Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps

FENGGUO WEI, University of South Florida
SANKARDAS ROY, Bowling Green State University
XINMING OU, University of South Florida
ROBBY, Kansas State University

We present a new approach to static analysis for security vetting of Android apps, and a general framework called Amandroid. Amandroid determines points-to information for all objects in an Android app component in a flow and context-sensitive (user-configurable) way, and performs data flow and data dependence analysis for the component. Amandroid also tracks inter-component communication activities. It can stitch the component-level information into the app-level information to perform intra-app or inter-app analysis. In this paper, (a) we show that the aforementioned type of comprehensive app analysis is completely feasible in terms of computing resources with modern hardware, (b) we demonstrate that one can easily leverage the results from this general analysis to build various types of specialized security analyses – in many cases the amount of additional coding needed is around 100 lines of code, and (c) the result of those specialized analyses leveraging Amandroid is at least on par and often exceeds prior works designed for the specific problems, which we demonstrate by comparing Amandroid's results with those of prior works whenever we can obtain the executable of those tools. Since Amandroid's analysis directly handles inter-component control and data flows, it can be used to address security problems that result from interactions among multiple components from either the same or different apps. Amandroid's analysis is sound in that it can provide assurance of the absence of the specified security problems in an app with well-specified and reasonable assumptions on Android runtime system and its library.

CCS Concepts: • **Security and privacy** → *Software security engineering*;

ACM Reference format:
Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2018. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. *ACM Trans. Priv. Sec.* 1, 1, Article 1 (January 2018), 31 pages.
https://doi.org/10.1145/3183575

1 INTRODUCTION

The Android smart-phone platform is immensely popular and has by far the largest market share among all types of smartphones worldwide. However, there have been widely reported security problems due to malicious or vulnerable applications running on Android devices [14, 22, 27, 39, 46, 52, 55, 60, 66, 67].

2471-2566/2018/1-ART1 $15.00
https://doi.org/10.1145/3183575

Many security problems of Android apps can be discovered by static analysis on the Dalvik bytecode of the apps, and there have been a number of earlier efforts along this line [6, 8, 10, 12, 20, 26, 34, 37, 43, 47, 51, 53, 58, 62, 65, 67]. Compared with dynamic analysis, static analysis has the advantage that a malicious app cannot easily evade detection by changing their behaviors in a testing environment, and it can also provide a comprehensive picture of an app's possible behaviors as opposed to only those that manifest during the test run. Due to the inherent undecidability nature of determining code behaviors, any static analysis method must make a trade-off between computing time and the precision of analysis results. Precision can be characterized as metrics on:

A *missed behaviors* (app behaviors missed by the analyzer that may present security risks, also referred to as false negatives), and
B *false alarms* (behaviors that an app does not possess but the analyzer fails to rule out, also referred to as false positives).

Android Static Analysis Challenges: A practical challenge in static analysis is to control the rate of false alarms while not missing any (potentially dangerous) behaviors of apps. This is especially significant due to a number of features of Android.

(1) Android is an event-based system. The control flow is driven by events from an app's environment that can trigger various method calls. How to capture all the possible control flow paths in this open and reactive system while not introducing too many spurious paths (false alarms) is a significant challenge.
(2) The Android runtime consists of a large base of library code that an app depends upon. The event-driven nature makes a large portion of the control-flow involve the Android library. While fully analyzing the whole library code could improve the analysis' faithfulness, it may also be prohibitively expensive (or imprecise).
(3) Android is a component-based system and makes extensive use of inter-component communication (ICC). For example, a component can send an *Intent* to another component. The target of an Intent could be specified explicitly in the Intent or be implicit and decided at runtime. Both control and data can flow through the ICC mechanism from one component to another. Capturing all ICC flows accurately is a major challenge in static analysis.

Prior research has attempted to address some of the above challenges. For example, FlowDroid [6, 24] formally models the event-driven lifecycle of an Android app in a "dummyMain" method, but it does not address ICC. Epicc [43] statically analyzes Intent and uses an IDE [49] framework to solve for Intent call parameters, but does not link the Intent call sources to targets and does not perform data flow analysis across component-boundaries. CHEX [37] uses a different approach to the modeling of the Android environment, by linking pieces of code reachable from entry points (called splits) as a way to discover data flows between the Android application components, but it does not address data flow through Intent channels. IC3 [42] is a composite constant propagation engine to solve Intent values in the whole application. IccTA [34] extends FlowDroid and uses IC3 as the Intent resolution engine, which can track data flows through regular Intent calls and returns. However, IccTA is yet to track a special category of ICC named *remote procedure call* (RPC) that invokes a method in a *bound* service component. DroidSafe [26] attempts to track both Intent and RPC calls. It performs an app-level analysis with flow-insensitive points-to information. None of the works mentioned above can capture data flows through "stateful ICC," where component A sends data to B through one ICC, and later component A retrieves that same data from B through another ICC.

All of these prior works have inspired our work. We designed and built **Amandroid**[1] – a component-based data flow analysis framework tailored for Android apps. The executable and source of Amandroid are publicly available.[2] **The main contributions from Amandroid are:**

(1) Amandroid computes points-to information for *all* objects and their fields at each program point and calling context. The points-to information is extremely useful for analyzing a number of security problems that have been addressed in prior works using customized methods. Amandroid can be used to address these wide-range security problems directly with very little additional work. We also show that such comprehensive analysis scales to large apps.
(2) As part of the computation of object points-to information, Amandroid can build a highly precise inter-procedural control flow graph (*ICFG*) of an app component , which is both flow and context-sensitive [40]. This is a side benefit of our approach compared to prior works that have adopted existing static analysis frameworks (*e.g.*, Soot [57] and Wala [23]), which build *ICFG* with less precision [32, 59].
(3) For each app component, Amandroid builds a Data Flow Graph (*DFG*), which consists of the component's *ICFG* together with each node's (in *ICFG*) reaching (points-to) fact set. Then, Amandroid builds the data dependence graph (*DDG*) for each app component from its *DFG*. Furthermore, for each app component Amandroid builds a summary table (*ST*) listing its inter-component communication (control and data flow) activities over multiple channels, such as Intent, RPC, and static fields. Amandroid is able to conduct an elementary string analysis (due to its object-sensitivity) for inferring Intent/RPC call parameters, and finds the correspondence between an ICC source and the ICC targets based on a flow/context-sensitive matching algorithm. Using *ST*s of multiple components, Amandroid can stitch the component-level *DDG*s into an app-level inter-component *DDG* that supports both intra-app and inter-app analysis.
(4) An analyst can add a plugin on top of Amandroid to detect the specific security problem he/she is interested in. Through extensive experimentation, we demonstrate that a variety of security problems can be reduced to querying *DFG*s and *DDG*s.

An earlier version of the work was published in ACM CCS 2014 [62]. The current version has substantial extension and enhancement from the original Amandroid tool:

(1) We designed a new component-based analysis algorithm, which supports reasoning information flow through stateful inter-component communication, such as `bindService`, RPC calls, and `startActivityForResult`.
(2) The new analysis approach obviates the need of computing an inter-component call graph. This avoids the call graph blowup problem due to the inherent over-approximation in intent resolution process, which could affect the original version of Amandroid as well as other Android analysis tools. More details are discussed in Section 4.3.
(3) We present more technical details on algorithms and implementations and extended experimental results.

We evaluated Amandroid on 4,600 real-world apps (2,300 Google Play apps shared by the AndroZoo [3] group, and 2,300 malicious apps from the AMD dataset [61]). Our experimental results show that Amandroid scales well. We used Amandroid to address security problems such as data leakage (*e.g.*, SMS message leakage), injection (*e.g.*, intent injection), and misuse/abuse of APIs (*e.g.*, to hide app icon). The core framework of Amandroid takes several minutes to analyze one

[1] Aman means safe/secure in the Indonesian language.
[2] Amandroid is available at http://pag.arguslab.org/argus-saf with new name Argus-SAF (Argus Static Analysis Framework).

app on average. All the specialized analyses require very little additional coding effort (around 100 LOC) to leverage Amandroid's *DFG*s and *DDG*s to address the specific problem, and the additional running time is negligible (typically in the order of tens of milliseconds).

We then experimentally compare Amandroid with two state-of-the-art static analyzers for Android apps: IccTA [34] and DroidSafe [26], and show that Amandroid can address a wider range of security problems due to inter-component communications. Amandroid also found multiple crucial security problems in Android apps that were never reported before in the literature.

Organization. The rest of the paper is organized as follows. Section 2 gives a motivating example. Section 3 describes in detail Amandroid's analysis methods. Section 4 presents Amandroid's component-based analysis model. We discuss implementation details in Section 5, experimentation of our approach in Section 6, discuss limitations in Section 7 and related research in Section 8.

2 MOTIVATING EXAMPLE

A malicious app can conduct bad behaviors by leveraging the design (*e.g.*, event-driven and inter-component nature) of Android system and try to obfuscate its true objectives. Figure 1 shows an example app (named "IMEI-leaking"), which consists of a few components while each one is a separate Java class. We note that Android apps are component-based where each component is an independent entity and is typically responsible for a specific task. For instance, an `Activity` component implements the UI of the app, a `Service` component typically performs a long-running task on the background, and a `Broadcast Receiver` component receives a broadcast message from one component (or the system) and takes certain actions, and more.

An Android app does not have a "main" method; rather, components are invoked through the various callback methods (including *lifecycle methods*). Depending on the events, the system invokes the lifecycle methods of the components. It also remembers the recently sent `intents` and passes them around, which can be abstracted in a component-level `environment`. Furthermore, there can be control flows and data flows among the app components through the Android system. For comprehensive analysis, the app analyzer tool needs to track such control and data flows.

As an example, the following sequence of events (as labeled in Figure 1) can happen in reality:

(1) FooActivity starts BarActivity (via "`startActivityForResults`" API) and waits for BarActivity to send back some result.
(2) When the user clicks on a button of BarActivity screen, the `onClick` method is triggered.
(3) BarActivity makes an RPC (Remote Procedure Call) call `getImei()` to a Service component named `MyService`, and `MyService` returns an inner field (which has already possibly stored the `IMEI` Id) to `BarActivity`.
(4) `BarActivity` sends back an intent (via `setResult` API), which contains the `IMEI` Id.
(5) Android system invokes `onActivityResult` method of `FooActivity` with the above intent as a parameter, and the `IMEI` Id is extracted and leaked (to the attacker) through a SMS message.

To track the data and control flow inside a component, a static analyzer needs a model of the Android system to track invocation of the callback methods including the component lifecycle methods as illustrated in the above example. Our model of the Android environment is inspired by FlowDroid [6, 24], which uses a "dummyMain" method to capture all possible sequences of lifecycle method invocations as followed in Android. However, unlike an app-level environment model used in FlowDroid, we design a component-level environment model. The motivation behind the component-level model choice is that Android apps work in this way.

