊛	⊛	○
ActivityLifecycle2	⊛	⊛	⊛
ActivityLifecycle3	○	⊛	⊛
ActivityLifecycle4	⊛	⊛	⊛
ServiceLifecycle1	○	⊛	⊛
General Java			
Loop1	○	⊛	⊛
Loop2	○	⊛	⊛
SourceCodeSpecific1	⊛	⊛	⊛
StaticInitialization1	⊛	○	⊛
UnreachableCode	*		
Miscellaneous Android-Specific			
PrivateDataLeak1	○	⊛	○
PrivateDataLeak2	⊛	⊛	○
DirectLeak1	⊛	⊛	⊛
InactiveActivity	*		
LogNoLeak			
Sum, Precision, and Recall			
⊛, higher is better	17	26	24
*, lower is better	4	4	6
○, lower is better	11	2	4
Precision $p = ⊛/(⊛ + *)$	81%	86%	80%
Recall $r = ⊛/(⊛ + ○)$	61%	93%	86%
F-measure $2pr/(p + r)$	0.70	0.89	0.82

Table 1: FUSE DroidBench results. Empty rows indicate benign tests (no flows exist to be found).

AppKit	Nexus 4	F-Droid	Malware
Number of Apps	186	2126	1261
Total Sinks	19178	48596	52342
Parametric Sinks	260	346	363
Intent Targets	6338	22573	20238
Resolved Targets	6318	22564	19348
Analysis Time	8.7 hours	6.8 hours	

Table 2: Sinks found in a selection of applications

target, many more have multiple targets, including many with more than 100 possible targets. There are a number of reasons that our analysis might find more than one target for an explicit intent:

- more than one intent object actually reaches the sink, and
- the component used to construct the intent could have been specified with a string assembled at run time that our string analysis could not uniquely resolve.

The first case is inherent to any analysis, though a fully context sensitive analysis could be more precise. The second case could also be improved with a more sophisticated string analysis, but some run time constructed strings will always introduce imprecision. In the case of the Nexus 4 appkit, we suspect that the applications are inherently more complex and generic, as they provide basic system services. Both complexity and generality reduce the precision of our static analysis. As a proxy for complexity, we can look at application size: proportionally more of the applications in the Nexus 4 appkit are very large compared to the F-Droid appkit.

We see a similar trend for implicit intents in Figure 6b. As expected, implicit intents rarely have a unique target. We always assume that implicit intents can be received by all possible recipients. Though implicit intents are only routed to a single receiver at run time, we cannot know that recipient statically. Different users of the same image are even likely to choose different recipients, so we cannot make any static assumptions. Again, our analysis is more precise on the smaller and simpler applications in the F-Droid appkit. We believe that our biggest source of imprecision is in our string analysis, though we have not yet had the opportunity to thoroughly analyze the results.

7.3 Performance

Run times for the single application analysis discussed in Section 3 are shown in Figure 7. The Y axis indicates the number of bytecode instructions in each application; this number does not include instructions in the Android framework that the application references. As expected, larger applications take longer to analyze. Our analysis completes in under 30 minutes for most of the applications we analyze, including the vast majority of the applications from F-Droid. Proportionally more of the applications in the Nexus 4 appkit take longer to analyze. These applications are, on average, both larger and more complicated than most of the applications in F-Droid. The single application analysis spends the vast majority of its time on pointer analysis.

Table 2 shows the run times for our multi-application analysis. Note that we did not analyze the malware collection

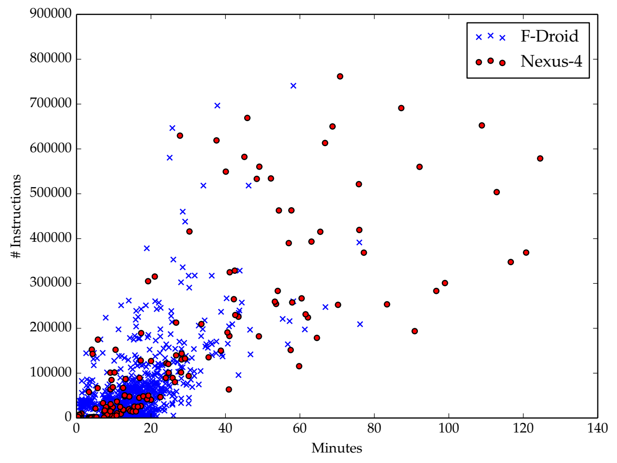


Figure 7: Application analysis times

as an appkit, since it would not represent a very realistic collection of applications to have installed on a single device. Despite the relatively small number of parametric intents, our analysis still takes a few hours to reach a fixed point on realistic appkits. The F-Droid appkit does not include all of core Android applications that are present in the Nexus 4 appkit. These core Android applications are again more complicated and provide larger public-facing interfaces than the average Android application, which makes the multi-application analysis more expensive.

8. RELATED WORK

With the ubiquitous deployment of Android, a number of projects have investigated security analysis for the platform. The work closest to ours is Epicc [12], which also considers the multi-application collusion problem. Epicc reduces the problem to an interprocedural dataflow environment problem (IDE) [15]. Compared to our work, Epicc is flow and context sensitive, which should eliminate some false positives. However, Epicc does not address communication through content providers, track sensitive information obtained through permission protected framework methods, or fully analyze the Android framework code. Furthermore, we introduce the notion of parametric sinks whose targets must be resolved through a fixed point computation. As we discuss in Section 7.2, parametric sinks are not uncommon in practice.

