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

Fengguo Wei, Sankardas Roy, Xinming Ou, Robby
Department of Computing and Information Sciences
Kansas State University
{fgwei,sroy,xou,robby}@ksu.edu

ABSTRACT

We propose a new approach to conduct static analysis for security vetting of Android apps, and built a general framework, called Amandroid for determining points-to information for *all* objects in an Android app in a flow and context-sensitive way across Android apps components. We show that: (a) this type of comprehensive analysis is completely feasible in terms of computing resources needed with modern hardware, (b) 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.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; K.6 [**Management of Computing and Information Systems**]: Security and Protection

General Terms

Static Analysis; Mobile Security

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2660267.2660357.

Keywords

Android Application; ICC (Inter-component Communication); points-to analysis; information leakage; vulnerable app; malware; security vetting

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 [15, 19, 34, 36, 37]. The current solutions to those security problems are mostly reactive (*e.g.*, pulling an app off the market after potential damage may have already been done). There have not been effective vetting methods that market operators can rely upon to ensure apps entering a market (*e.g.*, Google Play) are free of certain types of security problems. Often times, they have to resort to dynamic analysis — running an app in a testing environment with the hope of identifying the problematic behaviors, if any, during the test run (*e.g.*, Google Bouncer [3]).

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 [8, 11, 14, 17, 20, 23, 25]. 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 applying 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 precision, it may also be prohibitively expensive.
3. Android is a component-based system and makes extensive use of inter-component communication (ICC). A component can send an *intent* to another component. The target of an ICC 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, 17] formally models the event-driven life cycle of an Android app in a "dummyMain" method, but it does not address ICC. Epicc [25] statically analyzes ICC and uses an IDE [29] framework to solve for ICC call parameters, but does not link the ICC call sources to targets and does not perform dataflow analysis across component-boundaries. CHEX [23] 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 ICC. These prior works have all inspired this work. We designed and built **Amandroid**[1] – an inter-component 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 the whole app, that is both flow and context sensitive [24]. This is a side benefit of our approach compared to prior works that have adopted existing static analysis frameworks (*e.g.*, Soot [32] and Wala [16]), which build *ICFG* with less precision [4, 22].
3. Amandroid's *ICFG* includes inter-component communication (ICC) edges. That is, Amandroid treats ICC just like method calls, and both control and data can flow on the edges. Amandroid is able to conduct an elementary string analysis (due to its object-sensitivity) for inferring ICC call parameters, and links the ICC source to the call targets based on a flow/context-sensitive matching algorithm. Amandroid models the Android environment for both control and data, so that important intent data flows can be captured according to inherent Android properties. We call Amandroid's *ICFG* together with each node's reaching fact set as Inter-component Data Flow Graph (*IDFG*).
4. Amandroid builds the data dependence graph (*DDG*) of the app from the *IDFG*. 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 *DDG* and *IDFG*.

We evaluated Amandroid on hundreds of real-world apps (753 Google Play apps shared by the Epicc group, and 100 potentially malicious apps from Arbor Networks). Our experimental results show that Amandroid scales well. We used Amandroid to address security problems such as password leakage, OAuth token leakage, intent injection, and misuse of crypto APIs. The core framework of Amandroid takes tens of seconds to analyze one app on average. All the specialized analyses require very little additional coding effort (around 100 LOC) to leverage Amandroid's *IDFG* and *DDG* to address the specific problem, and the additional running time is negligible (typically in the order of tens of miliseconds).

We then experimentally compare Amandroid with two static analyzers for Android: FlowDroid [6, 17] and Epicc [25], and show that Amandroid can address a wide range of security problems due to inter-component communications in Android that cannot be handled by these existing tools. Amandroid also found multiple crucial security problems in Android apps that were never reported before in the literature.

The rest of the paper is organized as follows. Section 2 gives a motivating example. Section 3 describes in detail Amandroid's analysis methods. We discuss experimentation of our approach in Section 4, limitations of Amandroid in Section 5, and related research in Section 6.