Furthermore, we need to track data and control flow through each type of inter-component communication channel (*e.g.*, Intent, RPC, *etc.*). As an example, when `BarActivity` sends out

```
1  public class FooActivity extends Activity {
2    @Override
3    protected void onCreate(Bundle b) {
4      super.onCreate(savedInstanceState);
5      setContentView(R.layout.activity_foo);
6      Intent i1 = new Intent();
7      i1.setClass(getApplicationContext(), BarActivity.class);
8      startActivityForResult(i1, 1);
9    }
10   @Override
11   protected void onActivityResult(int req, int res, Intent
         data) {
12     String imei3 = data.getStringExtra("key");
13     SmsManager smsManager = SmsManager.getDefault();
14     smsManager.sendTextMessage("xxx", null, imei3, null, null)
         ; // sink, leak of device id
15   }
16 }
```

```
17 public class BarActivity extends Activity {
18   private MyService s;
19   @Override
20   protected void onCreate(Bundle savedInstanceState) {
21     super.onCreate(savedInstanceState)
22     setContentView(R.layout.activity_bar);
23     Intent i2 = new Intent();
24     i2.setClass(getApplicationContext(), MyService.class);
25     bindService(i2, mConnection, Context.BIND_AUTO_CREATE);
26   }
27
28   @Override
29   protected void onDestroy() {
30     super.onDestroy();
31     unbindService(mConnection);
32   }
33
34   // button click callback defined in activity_bar.xml
35   public void onClick(View view) {
36     if(s != null) {
37       String imei2 = s.getImei();
38       Intent i3 = getIntent();
39       i3.putExtra("key", imei2);
40       setResult(RESULT_OK, i3);
41       finish();
42     }
43   }
44
45   private ServiceConnection mConnection = new
         ServiceConnection() {
46     public void onServiceConnected(ComponentName className,
         IBinder binder) {
47       MyService.MyBinder b = (MyService.MyBinder) binder;
48       s = b.getService();
49     }
50     public void onServiceDisconnected(ComponentName
         className) {
51       s = null;
52     }
53   };
54 }
```

```
55 public class MyService extends Service {
56   private final IBinder mBinder = new MyBinder();
57   private String imei1 = null;
58   @Override
59   public IBinder onBind(Intent intent) {
60     return mBinder;
61   }
62   public class MyBinder extends Binder {
63     MyService getService() {
64       return MyService.this;
65     }
66   }
67   public void setImei() {
68     TelephonyManager m = (TelephonyManager) getSystemService(
         TELEPHONY_SERVICE);
69     this.imei1 = m.getDeviceId(); // source
70   }
71   public String getImei() {
72     return this.imei1;
73   }
74 }
```

Fig. 1. The IMEI-leaking App: The arrowed lines among the app components highlight some of the inter-component-communication.

an intent `i3` via `setResult()` API, the Android system invokes `onActivityResult` method of `FooActivity` with `i3` (*i.e.*, *data* = *i3*) as a parameter. The reason for the above action is that `FooActivity` has started `BarActivity` before with the `startActivityForResults()` API. To track the control and data flow involved in such a "stateful" ICC (inter-component communication) mechanism, the analyzer tool needs to remember which Activity has started a given Activity `A`. Another challenge for the analyzer tool is how to track the RPC channel, if any. As an example, when `BarActivity` invokes the `getImei()` method, the analyzer tool has to map the call to the corresponding method of `MyService` component. `BarActivity` receives some data flow as the return from the call. Furthermore, `MyService` might have been running already before this RPC takes place and has stored the IMEI Id in field `imei1` (*e.g.*, because another RPC method `setImei()` got invoked by others), and the `getImei()` call returns the sensitive information from `imei1` to `BarActivity`. This shows that the analyzer tool needs to address the *re-entry* nature of the component code. In addition to the above channels of communication among app components, two components can also exchange data via *static variables* and more. So, the app analyzer tool needs to track these channels too.

3 THE AMANDROID APPROACH

Figure 2 illustrates the pipeline of Amandroid's main steps:

(1) Amandroid converts an app's Dalvik bytecode to an intermediate representation (`IR`) amiable to static analysis.
(2) It generates an environment model that emulates the interactions of the Android System with the app.

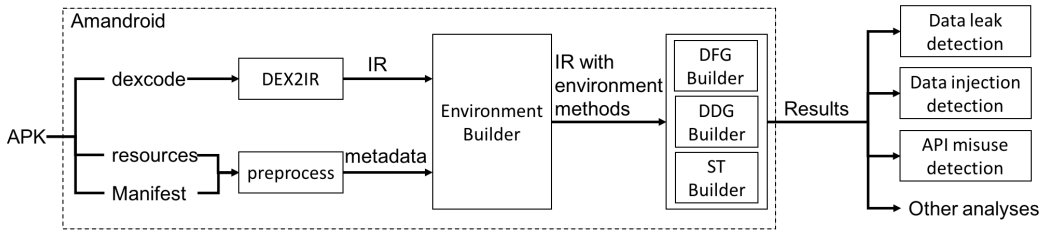


Fig. 2. The Amandroid Analysis Pipeline

(3) Amandroid does a component-based analysis. In particular, for each component of the app, it builds a data flow graph (*DFG*), which includes the control flow graph of the component plus the *points-to* information. Furthermore, Amandroid builds the component-level data dependence graph (*DDG*) on top of the *DFG*, which implies explicit information flow. Amandroid also builds a summary table (*ST*) documenting the component's possible communication channel with other components. Later, if necessary, an app-level *DDG* is built by stitching together the individual components' *DDG*s.
(4) Amandroid then can be applied in various types of security analysis using the information presented in *DFG*s and *DDG*s. For example, one can use *DDG* to find whether there is any information leakage from a sensitive source to a critical sink by querying whether there is a data dependence chain from source to sink.

3.1 IR Translation

Amandroid decompresses the input app apk file, retrieves a dex file, and coverts it to an `IR` format for subsequent analysis. The `IR` we use is called `Pilar`, a language designed by one of the co-authors and used in static analysis and model checking frameworks such as Bakar Kiasan [29] and Bakar Alir [54]. `Pilar` is a structured and annotation based language, where user can add different types of annotations to support different languages. This gives us the flexibility to extend Amandroid to support analysis for other frameworks and languages in the future. We wrote a translator `dex2IR` that takes as input the dalvik bytecode of an Android app, and outputs the program in `Pilar`.

3.2 Environment Modeling

An Android app is not a closed system; the Android system provides an environment in which the app runs. The code that may execute during the lifetime of an app is not all present in the app's package. The Android system (which includes the Android runtime) does a bulk of the work in addition to that by the app's code. With the "IMEI-leaking" app example in Section 2, we demonstrated that a static analyzer needs to model the Android system to analyze the system-defined control flows in the app[3]. Our modeling of the Android environment follows that of FlowDroid [6, 24] with a few crucial extensions described below.

In Android, numerous types of events (*e.g.*, system events, UI events, *etc.*) can trigger callback methods defined in an app. As an example, while an Activity `A` is running, if another Activity `B` comes to the foreground, it is considered an event. This event can trigger `A.onPause`, which is either defined in the app's code, or in the Android framework if the developer did not override the default method. There are seven important lifecycle methods of an Activity: `onCreate`, `onPause`, `onResume`,

[3]The alternative is to fully analyze the whole Android system's code, which is both expensive and unnecessary as also observed by others [24, 26, 34, 37].

Algorithm 1 Generating the Environment Method of Component C

Input: The name of the component C, manifest file, resource files, IR of C.
Output: C's environment method, Env_C

```
1:  procedure GenEnv(C)
2:      create a method Env_C having one parameter Intent i, and an empty body;
3:      callBacks ← collectCallbacks(C);
4:      add callBacks into the body of Env_C in the proper sequence emulating the reality;
5:      return Env_C;
6:  procedure collectCallbacks(C)
7:      callBacks ← empty Set;
8:      while fixed-point is not reached do
9:          perform reachability analysis to mark methods that are reachable from C
10:         callBacks ← callBacks ∪ callBacks from the XML-resource files
11:         callBacks ← callBacks ∪ interface-based callbacks as registered in C's source code
12:         callBacks ← callBacks ∪ other callbacks (system methods that are overridden) in C's source
13:     return callBacks;
```

etc.; they each represent a state in the transition diagram of the lifecycle. Android documentation specifies other states such as `Activity running` and `Activity shut down`. Similarly, other types of components (*e.g.*, Service, Broadcast Receiver, *etc.*) have a well-defined lifecycle involving multiple lifecycle methods.

Amandroid introduces component-level models instead of FlowDroid's whole app-level model. The environment of a component `C` represents a `main` method, `Env_C`, which takes as parameter an incoming intent `i` and invokes `C`'s lifecycle methods (*e.g.*, `onCreate`, `onBind`, or `onReceive`) based on `C`'s type (Activity, Service, Broadcast Receiver, *etc.*) and other callback methods (*e.g.*, `onLocationChanged`) so that all possible paths are included. This component-level model is more effective in capturing the impact of the Android system on both the control sequence and data flow of an app's execution. We have a dedicated environment for each component that invokes the set of callback methods implemented in the component; this is the control part of modeling Android's environment. In addition, the environment also keeps tracks of the intents received by the component (*e.g.*, Environment of `BarActivity` remembers the intents sent to start `BarActivity`) so that the intents could be made available when necessary (*e.g.*, to serve `getIntent()` at *L38* in the `BarActivity` component); this is the data part of modeling Android's environment. `Env_C` also passes the intent parameter when necessary for other relevant methods (*e.g.*, `onReceive` of a Broadcast Receiver).

Amandroid generates the Environment Method (`Env_C`) of each component `C` in the app automatically. Algorithm 1 shows the pseudocode for generating `Env_C` of a component `C`. As the first step, an empty method with an Intent *i* as the parameter is generated. (Note that Intent *i* typically represents the Intent which starts the component – for instance, *e.g.*, the parameter of Environment Method of `BarActivity` is basically the intent that starts `BarActivity`) Then, we collect basic information from the resource files in the apk and uses this information to collect layout callback methods. We then generate the body of `Env_C` with lifecycle methods based on the type of `C`. Finally, we collect other callback methods (*e.g.*, `onLocationChanged`) in `C` (through a reachability analysis) in an incremental fashion (following the FlowDroid [6] approach). All of these are done before performing the data flow analysis as discussed in Section 3.3 and Section 4.1.

3.3 Component-Based Analysis

Android is a component-based system, and hence analyzing the code at the component level fits more to the nature of Android applications. The example in Section 2 illustrates how data-flows can happen inside one component and across multiple components.

Amandroid takes each component's environment method as an entry point of analysis, and performs data-flow analysis as well as data dependency analysis.

For each component C that is reachable from outside, Amandroid builds a data flow graph (*DFG*). *DFG* includes the control flow graph spanning over all the reachable methods of C; it also tracks the set of object creation sites that reach each program point (thus, Amandroid knows the dynamic types of objects flowing to any particular program point, and where they were created and modified along the way). Then, Amandroid builds the data dependence graph (*DDG*) on top of the *DFG*, which implies explicit information flow. Amandroid also builds a summary table (*ST*) documenting the component's possible communication channel with other components. Later, when necessary, an app-level *DDG* is built by stitching together the individual components' *DDG*s. The detailed discussion and algorithm of how to perform those analyses, and the uses of such results will be presented in Section 4.

3.4 Using Amandroid for Security Analyses

Amandroid provides an abstraction of the app's behavior in the forms of *DFG*s and *DDG*s. We now discuss how they can be easily used for a number of useful security analyses.

3.4.1 Data Leak Detection. One important problem in app vetting is to find whether an app may leak any sensitive data. Examples of sensitive data include user-login credentials (*e.g.*, password), location information, and so on. This can be performed through standard data dependence analysis using the *DDG*. Given a source and a sink, one can find whether there is a path from source to sink in the *DDG*. For instance, prior research [7, 24] has documented a list of security-critical source and sink APIs, which can be used here. One could also customize the definition of the source and sink for the specific problem at hand. *DDG* can only capture explicit information leaks. For information leaks through controls (*e.g.*, leaking conditionals through the branches) one would need to build a *control dependence graph*, which can be obtained from the *DFG*s through the standard process [4].

Amandroid can perform a comprehensive analysis since it captures control and data flows across the component boundaries through Intent channel, RPC channel, and others so that security problems like the one shown in Figure 1 can be captured.

3.4.2 Data Injection Detection. An app can have a vulnerability which allows an attacker to inject data into some internal data structures, leading to security problems. Researchers [37] identified a subclass of this vulnerability called *intent injection*. The attacker can send an ill-crafted intent to a public component of a vulnerable app, which retrieves data from the incoming intent and uses it for security-sensitive operations. For instance, the app's logic can be such that the incoming intent determines the destination of a critical data flow — the *URL* of a backup server, the name of a file, the destination component of an ICC call, phone number of an outgoing SMS, or others. As a result, the attacker will be able to control the destination, which can lead to serious security problems.