DroidForce [14] addresses the multi-application collusion problem with a dynamic enforcement mechanism backed by a flexible policy language. Under this dynamic approach, all applications on a device are instrumented with dynamic checks. If an application attempts an operation that violates the policy, the operation is denied (i.e., skipped at the bytecode level). While this can introduce crashes, some applications can continue despite the denied operation. Like Epicc, our approach is completely static; static approaches are most helpful to analysts attempting to understand applications and developers attempting to secure their own applications. DroidForce complements static approaches, providing additional assurance to users, who might have their own policies; furthermore, the dynamic approach can compensate for the inability of static analysis tools to reason about dynamically loaded code.

DroidForce builds on FlowDroid from Arzt et al. [3], which discovers information flows within individual applications. FlowDroid casts the problem as an IFDS problem with support from an on-demand pointer analysis to handle heap references. Their analysis is field, flow, and context sensitive with some object sensitivity and is thus very precise. However, FlowDroid is unsound in the presence of threads, while FUSE is sound with regard to thread interleavings since it is not flow sensitive.

The CHEX tool [10] is designed to find component hijacking attacks, which involve applications using unsanitized inputs from other applications in sensitive method calls. Applications that expose privileged components with insufficient permission protection can inadvertently allow malicious applications to use their services for nefarious purposes. CHEX uses a flow and context-sensitive dataflow analysis to model the flow of information through Android applications. CHEX models Android framework code, but cannot discover the full call graph for applications like that in listing 3. Our single-app analysis infers the same types of information flows as CHEX; thus, our security linter can recognize possible component hijacking attacks. Additionally, we identify parametric sinks that CHEX does not address.

The SCanDroid tool [5, 7] has similar goals and direction to our work. It is implemented on top of the IBM WALA toolkit and implements a flow analysis in terms of a simple constraint system. Their constraint formulation is analogous to our taint analysis, which supports multiple taint labels. Both SCanDroid and FUSE work on arbitrary Android applications without source code; however, we have been unable to apply SCanDroid to analyze the full framework due to scalability issues, which prevents SCanDroid from reaching the precision and recall possible with FUSE. SCanDroid aims to incrementally evaluate flows as new applications are installed on devices; our work analyzes entire appkits at once. In our model of curated collections of applications, incremental analysis is less important. That said, our multi-application analysis usually runs quickly enough that adding a new application and re-running the analysis is not prohibitive.

Myers [11] extends the Java language with explicit notions of information control in JFlow. In this model, developers manually label values and provide information flow policies that are enforced with a mix of static and dynamic checks. The JFlow model is more expressive than ours, incorporating a notion of implicit information flow (i.e., information leaks due to control flow rather than data flow). However, it requires manual labeling, while our work does not. Furthermore, we do not enforce flow policies. Instead, we only discover information flows in existing applications. The notion of authorities and declassification of information flows would complement our work by allowing data to be untainted, which would add flexibility to our taint analysis.

9. CONCLUSION

The danger of Android application collusion is often overlooked, as single-app exploitation is still a profitable means for adversarial activity; however, we believe that the perceived difficulty of using multiple applications is much higher than the actual cost. While we are unaware of any widespread instances of malicious application collusion at this point, it is difficult to say with confidence that such attacks are not in active use already. The technologies to detect such behaviors are just beginning to arise, and the difficulty of analyzing an entire app store (such as the Google Play Store) severely limits our ability to ensure that such an attack has not been deployed.

We believe that the FUSE tools and similar analysis efforts are making steps in that direction. FUSE incorporates a state-of-the-art static binary analysis of individual applications, mitigates the inevitable precision and recall challenges with a light-weight security lint tool, then leverages the results from the single-app process to build a multi-app information flow graph that can be automatically analyzed against a specified security policy, or explored interactively.

10. ACKNOWLEDGMENTS

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) Program Office under Contract No. W31P4Q-12-C-0024. The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

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

[2] L. O. Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, University of Cophenhagen, 1994.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 29, 2014.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: analyzing the Android permission specification. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 217–228, 2012.

[5] A. Chaudhuri. Language-based security on android. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, pages 1–7, 2009.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[7] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical Report CS-TR-4991, Department of Computer Science, University of Maryland, College Park, Nov. 2009.

[8] O. Lhoták and L. J. Hendren. Scaling java points-to analysis using SPARK. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 153–169, 2003.

[9] V. B. Livshits, J. Whaley, and M. S. Lam. Reflection Analysis for Java. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, pages 139–160, 2005.

[10] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: statically vetting android apps for component hijacking vulnerabilities. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 229–240, 2012.

[11] A. C. Myers. Jflow: Practical mostly-static information flow control. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 228–241, 1999.

[12] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 543–558, 2013.

[13] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[14] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES)*. IEEE, Sept. 2014. to appear.

[15] S. Sagiv, T. W. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings*, pages 651–665, 1995.

[16] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, 2012. ISBN 978-0-7695-4681-0.