2. A MOTIVATING EXAMPLE

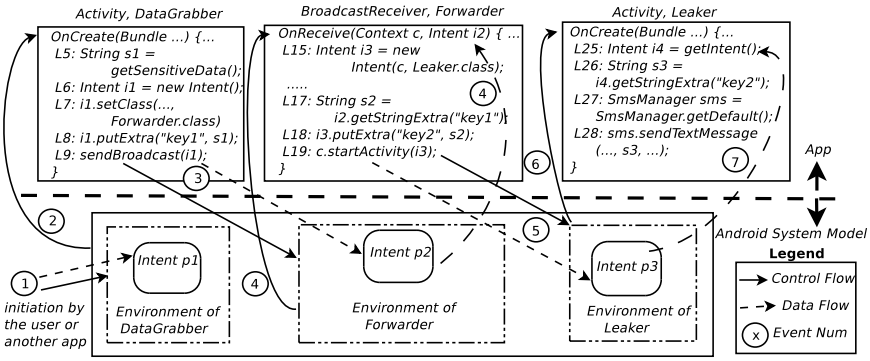
A malicious app can conduct bad behaviors by manipulating the inter-component nature of Android system and try to obfuscate its true objectives. Figure 1a shows an example of such apps (named "sensitive-sms"), with snippets of Java code shown in the boxes above the dotted line, each of which represents a component of the app. In Android, an *Activity* component implements the UI of the app, and a *Broadcast Receiver* component receives a broadcast message from one component (or the system) and takes certain actions. An Android app does not have a "main" method; rather, components are invoked through the various callback methods (including *lifecycle methods*). The control flows and data flows among the app components through the Android system are labeled with the event number. 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*s.

The following sequence of events as labeled in the figure can happen in reality: (1) the user (or another app) launches *DataGrabber*; (2) this causes the Android system to invoke the component's lifecycle method *onCreate()*; (3) this method creates an *explicit intent* and sends it (*L9*) to a

[1]Aman means safe/secure in the Indonesian language.

[2]Amandroid is available in the Sireum software distribution at http://amandroid.sireum.org

(a) A possible execution sequence of the "sensitive-sms" app



(b) The Life Cycle Diagram of an Activity (adapted from [1])

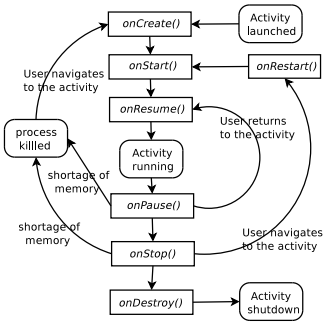


Figure 1: An Android app and an Activity lifecycle

BroadcastReceiver named *Forwarder*; (4) the system invokes *Forwarder*'s lifecycle method *onReceive()*; (5) this method sends an explicit intent (*L19*) to the *Leaker* Activity; (6) this intent causes the system to invoke *Leaker*'s *onCreate()* method; and (7) this method retrieves the intent from the system (*L25*), extracts the data, and sends the data through SMS (*L28*).

A static analyzer needs a model of the Android system to track invocation of component lifecycle methods as illustrated in this example. Our model of the Android environment is inspired by FlowDroid [6, 17], which uses a "dummyMain" method to capture all possible sequences of lifecycle method invocations as permitted by Android. Our model also extends that of FlowDroid by capturing the control and data dependencies among components. For instance, it is able to find the inter-component control flows from *DataGrabber.onCreate* to *Forwarder.onReceive* (Event 3 → 4).

However, we observe that FlowDroid is yet to find critical ICC *data* flows. As an example, it does not find the data flows through the intents sent from *DataGrabber* to *Forwarder* and from *Forwarder* to *Leaker*. *DataGrabber* puts the sensitive information inside *Intent i1* via *putExtra* (which actually populates *i1*'s *mExtras* field) and sends *i1* to *Forwarder*. The Android system then acts as an intermediary and dispatches *i1* to *Forwarder.onReceive* as *i2*.

We have found no prior works that have a mechanism to find the connection between *i1* and *i2*. Furthermore, *Forwarder* creates a new *Intent i3* and transfers the data from *i2* to *i3*. In fact, *s2 equals to s1* that carries the sensitive information, which is now contained in *i3*. Finally, the secret information is further forwarded to *Leaker* through intent *i3* which *Leaker* retrieves via *getIntent()* as *i4*. Mapping back *i4* to *i3* is more complicated as it is not passed as an explicit parameter to the callback method. Again, no prior works can find the link between *i3* and *i4*, without which, one will have no chance of knowing that *s3* (retrieved from *i4*) *equals to s1* and carries sensitive information, which is sent out through SMS.

We observe that to capture this type of intricate information flow, the model of the Android environment needs to include both the control and the relevant data specialized for Android (intent in this case). Moreover, the analyzer must be able to conduct data-flow analysis across component boundaries to identify this type of security problems that require multiple components working together. Prior works (*e.g.*, FlowDroid and Epicc) have made important steps towards this goal, but none has moved further enough. While one could extend the prior works to address this limitation, we use a different approach (outlined in Section 1) which we describe in more details in the following sections.