Amandroid can detect this vulnerability using the *DDG*, by defining the source as the possible entry point of attacker-controlled data (*e.g.*, a public-facing interface), and the sink being the critical parameters of the security-sensitive operations. If a data-dependency path exists between the source and the sink, the attacker can potentially manipulate the parameters of the security-sensitive operations.

3.4.3 Detecting Misuse/Abuse of APIs. Another critical part of security vetting is to find if the developer (intentionally or unintentionally) has used a library API in an improper way, which may lead to security problems. Past research has applied static analysis to identify misuse of Crypto APIs [18] and SSL APIs [21]. The main idea is to detect if the app satisfies a set of rules on proper use of the APIs. For example, if the parameters for calling the AES encryption method have certain

values the cipher will run in the insecure ECB mode. Amandroid can verify these rules by checking the possible values of the parameter objects in a relevant API call by querying the *DFG*s.

4 COMPONENT-BASED ANALYSIS

An Android app might have multiple components while the components can communicate with each other via various channels: Intent, RPC, static field, *etc.* Thus security sensitive data items can also flow through these channels. Moreover, in an inter-app communication, one component of app X interacts with one component of app Y; hence, communication across different apps can be considered as inter-component communication. Thus our approach considers the component-based analysis as the basic building block for app vetting. We do both intra- and inter-component analysis (covering both intra-app and inter-app analysis, if necessary).

Determining object points-to information is a core underlying problem in almost all static analyses for Android app security, such as finding information leaks, inferring Intent calls, identifying misuse of certain library functions, and others. Instead of addressing each of these problems using different specialized models and algorithms, it is advantageous to pre-calculate *all* object points-to information at once, and use this as a general framework for different types of further analysis. This way the cost of computing points-to information is amortized across the large number of specialized analyses one will likely need to perform on a given app.

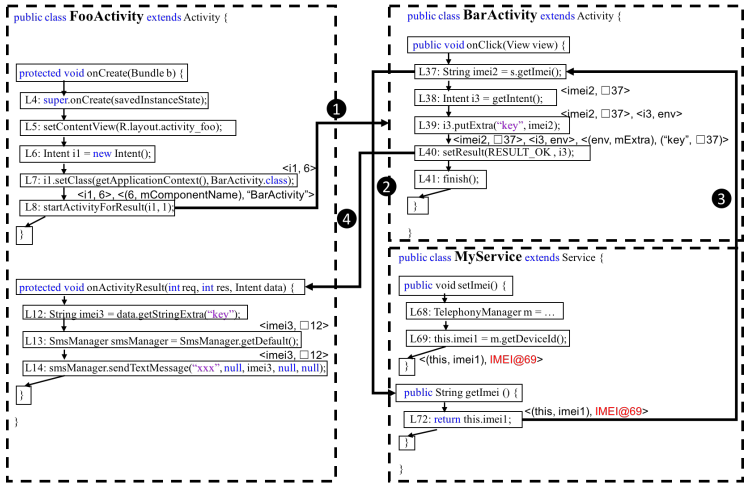
Existing off-the-shelf static analysis tools such as Soot [57] (used by FlowDroid [6, 24] and Epicc [43]) and Wala [23] (used by CHEX [37]) have not provided capability of calculating all objects' points-to information in a both flow and context-sensitive way [32, 59]. This is due to concerns about computation cost. However, with the advancements in hardware (*e.g.*, many-core machines), it opens new possibilities to perform a more precise analysis.

Generally speaking, the core task of Amandroid's analysis is aimed to build a precise inter-procedural data flow graph (*DFG*). The flow-sensitive and context-sensitive data flow analysis to calculate object points-to information is done *at the same time* with building inter-procedural control flow graph (*ICFG*). This is because in order for one to precisely know the implementation method of a virtual method invocation, one needs to know the receiver object's dynamic type; conversely, flow-sensitive data flow analysis requires one to know how the program control flows. Thus, there is a mutual dependency between the two analyses. Such integrated control and data flow analyses approach has been demonstrated to be both practical and effective for even analyzing temporal properties of *concurrent* Java programs including the standard Java library codebase [17]. However, [17] does not keep track of method calling context (typically termed *monovariant* calling context analysis or 0-calling context [40]). We generalize the approach to precisely track the last k calling contexts (*polyvariant* [40], *a.k.a.* k-limiting where k is user-configurable and the additional calling context beyond k is monovariant).

Our analysis approach consists of the following phases: (1) Build data flow graph (*DFG*) for each component (discussed in Section 4.1); (2) Build data dependency graph (*DDG*) for each component (discussed in Section 4.2); (3) Perform inter-component analysis (discussed in Section 4.3, 4.4, and 4.5).

4.1 Component-Level Data Flow Graph

Amandroid computes points-to facts for each statement. In the component-based analysis, we build the *DFG* of each component of an app. Due to space constraints, the description (including the algorithm and an example) of the basic *DFG* building process is presented in Appendix only. Below we introduce the notations in *DFG* and use the example app (ref. Figure 1) of Section 2 to explain its semantics. Figure 3 illustrates part of the resulting *DFG*s of the components in the example app.



FooActivity

Channel	Send-points	Receive-points
Intent	1. startActivityForResult()@L8: Intent@L6 with componentName("BarActivity")	1. ComponentName: FooActivity 2. IntentFilter: action("action.MAIN") category("category.LAUNCHER") 3. onActivityResult()
RPC		
Static Field		

BarActivity

Channel	Send-points	Receive-points
Intent	4. setResult()@L40	1. ComponentName: "BarActivity"
RPC	2. MyService.getImei()@L37	3. imei2@L37: return of MyService.getImei()
Static Field		

MyService

Channel	Send-points	Receive-points
Intent		1. ComponentName: MyService
RPC	3. Return from MyService.getImei()@L72	2. MyService.getImei()
Static Field		

Fig. 3. *DFG*s and *ST*s of the components in App "IMEI-leaking": An excerpt

4.1.1 Notations. There are two sets of facts associated with each statement: the set of facts entering into a statement s is called the *entry set* of s (or just $entry(s)$); the set of facts exiting a statement s is called the *exit set* of s (or just $exit(s)$). Statement s may change $entry(s)$ by killing stale facts ($kill(s)$) and/or generating new facts ($gen(s)$). The *gen* and *kill* sets can be calculated using flow functions that are based on s' semantics. In general, the flow equations have the following forms.

$$exit(s) = (entry(s) \setminus kill(s)) \cup gen(s) \tag{1}$$

Amandroid keeps track of *points-to facts*, which provide information about what objects a variable (register in Dalvik), an object field, or an array element may point to at a particular program point. A points-to fact has the general form of $\langle lhs, rhs \rangle$.

The *rhs* may refer to either an object or an aggregate (usually key-value pairs). Objects are dynamically allocated in the Dalvik VM heap space at *object creation sites* (through a "new" statement). In our IR, each statement in the program is assigned a unique location number N (represented as LN). We use this number to represent the fresh object created at the location, and refer to it as *instance N*. For example, (in Fig 3) location $L6$ generates the points-to fact $\langle i1, 6 \rangle$. Here 6 represents

instance 6, the object created at location *L6*. From the object creation site we can directly find the precise runtime type of the instance.

Let us use $\square N$ to indicate any possible value that is type compatible with the received objects at location N. For instance, for objects returned from inter-component communication such as RPC, we do not know the possible values that will be received from the communication. As an example, location *L37* generates a points-to fact $\langle imei2, \square 37 \rangle$, indicating that the string variable `imei2` points to an object that is returned from the RPC call at location *L37*. A `tuple-instance`, like ("key", $\square 37$) in the entry set of *L40*, denotes a key-value pair.

There are two types of *lhs* of a points-to fact, yielding two types of facts. A *variable-fact* is when the *lhs* is a variable. A *heap-fact* is when the *lhs* is an object field or an array element. For example, location *L7* generates a heap-fact $\langle (6, mComponentName), (\text{"BarActivity"}) \rangle$, meaning that the field *mComponentName* of *instance 6* points to the string "BarActivity".

4.1.2 Modeling Library and Native Calls. Android has a large number of library APIs (that an app can call) some of which are implemented natively. Similarly, an app developer may choose to natively implement some functionality (*e.g.*, for performance reasons). Amandroid does not analyze native code; thus, we need to provide models for native methods that summarize how the data flow facts may be changed. For library APIs that have well-understood simple semantics, one can summarize them as flow functions (*gen* and *kill*). Besides native methods, we also provide models for non-native library methods that are frequently used; this is useful to scale the analysis. In general, Amandroid adopts the following strategy in modeling Android library functions and native methods:

(1) For library functions that provide important information for static analysis (*e.g.*, intent manipulation functions), we manually build a precise model for them based on the function's implementation and/or documentation (each model simply consists of custom *gen* and *kill* functions).
(2) For all other library functions and native methods, we provide a uniform conservative model. The conservative model essentially assumes that for every object parameter, any of its fields may be modified and becomes *unknown*; that is, the field can point to a fresh object, or any existing object reachable from the method parameters (and static fields) that is type compatible. If the function also returns an object, the returned object is also considered unknown.

In Figure 3, line *L39* inserts a key-value pair ("key", `imei2`) into intent `i3`'s `mExtras`[4] field. The `putExtra` is an Android system API and we model it so that we can keep track of the data flow through the call. In this case, the model of the API will assign the key-value pair to the `mExtras` field of intent `i3`. The generated fact at Line *L39* is then $\langle (env, mExtras), (\text{"key"}, \square 37) \rangle$ following our notation for a field-fact, where `env` represents the creation site of intent `i3`, and $\square 37$ represents the String object `imei2` points to. Note that `env` represents the entry point of the environment method of `BarActivity`.

4.1.3 Handling Inter-component Channels. During the intra-component analysis phase, one cannot tell what data will be received by this component from others through inter-component

[4]The `mExtras` field is an aggregate object that may store multiple key-value pairs. We currently do not model such aggregates and instead "flatten" all the elements in an aggregate into singleton instances. This will create two possible interpretations of multiple facts regarding an aggregate object: either they are different possibilities from different program branches, or they are part of a single aggregate in the same branch. Amandroid's static analyzer conservatively assumes both are possible to ensure soundness, but this could lose some precision. Modeling aggregates is an engineering work that we will address in future work.

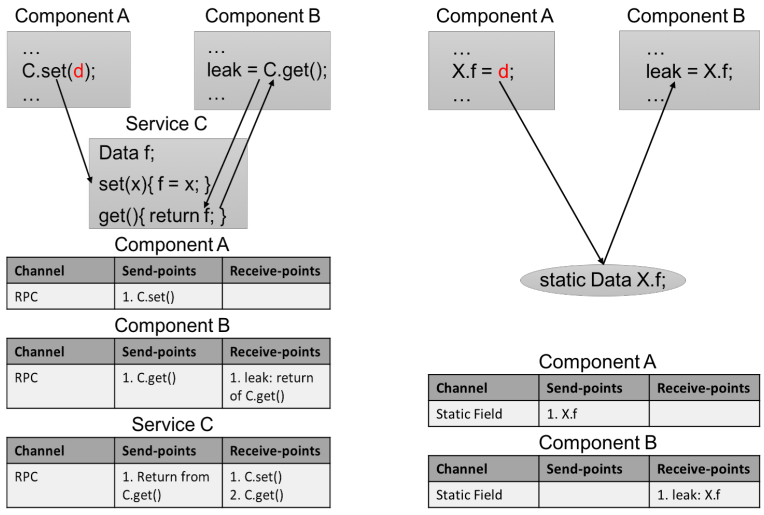


Fig. 4. Data flow between app components via RPC and Static Field.

channels, *e.g.*, Intent, RPC, static field, *etc.* Thus, at any information retrieval point for those channels we apply a conservative model like that used in Section 4.1.2. More detailed discussion on how to handle data flows across components will be discussed in Section 4.3.

4.2 Building the Component-Level Data Dependence Graph

A component-level data dependence graph (*DDG*) is derived from the component's *DFG*. With the help of *DDG*, we can determine which part(s) of the program a particular program point depends on. *DDG* is a directional graph; its node set is the same as the nodes in *DFG*, and has two types of edges: **(i)** object dependence edge – linking the use site of an instance to the creation site of the instance, and **(ii)** variable def-use edge – linking a use site of a variable to the def-site of the variable.

Since object flows in a component are captured in *DFG*, the constructed *DDG* automatically captures data dependencies within the component boundary. As an example, in Figure 3, the *L14* in *FooActivity* uses `imei3` while the *entry* of statement *L14* has a fact $\langle imei3, \square 12\rangle$. This tells us that the object $\square 12$ (generated at *L12*) is used in statement *L14*. Thus there is a data dependence path from *L14* of the `FooActivity` to the def-site *L12* in the same component.

4.3 Linking Inter-component Data Flows

When components interact through Inter-component communication (ICC) channels, the dataflow facts will propagate from one component to another. There are a couple challenges in analyzing inter-component data flows for Android apps.

(1) Android app components run concurrently and their execution sequence can be arbitrarily interleaving or parallel depending on the events that trigger the various call-back methods.
(2) Android app components are stateful. After component `A` invokes ICC on component `C` and changes its state, another component `B` may invoke ICC on `C` later and be impacted by the effect of the previous ICC from `A`.

Figure 4a shows a case where a Service C has a field f and two RPC methods set() and get() which set and get data from field f, respectively. These two RPC methods can be invoked in any order with any data from all other components. For example, component A may set a sensitive data into Service C's field f, and component B could retrieve such data from C via the get() RPC call later, forming an information flow path. Figure 4b shows another case where component A, B share data via static field X.f, which can form an information flow path from A to B.

Traditional context-sensitive call graph generation cannot capture this type of information flow from "stateful ICC." In the above example, neither the call sequence A -> C nor B -> C can capture the information flow A->C->B. The information flow only happens through interleaving the three components' execution in the order {A, C, B, C}, where the first two captures the RPC call A -> C and the latter captures the RPC call B -> C. Such concurrency execution semantics can be modeled by treating ICC in a context-insensitive manner, and merging all the dataflow facts at a component's ICC entry point – simulating the effect of all possible orders of interleaving.

Based on this idea, one approach is to compute a global fixed-point among all the components while flowing the points-to facts context-insensitively between components (intra-component dataflow is still context-sensitive).[5] The downside is that for any new set of components we want to analyze, we would have to re-compute the global fixed-point, making it impossible to re-use the per-component analysis result. Thus we adopt a different approach. When computing the *DFG* for each component in the intra-component analysis phase, we assume that any type-compatible data is possible to enter the ICC channels. In addition we book-keep all the data that may enter and leave the component through the channels. In the inter-component analysis phase, we then "stitch" the inter-component communication channels' receive points with the corresponding send points (between two different components), forming the inter-component data dependence graph.

This conservative approximation serves the purpose of our goal well: 1) Android is a component-based system and any component may receive data from any other component – not necessarily the ones in the same app; thus assuming any type-compatible data may come into the ICC channel is consistent with Android's execution semantics; 2) This reasoning model obviates the need of computing ICC call graphs, thus eliminates the call graph explosion problem that may happen in other Android analysis tools, including the original version of Amandroid [62]; 3) By analyzing each component separately, it allows us to re-use the intra-component analysis result for any further inter-component analysis, possibly involving different subsets of the components. This will scale better with large volumes of apps and naturally extends to inter-app analysis.

In the inter-component analysis phase the *DFG* of all the involved components are loaded. Based on the ICC channel book-keeping information we then find the data dependence between the sender and recipient points. The book-keeping information is stored in a data structure called the *summary table* (*ST*). We generate an *ST* for each component C via processing C's *DFG*, where *ST* lists the communication channels through which C communicates with other components. *ST* records specification of different types of channels including *e.g.*, Intent, RPC, and static fields[6]. In particular, for each such channel the *ST* of C records the following items: (1) *send-points* where C is the sender of the channel. The information recorded includes what kind of data is sent (*e.g.*, outgoing Intent value for an Intent channel) and the receiver's name. (2) *receive-points* where the component C is the receiver of the channel. The recorded information includes receiver's name which allows matching with other components' send-points. For example, for Intent channel, the

[5]It is quite non-trivial to compute this global fixed-point while at the same time simulating the non-determinism caused by the interleaving concurrent threads [17].

[6]Files can serve as an inter-component communication channel like static fields, and can be handled in a similar way. This would require a precise string value solver, which we leave for future work.

intent filter value; for RPC channel, the RPC method's signature, and so on. Table 1 lists the main items in a *ST*.

Table 1. Communication points of an app component as listed in its Summary Table

Channel	Send-points	Receive-points
Intent	Outgoing Intent	Intent Filter
RPC	Method signature, params, return	Method signature, params, return
Static Field	Field signature to write, data	Field signature to read

With the help of Figure 4, we now discuss how the *ST*s are constructed and used. There are three components in Figure 4a, whose *DFG* has already been built. In component `A`, we saw a RPC call `C.set(d)` that sends data `d` to Service `C` via the RPC channel `C.set()`. We add this to the RPC channel's send-point description in `A`'s *ST*. Component `B` has a RPC call `C.get()` which sends a request to Service `C` and expects a return value from it. We add it to both the send-point and receive-point description of `B`'s RPC channel. Service `C` has two RPC methods `C.set(x)` and `C.get()`; we add them to the receive-point of `C`'s RPC channel. `C.get()` is returning a value back to its caller; we add it to the send-point of `C`'s RPC channel. Figure 4b shows the inter-component communication caused by static field. Here the send-point description indicates a write to the static field, and a receive-point description indicates a read from the field. With the *ST*s for each component constructed, we can "stitch" the send and receive points of the channels between two components to identify all possible inter-component data dependency. The "stitching" process is basically matching each channel's send-point with receive-point between two components based on channel specific criteria. For example, in Figure 4a we can stitch component `A`'s send-point `1` to component `C`'s receive-point `1`, because their method signatures match. After "stitch" all the send-points and receive-points (the arrows shown in Figure 4a), we can easily see the information flow path from `d` in component `A` to `leak` in component `B`.

In the next three subsections we further discuss the *ST* construction and this "stitching" process for each type of ICC channels.

4.3.1 *Intent.*

ST Construction.

Section 2 illustrates that malicious apps can easily manipulate Android's inter-component communication (ICC) to stealthily leak sensitive data. To track data flow through the Intent channel we need to solve statically certain values for the intent involved. At a send-point we need to solve for the Intent call parameters to infer the value of the outgoing Intent, so we can match it with the correct receive-points. At the receive-point we need to discover the Intent filter value so we can match it with the possible send-points. Amandroid infers the Intent API call parameters and Intent filters using the points-to facts computed and the app manifest file. This information will enable us to discover the source-destination component pair of the Intent call in the inter-component analysis phase.

The destination of an Intent can be either explicitly or implicitly specified in the outgoing intent. The common way of creating an *explicit intent* is by adding the destination component's name using Android APIs such as `setClass` (*L7* in Figure 3). For instance, at *L8* in Figure 3 Amandroid can derive that the intent parameter `i1`'s field `mComponentName` is "BarActivity." This fact comes from the modeling of the API function `setClass` called at *L7*, which generates a field-fact ⟨*(6, mComponentName), "BarActivity"*⟩, where *6* represents Intent `i1` which was created at *L6*. We record the destination component name as a send-point in *ST*. Also, we document in *ST* whether the

Intent caller expects a result returning later from the callee component (in case of stateful Intent call like "startActivityForResult" as opposed to stateless Intent call like "startActivity", "bindService", *etc.*).

An *implicit intent* does not include the name of a specific destination component, but instead requests a general *action* to perform, and the System finds a capable component (from the same app or another) which can fulfill the request. Some fields of an Intent object are used in this matching: `mAction` (String), `mCategories` (set of String), `mData` (Uri), and `mType` (String). These intent fields can be manipulated by invoking certain Android APIs. For instance, `i.setData(Uri.parse(http://abc.com/xyz))`, which sets the `Uri` corresponding to a *http url* to the `mData` field of an Intent `i`. Through proper modeling of these API functions (Section 4.1.2), Amandroid can derive possible (String) values of the relevant fields of an Intent object, which the Android system bases its decision on Intent destinations. Amandroid documents these fields of the Intent as send-points in *ST*.

Stitching Intent channels – Intent destination resolution.

For explicit intents it is straightforward to find the correspondence between the source component and the destination component. The matching information is directly available as the send-point (in the *ST*) of the source component and as the receive-point (in the *ST*) of the destination component. For example, FooActivity has a send-point at *L8* (`startActivityForResult()`) where Intent `i1` has the target component name set to "BarActivity", which matches the receive-point in the *ST* of BarActivity. Hence we discover the correspondence.

However, tracking the "return" intent `j` sent by the callee component `X` in a stateful Intent is more complicated, *e.g.*, the name of the destination component of the intent `i3` sent through the "setResult" API as in *L40* of BarActivity is not available in the app code (neither in the *ST* of `BarActivity`). To know the possible destinations of intent *j*, we first check through all components' *ST* to find components `Ys` which have initiated a stateful Intent call (*i.e.*, `startActivityForResult`) to component `X` (*e.g.*, `BarActivity`). Then, we infer that `onActivityResult` API of each of components `Ys` will receive intent `j` as a parameter.

Furthermore, there are some challenges in resolving the target of an implicit intent. The Android system finds the destination based on the intent fields as well as the manifests of all the apps which specify *intentfilters* for a component. An *intentfilter* is an XML expression involving the *action* tag, *category* tag, and *data* tag (which includes both `Uri` and `type`). The Android system determines the destination of an implicit intent by applying a set of rules [1] matching the relevant intent fields and the intent filter specification for every component on the system. Amandroid implements all those matching rules, using the static analysis results that show the possible string values of the relevant intent object fields. It runs a precise *actiontest*, *categorytest*, and *datatest* (having both `Uri` and `type`) to find the destination component(s). Our static analysis can readily handle `Intent` fields. For complicated String operations (*e.g.*, concatenation in a while loop), if Amandroid cannot infer the exact string value, it reports it as *any* string, ensuring the soundness of our analysis. We are able to run the *Uri test* matching different parts of the `Uri` (*e.g.*, `scheme`, `path`, `host`, `port`) between the intent and an intent filter. Furthermore, Amandroid is also able to find the specifications of dynamically registered Broadcast Receivers, if any.

4.3.2 RPC.

ST Construction.

A service provides the programming interface that a client component can use to interact with. This allows a client component to send/receive data to/from the service via a RPC call. In the example app of Figure 1, `MyService` defines an inner class `MyBinder` which extends the `Binder` class, and returns such a Binder instance in `onBind()` lifecycle method. `MyBinder` returns handle of

MyService which exposes two RPC methods, MyService.setImei() and MyService.getImei(). BarActivity binds to MyService at *L25* which uses a ServiceConnection defined at *L45*. After the bind succeeds, it will set the above handle to the s field of BarActivity. At *L37* when user clicks on a button at BarActivity, it will invoke the RPC call of MyService.getImei() to retrieve data from MyService.

Fortunately, in static analysis, discovering the above RPC connection between two components (intra-app, or *Local Service*) is straightforward. After resolve bindService() call at *L25*, we know the target service is MyService. Then at *L37* we know the target method's signature is MyService.getImei(). In addition to the *Local Service* (intra-app) case above, there are two more cases, *Messenger Service* and *AIDL* (*a.k.a. Remote Service*), which allows both intra- and inter- app RPC calls. For *Messenger Service* case, we first infer the Handler type registered to the Messenger instance that used at the service side, and mark the Handler's handleMessage() as the RPC callee. At the client side, we mark the invocation of Messenger.send() as the RPC caller. *AIDL* case is like the local service case, we can resolve the bindService() call to find target service and then find the RPC callee. For both the caller component and the callee component, we document the RPC method signature, parameters, return variable (some as send-points and some as receive-points) in *ST*.