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 to limit the scope of the analysis for scalability; (3) Amandroid builds an inter-component data flow graph (*IDFG*) of the whole app. *IDFG* includes the control flow graph spanning over all the reachable components of the app; 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); (4) it builds the data dependence graph (*DDG*) on top of the *IDFG*, which implies explicit information flow; and (5) Amandroid then can be applied in various types of security analysis using the information presented in *IDFG* and *DDG*. 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 and retrieves a *dex* file and covert it to an *IR* format for subsequent analysis. Our *dex2IR* translator is a modification of the original *dexdump* tool shipped with the Android platform tool set; the C++ source of the original *dexdump* is available in the Android build package, and we modified it so that it can also produce the app representation in our IR format.

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 "sensitive-sms" app example in Section 2, we demonstrated that a static analyzer needs to model the Android system to analyze the

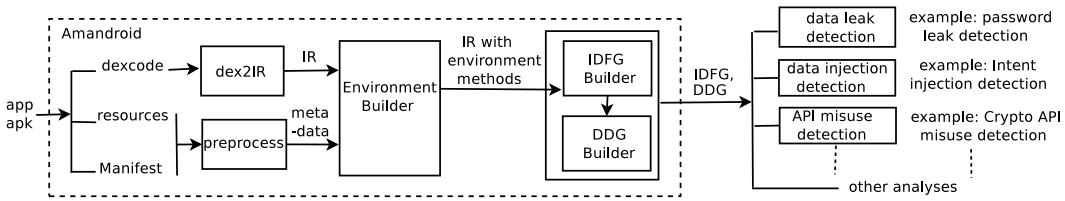


Figure 2: The Amandroid Analysis Pipeline

system-defined control flows in the app[3]. Our modeling of the Android environment follows that of FlowDroid [6, 17] 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 the 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. Figure 1b depicts the life cycle of an Activity. There are seven important life-cycle methods of an Activity: *onCreate*, *onPause*, *onResume*, *etc.*; they each represent a state in the transition diagram. Android documentation specifies other states such as *Activity running* and *Activity shut down*.

Amandroid introduces component-level models instead of FlowDroid's whole app-level model. The environment of a component C represents a main method, E_C, which takes as parameter an incoming intent i and invokes C's life-cycle methods (*e.g.*, *onCreate* 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 and data of an app's execution. The part below the dotted line in Figure 1a highlights this idea: a dedicated environment for each component invokes the set of implemented callback methods; 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 Leaker remembers that *p3* was sent to start *Leaker*) so that the intents could be made available when necessary (*e.g.*, to serve *getIntent()* in the *Leaker* component); this is the data part of modeling Android's environment. E_C also passes the intent parameter when necessary for other relevant methods (*e.g.*, *onReceive*).

Amandroid generates E_c automatically. First, it collects basic information from the resource files in the *apk* and uses this information to collect layout callback methods. It then generates the body of E_c with lifecycle methods based on the type of C. Finally, it collects 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.

3.3 Inter-component Data Flow Graph (*IDFG*)

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 ICC 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 and more elegant to pre-calculate *all* object points-to information at once, and use this as a general framework for different types of further analysis.

Existing off-the-shelf static analysis tools such as Soot [32] (used by FlowDroid [6, 17] and Epicc [25]) and Wala [16] (used by CHEX [23]) have not provided capability of calculating all objects' points-to information in a both flow and context-sensitive way [4, 22].[4] 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.

Thus, the core task of Amandroid's analysis is aimed to build a precise inter-component data flow graph (*IDFG*) of the app; 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 [10]. However, [10] does not keep track of method calling context (typically termed *monovariant* calling context analysis or 0-calling context [24]). We generalize the approach to precisely track the last k calling contexts (*polyvariant* [24], *a.k.a.* k-limiting where k is user-configurable and the additional calling context beyond k is monovariant).

Amandroid follows the classical static analysis approach [24] customized to address the number of aforementioned challenges in analyzing Android apps. It computes points-to facts for each statement. 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.

$$exit(s) = (entry(s) \setminus kill(s)) \cup gen(s) \quad (1)$$

Due to space constraints, the description of the basic *IDFG* building process can be found in Appendix. Below we intro-

[3]The alternative is to fully analyze the whole Android system's code, which is both expensive and unnecessary as also observed by others [17, 23].

[4]More detailed comparison between Amandroid and FlowDroid can be found in Section 6.

duce the notations in *IDFG* and use the example in Section 2 to explain its semantics. Figure 3 is the resulting *IDFG* of the exmaple app, using *DataGrabber* as the entry point.