Stitching RPC channels. Amandroid first evaluates Intent channel of *ST* to find the binding relation between client component and service component. Then, based on the binding relation to match the RPC caller and callee. For *Local Service* and *AIDL* case, we match the call signatures to link the RPC caller and RPC callees. For *Messenger Service*, we match the Messenger.send() to Handler.handleMessage().

4.3.3 Static Field.

ST Construction. Documenting static field is straightforward as each static field has its unique name. In our *ST* we just need to record from which program point which static field is read (receive-point) or written to (send-point).

Stitching static field channels. We just need to match the static field's name at send-point and receive-point to make the connection.

4.4 Building App-level Data Dependence Graph

After figuring out all the channel matchings, we connect the data dependency links among components' *DDG*s to build an app-level *DDG*. The time complexity of this stitching process is in the worst case quadratic to the number of components being analyzed. Then we can perform data dependency analysis of the app. For instance, to query the data leakage on the example app in Figure 3, we can find a taint source at MyService.setImei() method – any other component can use this RPC call to set the phone IMEI to the MyService.imei field. Then at the MyService.getImei() RPC method the return point can get IMEI and return back to *L39* at BarActivity; then it puts this information into Intent i3's mExtra field, and at *L40* sends as a result Intent to the caller component FooActivity. At FooActivity.onActivityResult(), *L6* extracts IMEI and sends it out via sendTextMessage(), which is a sink point.

Since *DDG* is a directed acyclic graph, the complexity of shortest path finding is linear to the number of nodes and edges, which in the worst case is quadratic to the size of all the components combined. Thus, even if over-approximation in the Intent destination resolution resulted in spurious data dependence paths, it will not have a substantial impact on the running time.

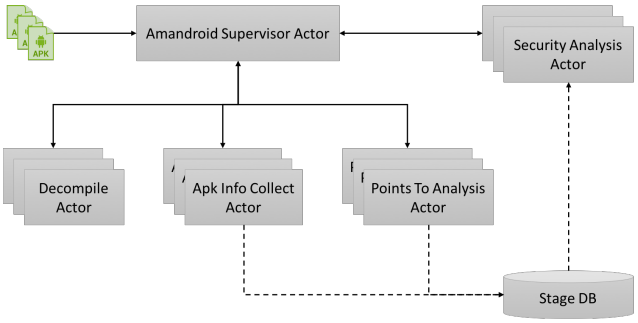


Fig. 5. Amandroid actor model.

4.5 Inter-app Analysis

Inter-app communication is simply inter-component communication which passes control and data across app boundaries. Thus, our component-based analysis can be directly used to perform inter-app analysis. However, it has some challenges.

(1) Only a subset of ICC channels can be used for inter-app communication. For example, local service does not support another app to bind to it; static field only allows the same app to read and write as they run in the same JVM.
(2) Multiple apps may share the same package and class name which can cause trouble for static analysis tool if it is not aware of the different app contexts.

To address challenge (1), Amandroid manages different scopes for different ICC channels. When linking the inter-component data dependence, it knows which channel can cross app boundary. To address challenge (2), Amandroid uses different class loaders for different apps, and in the stitching phase it adds origin information for each program point to avoid any naming conflict.

5 IMPLEMENTATION

Amandroid's modules are implemented using Scala, leveraging Akka's actor-model [2] to achieve distributed computation. Actor-model is a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent computation [63]. Each actor is a computation unit which maintains its private state and can only affect each other through messages to avoid usage of any locks.

As Figure 5 indicates, Amandroid's individual phases are encapsulated as actors whereas each of them maintains its own state and behavior. *Amandroid Supervisor Actor* is responsible for handling the user's app analysis request and dispatching orders to individual worker actors, and based on the response (of worker actors) moving the analysis to the next phase. Each phase of the analysis has multiple worker actors that perform the computation concurrently, leveraging parallel computing power. The actors communicate with each other with only a small amount of data; thus Amandroid could run in a highly distributed fashion.

The component level *DFG*, *DDG* and app metadata make the core information to be used in the security analysis phase. New security analyses may be needed to be performed from time to time while we observe that the required core information is the same for the same app. Thus storing the core information can save huge amount of compute time. However, the data dependency graphs can be quite big (GBs for a typical app). Thus we do not attempt to store the graphs, but rather only store the dataflow facts computed during the static analysis phase. The graph structure can be reconstructed efficiently when needed. This *staging* strategy is illustrated in Figure 5. *Apk*

Info Collect Actor and *Points To Analysis Actor* store the collected apk information and computed dataflow facts into the stage database, which can be used to rebuild the component-level *DFG*s, *DDG*s for the *Security Analysis Actor*. The dataflow facts stored in the database does not take much space — few MBs for an app.

Amandroid not only can do dataflow-based analysis, but also can be used as a general-purpose static analysis framework for Android apps. Amandroid provides comprehensive functionalities and APIs for other tools to build on, and performs analysis ranging from simple information collection to data flow/dependence analysis.

6 EXPERIMENTATION AND EVALUATION

We extensively experimented Amandroid in multiple types of security analyses. We used several sets of apps: 2,300 popular apps from Google Play, 2,300 malware apps from the AMD dataset [61], and two benchmarks (hand-crafted apps by other researchers and us). For brevity, we call the first two data sets GPlay and MAL, respectively.

To evaluate the effectiveness of Amandroid, we aim to answer the following research questions:

RQ1: How does the running time of Amandroid scale?
RQ2: How accurate is Amandroid compared with other tools?
RQ3: How effective is Amandroid's Intent resolution for real-world apps?
RQ4: Is Amandroid capable of discovering crucial security issues to aid in real-world app vetting?
RQ5: How much effort does it take to build a new analysis on top of Amandroid core framework?

We ran our experiments on a machine with 2.7 GHz, 12-core Xeon, and 64 GB RAM.

6.1 RQ1: How does the running time of Amandroid scale?

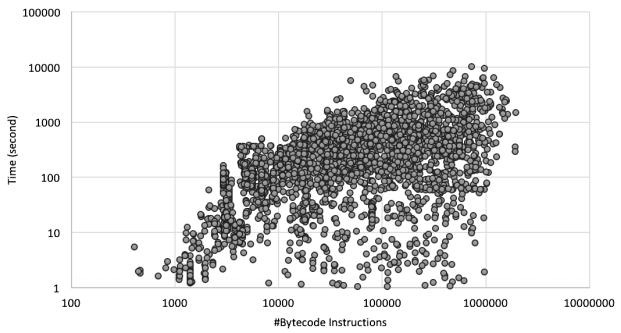


Fig. 6. Time to Build *DFG*.

Amandroid offers the user options of choosing multiple precision levels. For instance, the context depth k (of the control flow graph) serves as a parameter to set the trade-off between precision and performance. Our reported experiment results correspond to $k = 1$ (unless otherwise mentioned), meaning that the static analyzer tracks up to one calling context. Amandroid also allows the user to define the scope of the analysis by applying conservative model for certain third-party libraries (section 4.1.2). In our experiment we applied the conservative model for about 100 well-known third-party libraries.

The most computation-intensive step in Amandroid is building the *DFG* for each component. Once the *DFG* is built, the running times of the subsequent analyses are negligible – these include building *ST*, *DDG* and running the specialized analyses on top of them. Figure 6 presents the time taken by Amandroid to construct *DFG* for 4,600 real-world apps (GPlay and MAL).

These apps have 141319.50 lines of bytecode instructions on average. The median running time for computing the *DFG* for all the components in an app is 3 minutes; the minimum is 0.15 seconds whereas the maximum is 169 minutes. The scatter plot shows both the running time and the size of the app (in number of bytecode instructions).

We observe an increase in running time of new Amandroid compared to the original version [62]. The reason is two-fold: (1) The complexity of Android apps (*i.e.*, the dataset on which Amandroid runs) has increased over years, and the dataset we used in this experiment is more recently collected; (2) The new Amandroid has a more complete model (*i.e.*, component-based analysis as discussed in Section 4) to simulate the semantics of Android application, which was not captured in the old version.

6.2 RQ2: How accurate is Amandroid compared with other tools?

We use two benchmarks, Droid-Bench and ICC-Bench to compare Amandroid with two most well-known static analysis tools for Android: IccTA [34], and DroidSafe [26]. The benchmark testsuites consist of hand-crafted apps designed to test certain analysis features. Since those apps are hand-crafted, the ground truth is known, which allows us to compute metrics such as precision and recall. However, one needs to keep in mind that these metrics are not representative of the performance of the tools on real-world apps. They can only be used for comparison purposes.

Table 2. Results on Benchmarks. O = True Positive, * = False Positive, X = False Negative.

(a) Droid-Bench

App Name	IccTA	DroidSafe	Amandroid
Inter-component Communication (ICC)			
ActivityCommunication1	O	O	O
ActivityCommunication2	OO*	OO	OO*
ActivityCommunication3	X	O	O
ActivityCommunication4	OO*	OO	OO
ActivityCommunication5	O	O	O
ActivityCommunication6	X	O	O
ActivityCommunication7	O	O	O
ActivityCommunication8	OO*	OO	OO
BroadcastTaintAndLeak1	OO	OX	OO
ComponentNotInManifest1			
EventOrdering1	O	O	O
IntentSink1	O	O	O
IntentSink2	O	O**	O
IntentSource1	O	O	O
ServiceCommunication1	X	O**	O
SharedPreferences1	O	O	O
Singletons1	X	O	X
UnresolvableIntent1	OOO	OOO	OOO
Sum, Precision and Recall — ICC			
O, higher is better	19	22	22
*, lower is better	3	4	1
X, lower is better	4	1	1
Precision p = O/(O + *)	86%	85%	96%
Recall r = O/(O + X)	83%	96%	96%
F-measure 2pr/(p + r)	85%	90%	96%
Inter-app Communication (IAC)			
Echoer SendSMS StartActivityForResult1	N/A	N/A	O^{14*5}
Precision and Recall — IAC			
Precision p = O/(O + *)			74%
Recall r = O/(O + X)			100%
F-measure 2pr/(p + r)			85%

(b) ICC-Bench

App Name	IccTA	DroidSafe	Amandroid
Part A — Intent Addressing			
Intent_Explicit1	O	X	O
Intent_Implicit_Action	OO	XX	OO
Intent_Implicit_Category	OO	XX	OO
Intent_Implicit_Data1	OO	XX	OO
Intent_Implicit_Data2	OO	XX	OO
Intent_Implicit_Mix1	OOO	XXX	OOO
Intent_Implicit_Mix2	OO	XX	OO
Intent_DynRegisteredReceiver1	OO	XX	OO
Intent_DynRegisteredReceiver2	OO*	XX	OO*
Part B — Intent Data Flow Tracking			
Intent_Explicit_NoSrc_NoSink			
Intent_Explicit_NoSrc_Sink			
Intent_Explicit_Src_NoSink			
Intent_Explicit_Src_Sink	O	X	O
Intent_Implicit_NoSrc_NoSink			
Intent_Implicit_NoSrc_Sink			
Intent_Implicit_Src_NoSink	O	X	O
Intent_Implicit_Src_Sink	OO	XX	OO
InteneIntentService	O	X	O
Intent_Stateful	OOO	OXX	OOO
Part C — RPC			
RPC_LocalService	O	X	O
RPC_MessengerService	X	X	O
RPC_AIDL	X	X***	O
RPC_ReturnSensitive	O	X	O
Part D — Mixed			
Intent_RPC_Comprehensive	X	X******	O
Sum, Precision and Recall — ICC-Bench			
O, higher is better	28	1	31
*, lower is better	1	9	1
X, lower is better	3	30	0
Precision p = O/(O + *)	97%	10%	97%
Recall r = O/(O + X)	90%	3%	100%
F-measure 2pr/(p + r)	93%	5%	98%

DroidBench [16] is a benchmark testsuite published by the FlowDroid team, which consists of Android apps for evaluating information-flow analysis. The version we used contains 21 apps, including inter-component communication challenges as well as inter-app communication challenges. ICC-Bench [30] contains 24 apps for testing various Intent communication, RPC communication, static fields tracking capabilities as well as multi-app analysis capabilities. The test apps in ICC-Bench are categorized in four parts each of which focuses on one type of ICC: Part A involves various types of intent handling: explicit intent target finding, implicit intent target finding (via matching action, categories, data and type), and dynamically registered component handling, *etc.*; Part B focuses on the accuracy of the analysis by including a variety of scenarios where certain Intent-related information flow paths do or do not exist, and the capability to handle `IntentService`[7] and `Stateful ICC`; Part C tests the ability of handling different types of RPC communications; Part D contains one comprehensive test case to test whether the tool can handle complex scenarios where data may flow via various communication channels. ICC-Bench is designed by us and publicly available [30]. The apps in these testsuites are not crafted to favor a particular tool. They represent common scenarios one will find when reasoning about the relevant security issues.

We run each tool on each test app to check if the tool can report the correct data leak paths. The detailed comparison of the performance of IccTA, DroidSafe and Amandroid on DroidBench and ICC-Bench is available in Table 2. The results are shown in terms of True Positive (O), False Positive (*), and False Negative (X), if any. If a test app contains multiple data leak paths, the result is shown for each of them. As an example, in Table 2 for `ActivityCommunication2` app of DroidBench, both IccTA and Amandroid have entry "OO*", which indicate that these tools detect two paths (*i.e.*, OO) but also report one false path (*i.e.*, *). We observe that Amandroid outperforms IccTA and DroidSafe on both benchmarks. The sole false negative of Amandroid for Droid-Bench is due to Amandroid not modeling Java `Singleton`. The false positives of Amandroid on both benchmarks are due to context-insensitive inter-component data flow handling and the rudimentary string analysis.

Although IccTA's website claims that the tool is capable of performing inter-app analysis by combining multiple apks into a single apk, in our experience their `ApkCombiner` failed to combine the inter-app communication apps in DroidBench. Thus we could not obtain any result from IccTA on the inter-app communication experiment for Droid-Bench. Moreover, the ICC-Bench apps have all been updated to the newest Android version (Android 7.1.1), representing the current Android application design with the new permission acquiring mechanism introduced by Android M and later versions. Neither IccTA nor Amandroid had problem of detecting data leaks in the new version of apps after we manually updated some of their dependency libraries and Android sdk. However, DroidSafe could not handle the new design even after we updated the dependency libraries and Android SDK, and that is the reason DroidSafe is shown to be missing so many paths over ICC-Bench testsuite.

6.3 RQ3: How effective is Amandroid's Intent resolution for real-world apps?

We evaluate Amandroid on all 4,600 real-world apps in our dataset, to calculate the precision of the Intent resolution. As Table 3 indicates, in GPlay 21,062 ICC calls require Intent resolution, and Amandroid is able to infer precise Intent string values for 17,354 (89%) of them. In MAL apps, 18,749 ICC calls require Intent resolution, and Amandroid is able to infer precise Intent string values for 13,883 (80%) of them. Overall 85% of the ICC cases can be precisely resolved, showing that Amandroid's Intent resolution is capable of handling real-world apps and will not introduce too much over-approximation for ICC data flows.

[7] `IntentService` is a special `Service`, which receives an Intent and executes the corresponding operation in background.

Table 3. Intent resolution precision.

Dataset	GPlay	MAL	Total
Total Intent invocation point	21,062	17,354	38,416
Precise Intent resolution	18,749	13,883	32,632
Precision	89%	80%	85%

6.4 RQ4: Is Amandroid capable of discovering crucial security issues to aid in real-world app vetting?

Amandroid is a highly extensible framework that allows analysts to write customized security checkers as plugins on top of it. To evaluate Amandroid for real-world app vetting, we wrote five security checkers (where each checker detects a particular security problem) and apply them on the GPlay and MAL dataset.

The security checkers are listed below: (1) Hiding-Icon Checker, (2) Crypto Library Misconfiguration Checker, (3) SSL/TLS Misconfiguration Checker, (4) Data Leakage Checker, (5) Intent Injection Checker.

Checkers (1), (3), (4) are new and not reported before. (2), (5) were first reported in the original Amandroid paper [62] and have been substantially extended since then.

```
..ComponentName v0_3 = this.getComponentName();¶
..Context v1_1 = this.getApplicationContext();¶
..PackageManager v1_2 = v1_1.getPackageManager();¶
..v1_2.setComponentEnabledSetting(v0_3, ¶
....PackageManager.COMPONENT_ENABLED_STATE_DISABLED, ¶
....PackageManager.DONT_KILL_APP);¶
```

Fig. 7. Hiding-Icon code snippet.

6.4.1 Hiding-Icon Checker. Hiding-icon is one common malware scheme to hide the application's physical existence on the phone. In particular, it hides the malware app's launcher icon while making the malware's background service run. To do this, the app needs to disable its main component while telling the android system not to kill its background service by calling an API `Context.setComponentEnabledSetting()` with specific parameters as shown in Figure 7.

The idea of detecting such suspicious behavior is to extract from *DFG* the values passed to the `Context.setComponentEnabledSetting()` API and match them with the malformed parameters (as shown in Figure 7). Applying this checker to the app dataset, we found 4 GPlay apps and 75 MAL apps having this suspicious behavior.

6.4.2 Crypto Library Misconfiguration Checker. We implemented a plugin to check whether an app conforms to the following crypto API configuration rules [18]:

Rule 1: Do not use ECB mode for encryption.
Rule 2: Do not use a non-random IV for CBC encryption.

Rule 1 is to evaluate the string value used to create the `javax.crypto.Cipher` instance. If the string value indicates that the cipher will run in ECB mode, the checker will report an alarm. To check **Rule 2**, the checker first detects the cipher is using the CBC mode, and then checks the IV creation process to see whether a constant IV is used. Table 4 summarizes the results we obtained through running the above checker on the app dataset.

6.4.3 SSL/TLS Misconfiguration Checker. SSL/TLS protocols are widely adopted in Android applications to provide secure data transmission between the client app and their backend server. App developers may not be properly trained for correctly using SSL/TLS library and there is a lack

Table 4. Crypto Library Misconfiguration Checker Report.

Dataset	GPlay	MAL
#apps using ECB mode	438	303
#apps using non-random IV	210	87

of visual security indicators for SSL/TLS usage in the development environment (IDE). As a result SSL/TLS library APIs can be easily misconfigured [21, 51].

One common misuse case is allowing all hostnames for the SSL/TLS's `HostnameVerifier`, by invoking `SSLSocketFactory.setHostnameVerifier()` with parameter *ALLOW_ALL_HOSTNAME_VERIFIER*. To capture this, the checker will evaluate whether the parameter passed to `SSLSocketFactory.setHostnameVerifier()` is equal to *ALLOW_ALL_HOSTNAME_VERIFIER*.

Another misuse case is accepting all certificates or accepting all hostnames for a certificate as long as a trusted CA signed the certificate, by providing their own or third-party-implemented `TrustManager` and `SocketFactory`. [21] provides a list of problematic `TrustManager` and `SocketFactory` implementations with its class names, which our checker plugin searches for in a given app. Table 5 summarizes the results we obtained through running the above checker on the app dataset.

Table 5. SSL/TLS Misconfiguration Checker Report.

Dataset	GPlay	MAL
# apps with Bad TrustManager	63	18
# apps with Bad SSLSocketFactory	37	13
# apps with Bad SSL hostname configuration	288	192

6.4.4 Data Leakage Checker. Phone call logs, contacts, and SMS messages are a few examples of user's sensitive information which should be kept private. Amandroid can be used to check whether an app obeys the above data usage policy. We apply simple strategies to identify the various communication data sources. Basically, Amandroid tracks the corresponding (*i.e.*, tied with the data source) string literals or BroadcastReceivers: (1) Call logs: "content://call_log/calls"; (2) Sim card contacts: "content://icc/adn"; (3) Phone contacts: "com.android.contacts"; (4) SMS: "content://sms/inbox/" and input for BroadcastReceivers handling the "SMS_RECEIVED" event.

On the other hand, the sinks are any outgoing communication channel, such as http/https write, SMS send, implicit Intent send, *etc.* We found several potential sensitive data leakage cases, some of which are shown in Table 6.

Table 6. Data Leakage Checker Report.

App Name	Dataset	Description
com.skymoons.hqg.anzhi.apk	GPlay	Read user's SMS inbox, write into log, then send text message to the senders.
12050f267d5e8ce6f77d2111cd3043f0.apk	MAL	Read user's SMS inbox, store in a JSON object, write into SharedPrefeferences, then upload to its C&C server.
5339a0e7e86ac1f5472f832874426c25.apk	MAL	Upload user's SMS content and information to its C&C server.
51bf3112982473e99b88965f6e271799.apk	MAL	Read user's SMS inbox, upload to its C&C server, send text message to senders.

6.4.5 Intent Injection Checker. Intent is one of the most common ways for an Android component to receive and process data from outside. If an app makes wrong assumptions for the incoming intent and performs sensitive operations based on it, that may result in serious security holes [37, 60].

To detect the above issue, in Amandroid we mark the intent receiving point as the source, and sensitive operations (*e.g.*, open URL connection, crafting another intent, *etc.*) as sink. We then query the *DDG* to find whether there is a data dependence path between them. We found several potential intent injection cases, some of which are shown in Table 7.

Table 7. Intent Injection Checker Report.

App Name	Dataset	Description
com.qryptal.verifydetailsauthenticate.android.apk	GPlay	Allows any app inject URL to its ShareActivity, which will then encode it to a Barcode and display to the user. If user scan the Barcode, they might be redirected to malicious websites.
com.freegame.basketball.apk	GPlay	Allows any app inject data into its SharedReference, which will disable this app's functionality.
com.mmmono.mono.apk	GPlay	Allows any app send commands to start/stop its service's heartbeat and connectivity status.
com.bigfishgames.dmddgoogfree.apk	GPlay	Allows any app send commands to launch arbitrary URL and components.

6.5 RQ5: How much effort does it take to build a new analysis on top of Amandroid core framework?

The advantage of Amandroid's approach is that the general framework provides a means for building a variety of further security analyses in a straightforward and easy way. Each special analysis built on top of Amandroid involves developing a "Checker plugin" that leverages the *DFG*s and *DDG*s from Amandroid's analysis. Moreover, once the core analysis produces *DFG*s and *DDG*s for an app, they can be stored and reused in multiple security analyses. We present the summary of the plugins used in the above applications in Table 8, which shows their size in Scala LOC, as well as the average running time. This can be compared with the size of the core engine and its average running time, shown in the last row of the table.

Table 8. Code Size and Running Time (Checkers and Core)

Name	Approx. Size (Scala LOC)	Avg. Time
Hiding-Icon Checker	40	50ms
Crypto Library Misconfiguration Checker	109	50ms
SSL/TLS Misconfiguration Checker	62	20ms
Data Leakage Checker	73	50ms
Intent Injection Checker	23	100ms
Core Framework	46,345	440s

7 DISCUSSIONS

Currently Amandroid only does constant propagation for string values, and has a conservative model for string operations. This limitation causes 15% imprecision on the Intent resolution in our experiment (section 6.3). Precise and general string analysis in static analysis is nontrivial and we leave this for future research. For example, we could follow the approaches from prior research [13, 33]. Moreover, recent work [41] shows that ICC resolution can benefit from domain knowledge and probabilistic models, which we could adopt to prioritize inferred ICC destination choices.

Amandroid does not currently handle Java reflection, dynamic class loading, and native method invocation. Adding preliminary support for reflections and dynamic class loading is similar to handling ICC in Amandroid. Moreover, [35] has proposed ways to handle Java Reflection and dynamic class loading in a reliable way, which we might be able to leverage in the future. Tracking data flow through Java Native Interface (JNI) is out of scope for this work.