3.3.1 Notations

A *points-to fact* provides information about what objects a variable (register in Dalvik), an object field, or an array element may point to at a particular program point. 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 number N (represented as LN). We use the term *instance N* to denote the object instance created at statement N (note that the statement gives the exact object runtime type). A *tuple-instance* (*e.g.*, (*"key1"*, 5)) denotes a key-value pair in the fact sets. Amandroid keeps tracks of two kinds of information:

- *variable-fact*: A points-to fact for a variable; it is denoted as $\langle v, l \rangle$, where v is the variable (whose type is an object reference type) and l is an object instance. For example, in Figure 3 statement *L6* generates a variable-fact $\langle i1, 6 \rangle$, meaning that variable *i1* points to *instance 6*.
- *heap-fact*: A points-to fact for an object field or an array element. For example, statement *L8* in Figure 3 generates a heap-fact $\langle (6, mExtras), ($"key1"$, 5) \rangle$, meaning that the field *mExtras* of *instance 6* points to a key-value pair (*"key1"*, 5)[5].

Amandroid starts the *IDFG* building from the *CFG* of the *DataGrabber* component's environment method (the left column of the leftmost dashed box in Figure 3). For brevity only a subset of the nodes and facts are shown.

3.3.2 Modeling Library and Native Calls

Android has a large number of library API's an app may call into, some of which are implemented natively. Similarly, an app developer may choose to natively implement some functionality due to various reasons (*e.g.*, performance). Amandroid does not analyze native code; thus, in order to enable analysis of app making use of native code, we need to provide models for native methods that summarize how the data flow facts that may be changed. For library APIs that have well-understood simple semantics, one can summarize them as flow functions (*gen* and *kill*). Moreover, providing models for non-native library methods that are frequently used are also useful to scale the analysis. This is in line with how we model the Android environment described in Section 3.2.

In general, Amandroid adopts the following strategy in modeling Android library functions: (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); and (2) for all other library functions, 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 with the method's return type. If the function also returns an object, the returned object is also considered "unknown."

In Figure 3, line *L5* in *DataGrabber* generates a variable-fact $\langle s1, 5 \rangle$, indicating that an object is returned from the API call and assigned to *s1*; we use *getSensitiveData* in this example as a generic name for any methods that returns an object with sensitive information. At Line *L8* the sensitive data is inserted as a key-value pair (*"key1"*, *s1*) into intent *i1*'s *mExtras* 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 *i1*. The generated fact at Line *L8* is then $\langle (6, mExtras), ($"key1"$, 5) \rangle$ following our notation for a field-fact, where *6* represents the intent *i1* created at Line *L6*. Note that *instance 5* represents the String object returned from *getSensitiveData()*.

3.3.3 Handling ICC

Section 2 illustrates that malicious apps can easily manipulate Android's inter-component communication (ICC) to stealthily achieve undesired effects. To identify such security problems, a static analyzer needs to be aware of control and data flows across component boundaries. Handling ICC requires a number of steps: (1) solve for ICC call parameters, (2) find the target component(s), and (3) track data flow from the ICC caller to callee.

Prior work [25] has investigated how to infer Android ICC API call parameters (Step 1). Amandroid not only infers such ICC API call parameters using the points-to facts computed, but also uses such information to resolve ICC call targets (Step 2) and link the source with the possible targets in its dataflow analysis (Step 3). This will enable us to detect the security problems like that illustrated in Section 2.

The destination of an ICC 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) or a special constructor for Intent (*L15*). 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. Through proper modeling of these API functions (Section 3.3.2), Amandroid can derive possible (String) values of the relevant fields of an Intent object, upon which the Android system bases its decision on ICC destinations.

For instance, Amandroid can derive that at *L9* in Figure 3, the intent parameter *i1*'s field *mComponentName*[6] is

[5]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.

[6]For the ease of exposition, in this article we represent the *mComponent* field of an intent by its name string. However, we handle this field accurately in the Amandroid implementation.