Amandroid can capture the concurrent execution semantics at the inter-component level as discussed in section 4.3. However, Android also allows general thread-based concurrent execution within a component. This is not currently handled by Amandroid. We leave it as future work to fully account for all possible concurrent executions, by leveraging existing work such as Indus [17].

Amandroid's data and control flow analysis depends on the faithfulness of the models, including the models of the Android environment and its APIs. We designed a DSL[8] to let researchers more easily model library APIs for their analysis purpose. Amandroid currently provides more than 1,000 API models using the DSL, which covers a large portion of the real world Android API usage. However, due to the size of the Android library and complexity of third party libraries, it remains a challenge to reliably detect all library API and provide precise and sound model for them. Recent work [9, 36, 38] offer approaches to detect third party libraries. Prior work [5] shows that precise data flow summary for Android framework can be computed by static analysis. We might be able to leverage them in Amandroid in the future.

8 RELATED WORK

There has been a long line of works on applying static analysis for Android security problems [6, 12, 18, 21, 24, 26, 28, 31, 34, 37, 42, 43, 56]. Below we describe a few works that are most closely related to ours.

The design of Amandroid leverages a number of approaches from FlowDroid [6, 24] (*e.g.*, callback collection algorithm during environment generation), but the two also have a few important differences. FlowDroid does not handle ICC and as such cannot address security issues involving intent passing among multiple components. FlowDroid builds a call graph based on Spark/Soot [57], which conducts a flow-insensitive points-to analysis. FlowDroid then conducts a taint and on-demand alias analysis based on the above call graph, using IFDS [48, 49] which is flow- and context-sensitive. The flow-insensitivity in the call graph construction may introduce spurious call edges (false positives), which could impact the analysis precision of the subsequent IFDS analysis. Amandroid computes the call graph at the same time as the dataflow analysis by computing the flow- and context-sensitive points-to facts; thus its callgraph is more precise, which could lead to fewer false positives in the final analysis results. Moreover, FlowDroid does not calculate alias or points-to information for *all* objects in a both context- and flow-sensitive way. This is a design decision from computing cost concerns [24]. Amandroid calculates all objects' points-to information in a both context- and flow-sensitive way, with reasonable computing cost (ref. Section 6.1). This enables us to build the generic framework supporting multiple security analyses.

Epicc [43] computes Android Intent call parameters using the same IDE framework as FlowDroid, by modeling the intent data structure explicitly in the flow functions. To the best of our knowledge, Epicc does not use the Intent parameter analysis result to resolve the Intent call targets in the general case, and has not used the result to perform inter-component dataflow analysis. Amandroid's approach to deriving Intent parameters is to simply use the flow and context-sensitive points-to information (including that for string objects) already computed in the *DFG*, without the need for a separate data flow analysis just for Intent. Amandroid also uses the Intent call parameter information to link Intent call sites to call targets, resulting in an *DFG* that includes data flow paths both within and across components. Moreover, recent work [41] shows that domain knowledge and probabilistic models can be leveraged in Intent destination resolution, which Amandroid could adopt.

Recently, IccTA [34] and DroidSafe [26] made advancement in the state-of-the-art of Android app static analysis. IccTA extends FlowDroid and uses IC3 [42] as the Intent resolution engine, which can now track data flows through regular Intent calls and returns. However, IccTA is yet to track the information flow through remote procedure call (RPC). DroidSafe [26] tracks both Intent and RPC calls, but does not capture data flows through "stateful ICC" nor inter-app analysis.

[8]Grammar of the DSL can be found at https://github.com/arguslab/Argus-SAF/blob/master/org.argus.jawa/src/main/java/org/argus/jawa/summary/grammar/Safsu.g4

Recent works [31, 56] explored various approaches to tracking Android inter-app communication for security vetting. The new component-based analysis of Amandroid as discussed in this paper supports inter-app analysis naturally (see section 4.5).

Lu *et al.* [37] uses a static-analysis scheme called CHEX to detect *component hijacking* problem in Android, which is reduced to finding information flows. CHEX first constructs *app-splits*, each of which is a code segment reachable from an entry point. It then computes the data-flow summary for each split using Wala [23]. The split summaries are linked in all permutations that do not violate the Android system call sequences and could result in transitive information flow. Amandroid computes information flow in a different way – through the usage of an environment method for each component that calls the relevant callbacks in the right order (per Android system specification), and by building the *DFG* and *DDG* for the complete app. CHEX does not have the provision to track data flow through the ICC channels, which Amandroid does.

Chin *et al.* [12] first systematically studied the attack surface related to Intent. In particular, they identified problems such as *unauthorized intent receipt* and *intent spoofing*. They also developed a static analysis tool which can raise warnings for the above problems in an over-conservative manner. Their tool ComDroid performs flow-sensitive, intra-procedural static analysis, and the paper states that there is a limited inter-procedural analysis that "follows method invocations to a depth of one method call." Amandroid performs a full-fledged inter-procedural data-flow analysis in a flow- and context-sensitive way, and also tracks the data flows over the ICC channels. While we would like to conduct comparison study between ComDroid and Amandroid, the link to the ComDroid tool (used to be http://www.comdroid.org) is no longer there.

There has been a large body of work reporting Android app security issues [66, 67], some of which use static analysis techniques [18, 21, 25, 27]. Those works focus on finding specific security problems, and the static analyses used do not seem to address some key issues such as the inter-component nature of Android app's execution and the precise modeling of Android's callback sequences. In contrast, Amandroid is a precise and general inter-component static analysis framework which can address a large range of security issues in Android apps.

Multiple prior works [15, 44, 64] investigated the root security problems in the Android system and proposed augmented infrastructures to enforce the given security policy. Recently, SEAndroid [50] has been proposed which enforces Mandatory Access Control (MAC) both in the kernel layer and in the middleware. This system provides a better mechanism for sand-boxing the apps. However, MAC will not stop the security problems which happen within an app or through the legitimate ICC channels. In this paper, we assume the sand-boxing (and isolation) of apps by the Android system is not compromised; thus, our approach is complementary to those prior works.

TaintDroid [19] is a dynamic (runtime) taint-tracking and analysis system to find potential misuse of the user's private information. All dynamic analyses are subject to evasion attacks. For example, researchers have shown [45] that Google's Bouncer [11] can be fingerprinted and hence evaded by a well-crafted app. On the other hand, static analysis investigates the code of the app (along with the app's manifest, *etc.*), which determines the runtime behaviors of the app; this makes it attractive for security vetting. Recently Sounthiraraj *et al.* [51] showed that static and dynamic analysis can be combined to achieve more effective detection/confirmation of security problems. Our approach provides a precise and general static analysis framework that can complement dynamic analyses.

9 CONCLUSIONS

In this paper we presented Amandroid – a general static analysis framework that can be used for security vetting of Android applications. In particular, Amandroid can precisely track the control and data flow of an app across multiple components, and can compute an abstraction

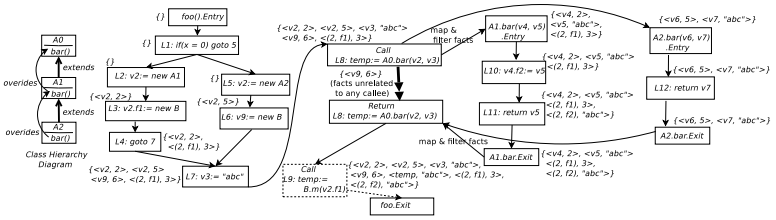


Fig. 8. Building the *DFG* for *foo*: The intra-procedural control flow graph (*CFG*) of *foo* is extended to a callee, *bar*.

of the app's behavior in the forms of data-flow graph and data dependence graph. As a general framework, Amandroid can be easily extended to achieve a number of specialized security analyses. Our experiment results showed that Amandroid scales well. We also demonstated that Amandroid can be readily applied to effectively address multiple specialized security problems. Our experiment results showed that Amandroid out-performs existing static analysis tools for Android apps.

APPENDIX

The Basic DFG Building Process.

A static analyzer simulates the program and keeps track of the fact sets, until a fixed point is reached. The convergence to a fixed point (analysis termination) is guaranteed as long as the flow equations are monotone, and the number of facts is finite, which hold for Amandroid's analysis. For a given app, it contains a finite number of object creation sites and variables/fields (and as typically done, elements of an array are summarized as one); moreover, we keep tracks of calling contexts up to a finite number k.

Amandroid builds the *DFG* by flowing the points-to facts from the program's entry points. Here the program is the IR of the app's dex code augmented with the environment methods as discussed in Section 3.2. Unlike Java applications, there is no "main" method in an Android app; every component could be the starting point of an app. Our component-based environment model captures the full life cycle of a component and all of its possible execution paths, including those due to interacting with other components. Thus, if we assume one particular execution path starts from component C, we can use C's environment method E_C as the program's entry point. To include all possible execution paths from all possible components, we do this for every component in the app, yielding multiple *DFG*s. Formally, let C be a component, the *DFG* from C is denoted $DFG(E_C)$ where E_C is the environment method of C, and is a tuple defined as the following.

$$DFG(E_C) \equiv ((N, E), \{entry(n) | \ n \in N\}),$$

where N and E are the nodes and edges of the inter-procedural control flow graph starting from E_C (denoted $ICFG(E_C)$). $entry(n)$ is the entry set of the statement associated with node n. Each $DFG(E_C)$ captures the execution that starts from component C, and may involve other components due to ICC. Each statement node is annotated with the statement entry set (the exit set is not shown for presentation sake). In this example, Amandroid starts building the *DFG* from the entry point method `foo` with an empty fact set. Amandroid then simulates the program statically based on each statement's semantics and transforms the fact sets along the way based on the flow equation (1).

Figure 8 illustrate one example. At a control-flow join point, the exit fact sets from all incoming edges are unioned (*e.g.*, at *L7*); facts such as $\langle v2, 2\rangle$ and $\langle v2, 5\rangle$ coming from the different branches accumulate in $entry(7)$. Similarly, one can compute $entry(8)$. At this point, Amandroid needs to

Algorithm 2 Building Data Flow Graph (*DFG*)

Require: The entry point procedure, *EP*.
Ensure: *DFG*(*EP*)

procedure BuildDfg(*EP*)
  icfg ≡ (*N*, *E*) ← *empty graph*;
  addCFG(*icfg*, *CFG*(*EP*));
  ι ← *initial fact set*;
  entry ← *emptyMap*;
  worklist ← *emptyList*;
  entry (*EntryNode*$_{EP}$) ← ι;
  worklist ← *worklist* :: *EntryNode*$_{EP}$;
  while $worklist \neq empty$ **do**
    n ← get (and deque) *head* from *worklist*;
    nodes ← *processNode*(*icfg*, *n*);
    worklist ← *worklist* ::: *nodes*;
  return (*icfg*, *entry*);

Algorithm 3 *processNode:* Pushing facts to *successors*

Require: *ICFG*, *icfg* ≡ (*N*, *E*) and a *node*, *n* ∈ *N*
Ensure: *n*'s successor nodes whose *entry* are updated.

procedure processNode(*icfg*, *n*)
  tempList ← $empty$;
  if *n* is an *EntryNode* or a *ReturnNode* **then**
    for all *p* ∈ *successors*(*n*) **do**
      entry(*p*) ← *entry*(*p*) ∪ *entry*(*n*);
      tempList ← *tempList* :: *p*;
  else if *n* is an *ExitNode* **then**
    for all *p* ∈ *successors*(*n*) **do**
      passRequiredFactsToCaller(*n*, *p*);
      if *p* gets any new fact **then**
        tempList ← *tempList* :: *p*;
  else if *n* is a *CallNode* or a *RegularNode* **then**
    if *visit*(*icfg*, *n*) = *true* **then**
      tempList ← *tempList* ::: *successors*(*n*);
  return *tempList*;
procedure visit(*icfg*, *n*)
  if *n* is a *CallNode* **then**
    (*fMapForCs*, *factsToR*) ← *reslvCall*(*icfg*, *n*);
    update callees' *EntryNode*s with *fMapForCs*;
    update *ReturnNode*(*n*) with *factsToR*;
  else if *n* is an *RegularNode* **then**
    for all *p* ∈ *successors*(*n*) **do**
      entry(*p*) ← *entry*(*p*) ∪ *exit*(*n*);
  if any *p* ∈ *successors*(*n*) gets any new fact **then**
    return $true$;
  return $false$;
procedure reslvCall(*icfg*, *n*) ▷ *n* is a *CallNode*
  calleeSet ← *getCallees*(*entry*(*n*), *callSig*(*n*));
  for all *M* ∈ *calleeSet* **do**
    if (*EntryNode*$_M$ ∉ *N*) **then**
      addCFG(*icfg*, *CFG*(*M*));
      E ← *E* ∪ (*n*, *EntryNode*$_M$);
      E ← *E* ∪ (*ExitNode*$_M$, *ReturnNode*(*n*));
  fToCallees ← *empty*;
  factsMapForCallees ← *emptyMap*;
  for all *p* ∈ *successors*(*n*) **do**
    factsToCallee ← *filterFunc*(*n*, *p*, *entry*(*n*));
    factsMapForCallees(*p*) ← *factsToCallee*;
    fToCallees ← *fToCallees* ∪ *factsToCallee*;
  factsToReturn ← *exit*(*n*) \ *fToCallees*;
  return (*factsMapForCallees*, *factsToReturn*);

resolve the target for *L8*'s virtual method invocation with static type `A0`. The first argument of the call instruction, `v2`, is the `receiver` object. Since we now have calculated the possible points-to values of `v2` — *instance 2* or *instance 5*, we can resolve the possible call targets precisely: `A1.bar` for *instance 2* and `A2.bar` for *instance 5* (because both `A1` and `A2` override `A0.bar`). This shows the advantage of doing a precise points-to analysis concurrently with *ICFG* building — not only can we have more precise information on the call targets, but also it allows us to flow more accurate facts to the different call targets. All of these increase the precision and can potentially reduce the number of false alarms in the analysis results.

As shown in Figure 8, a call statement contributes a pair of *CallNode* and *ReturnNode* to the *ICFG*. The *CallNode* connects to the callee's *EntryNode* while the callee's *ExitNode* connects to the *ReturnNode*. In transferring facts between the caller and the callee, the variable-facts need to be remapped to the formal parameters of the callee (*e.g.*, `v2` in the caller maps to `v4` in the callee). This should be restored when the control returns to the caller. Only heap-facts reachable from the call parameters are passed to the callee. The unreachable heap-facts as well as unrelated variable-facts are transferred to the *ReturnNode* directly to improve efficiency. In the example of *L8*'s method invocation, there is one variable-fact ⟨*v9, 6*⟩ which is unrelated to both arguments `v2` and `v3`. The flow of such fact (which is unrelated to any callee) is represented as a double-head arrow from the *CallNode* to the *ReturnNode*. Similarly, there can be some facts at the callee side that are unrelated to the caller (*e.g.*, callee's local variables and temporary objects), and we filter them out at the callee's *ExitNode* to improve efficiency.

Consider the dataflow analysis for `A1.bar` or `A2.bar`, which is a callee for *L8*'s method invocation. Amandroid tracks the *entry* of each statement of `A1.bar` (or `A2.bar`). We observe that *entry*(`Return 8`) contains heap-facts which show that field `f2` of `Instance` *2* points to the String "abc". This is the effect of *L10*. It is interesting to see that this is not true for the same field (*i.e.*, `f2`) of `Instance` *5* because no assignment like *L10* happens inside `A2.bar`.

Now, we can get *entry*(9), and continue to process the next call similarly. The process is similar to what we did for *L8*, except that we have to handle the possibility of a *null* receiver (because there is no fact associated with `v2.f1` for ⟨*v2, 5*⟩). For a virtual method statement, if the facts show that the receiver variable maybe *null*, then we do not process this particular instance; instead, we only propagate the non-null receiver instances (if any) to the callee and flag the call site as a possible runtime error.

Algorithm for Building DFG. The algorithm for the *DFG* building process is formally presented as Algorithm 2. This is a fixed-point algorithm (ref. the while loop from *L9* to *L13*), which tracks what *points-to facts* reach each statement from the given entry point (*EP*). The core of Algorithm 2 is *L11*, which processes different type of nodes in the control flow graph, and this is formally elaborated in Algorithm 3. Algorithm 3 presents how to process each type of node (*e.g.*, *CallNode*, *ReturnNode*, *etc.*). As an example, if it's a *CallNode*, the *ICFG* will be expanded by including the callee graph based on the points-to facts flowing there.

ACKNOWLEDGMENTS

We express our gratitude to Venkatesh Prasad Ranganath for contributing many ideas to this work. This research was partially supported by the U.S. National Science Foundation under grant no. 0644288, 0954138, 1018703, 1717862, and 1718214, and the U.S. Air Force Office of Scientific Research under award no. FA9550-09-1-0138. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the above agencies.

REFERENCES

[1] Android documentation: Intent and Intent Filter. http://developer.android.com/guide/components/intents-filters.html.
[2] akka. 2016. Actors. http://wala.sourceforge.net/wiki/index.php/UserGuide:CallGraph. (2016).
[3] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the Mining Software Repositories (MSR)*.
[4] Andrew W. Appel. 1998. *Modern Compiler Implementation in Java.* Cambridge University Press.
[5] Steven Arzt and Eric Bodden. 2016. StubDroid: automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the IEEE ICSE*. 725–735.
[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the ACM PLDI*.
[7] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android permission specification. In *Proceedings of the ACM CCS*.
[8] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2015. Mining Apps for Abnormal Usage of Sensitive Data. In *Proceedings of the IEEE ICSE*.
[9] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in Android and its security applications. In *Proceedings of the ACM CCS*.
[10] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving Apps to Test the Security of Third-party Components. In *Proceedings of the 23rd USENIX Conference on Security Symposium*. 1021–1036.
[11] Google Bouncer. 2012. http://googlemobile.blogspot.com/2012/02/android-and-security.html. (2012).
[12] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. Analyzing inter-application communication in Android. In *Proceedings of the ACM Mobisys*.
[13] Aske Christensen, Anders Møller, and Michael Schwartzbach. 2003. Precise analysis of string expressions. *Static Analysis* (2003), 1076–1076.
[14] Cisco. 2014. Cisco 2014 Annual Security Report. http://www.cisco.com/web/offer/gist_ty2_asset/Cisco_2014_ASR.pdf.
[15] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich. 2012. CRePE: A System for Enforcing Fine-Grained Context-Related Policies on Android. *Information Forensics and Security, IEEE Transactions on* 7, 5 (2012), 1426–1438.
[16] DroidBench. 2015. https://github.com/secure-software-engineering/DroidBench.
[17] Matthew B Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Ranganath, Robby, and Todd Wallentine. 2006. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Proceedings of the TACAS*.
[18] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the ACM CCS*.
[19] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones.. In *Proceedings of the USENIX OSDI*.
[20] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: An information flow tracking system for real-time privacy monitoring on smartphones. *Commun. ACM* 57, 3 (2014), 99–106.
[21] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the ACM CCS*.
[22] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. 2011. A survey of mobile malware in the wild. In *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*.
[23] Stephen Fink and Julian Dolby. 2012. WALA–The TJ Watson Libraries for Analysis. http://wala.sf.net/.
[24] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. 2013. *Highly Precise Taint Analysis for Android Application.* Technical Report. EC SPRIDE.
[25] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. 2012. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proceedings of the International Conference on Trust and Trustworthy Computing*.
[26] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the NDSS*. Citeseer.
[27] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. 2012. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the NDSS*.
[28] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad Reza Sadeghi. 2012. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks*.

[29] John Hatcliff, Patrice Chalin, Jason Belt, et al. 2013. Explicating symbolic execution (xSymExe): An evidence-based verification framework. In *Proceedings of the IEEE ICSE*. 222–231.
[30] ICC-Bench. 2017. https://github.com/fgwei/ICC-Bench.
[31] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. 2014. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. 1–6.
[32] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java Points-to Analysis Using Spark. In *Proceedings of the Compiler Construction*.
[33] Ding Li, Yingjun Lyu, Mian Wan, and William GJ Halfond. 2015. String analysis for Java and Android applications. In *Proceedings of the ACM FSE*. 661–672.
[34] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the IEEE ICSE*.
[35] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the ACM ISSTA*.
[36] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. Libd: Scalable and precise third-party library detection in Android markets. In *Proceedings of the IEEE ICSE*.
[37] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. 2012. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the ACM CCS*.
[38] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: fast and accurate detection of third-party libraries in Android apps. In *Proceedings of the IEEE ICSE*.
[39] McAfee. 2017. Trojans, Ghosts, and More Mean Bumps Ahead for Mobile and Connected Things. https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2017.pdf.
[40] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 1999. *Principles of program analysis*. Springer.
[41] Damien Octeau, Somesh Jha, Matthew Dering, Patrick McDaniel, Alexandre Bartel, Li Li, Jacques Klein, and Yves Le Traon. 2016. Combining static analysis with probabilistic models to enable market-scale android inter-component analysis. In *Proceedings of the ACM POPL*, Vol. 51. 469–484.
[42] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite Constant Propagation: Application to Android Inter-Component Communication Analysis. In *Proceedings of the IEEE ICSE*.
[43] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-component Communication mapping in Android with Epicc: An Essential Step towards Holistic Security Analysis. In *Proceedings of the USENIX Security Symposium*.
[44] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. 2012. Semantically rich application-centric security in Android. *Security and Communication Networks* 5, 6 (2012), 658–673.
[45] Nicholas J Percoco and Sean Schulte. 2012. Adventures in Bouncerland. *Black Hat USA* (2012).
[46] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2014. Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications. In *Proceedings of the NDSS*. 23–26.
[47] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*.
[48] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM POPL*.
[49] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1 (1996), 131–170.
[50] Stephen Smalley and Robert Craig. 2013. Security enhanced (SE) Android: Bringing flexible MAC to Android. In *Proceedings of the NDSS*.
[51] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. 2014. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the NDSS*.
[52] Symantec. 2017. Internet Security Threat Report. https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf.
[53] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the NDSS*.
[54] Hariharan Thiagarajan, John Hatcliff, Jason Belt, et al. 2012. Bakar Alir: Supporting Developers in Construction of Information Flow Contracts in SPARK. In *Proceedings of the IEEE SCAM*. 132–137.
[55] TrendMicro. 2017. In Review: 2016's Mobile Threat Landscape Brings Diversity, Scale, and Scope. https://blog.trendmicro.com/trendlabs-security-intelligence/2016-mobile-threat-landscape/.

[56] Yutaka Tsutano, Shakthi Bachala, Witawas Srisa-an, Gregg Rothermel, and Jackson Dinh. 2017. An efficient, robust, and scalable approach for analyzing interacting android apps. In *Proceedings of the IEEE ICSE*. 324–334.
[57] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java bytecode using the Soot framework: Is it feasible?. In *Proceedings of the Compiler Construction*.
[58] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. 2014. A5: Automated Analysis of Adversarial Android Applications. In *Proceedings of the SPSM*. 39–50.
[59] WALA. 2014. WALA documentation: CallGraph. (2014).
[60] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. 2013. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the ACM CCS*.
[61] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. 2017. Deep Ground Truth Analysis of Current Android Malware. In *Proceedings of the DIMVA*. Springer, Bonn, Germany.
[62] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the ACM CCS*. Scottsdale, AZ.
[63] Wikipedia. 2016. Actor model. https://en.wikipedia.org/wiki/Actor_model. (2016).
[64] Rubin Xu, Hassen Saïdi, and Ross Anderson. 2012. Aurasium: Practical policy enforcement for Android applications. In *Proceedings of the USENIX Security Symposium*.
[65] Lok-Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis.. In *Proceedings of the USENIX Security Symposium*. 569–584.
[66] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android malware: Characterization and evolution. In *Proceedings of the IEEE SP*.
[67] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the NDSS*.