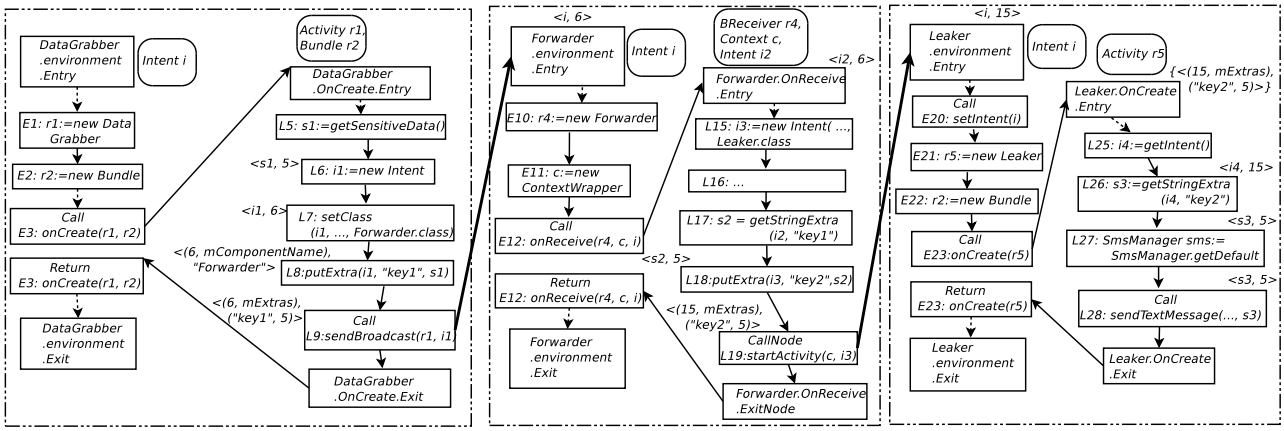


Figure 3: An excerpt from the *IDFG* of the app "sensitive-sms."

"Forwarder." This fact comes from the modeling of the API function *setClass* called at *L7*, which generates a field-fact ⟨*(6, mComponentName), "Forwarder"*⟩, where *6* represents Intent *i1* which was created at *L6*.

For an implicit intent, the Android system finds the destination depending on the intent fields as well as the manifests of all the apps which specify *intent filters* for a component. An intent filter 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 [2] 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 *action test*, *category test*, and *data test* (having both *Uri* and *type*) to find the destination component(s). Our static analysis can readily handle the String literals. 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.

3.4 Building the Data Dependence Graph

The data dependence graph (*DDG*) is derived from the *IDFG*. The *DDG* reflects how instance and variable definitions flow through the program. With the help of *DDG*, we can argue which part(s) of the program a particular program-point depends on with respect to these two types of flows. As a matter of fact, the *DDG* is a directional graph like the *ICFG*. The basic *DDG*'s node set is the same as the *ICFG*'s, but their sets of edges are different. In fact, *DDG* has two kinds of edges: **(i)** object dependence edge – an edge linking the use site of an instance to the creation site of the instance, and **(ii)** variable def-use edge – an edge which links a use site of a variable to the def-site of the variable.

Since object flow along ICC edges is already captured in *IDFG*, the constructed *DDG* automatically captures data dependencies across component boundaries. As an example, in Figure 3, the *sendTextMessage(..., s3)* in *Leaker* uses *s3* while the *entry* of this statement has a fact ⟨*s3, 5*⟩ which implies that *Instance 5* is used in this statement. So, there is an object dependence edge from the corresponding *CallNode* (*L28*) in the *Leaker* component to the creation site (*L5*) in the *DataGrabber* component.

3.5 Using Amandroid for Security Analyses

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

3.5.1 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*. All that is needed for this analysis is to specify the source and the sink, which can be any node depending on the specific problem. For instance, prior research [7, 17] 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 *ICFG* through the standard process [5].

Compared with prior works on detecting information leaks on Android apps, Amandroid can perform a more comprehensive analysis since it captures control and data flows across the component boundaries through ICCs, so that security problems like the one shown in Figure 1a can be captured.

3.5.2 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. Recently, researchers [23] 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.

As an example, recent research [34] found (*manually*) the *next intent vulnerability* of popular apps such as Dropbox, which is a special intent injection problem involving ICC. As Amandroid is able to track data dependencies over the ICC links, our aforementioned analysis technique is able to find the "next intent" problems in an *automated* fashion. In fact, Amandroid was able to rediscover this issue in the Dropbox app.

3.5.3 Detecting Misuse of an API

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. Recent research has applied static analysis to identify misuse of Crypto APIs [11] and SSL APIs [14]. 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 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 *IDFG*.

4. EXPERIMENTATION AND EVALUATION

We extensively experimented with Amandroid in multiple types of security analyses. We used several sets of apps: 753 popular apps from Google Play (the same dataset used in the Epicc work [25] and made available by the authors), a sample malware set (containing 100 apps) from Arbor Networks, and two benchmarks (hand-crafted apps by other researchers and us). For brevity, we call the first two data sets GPlay and MAL, respectively.

Our security analysis found multiple crucial problems in the apps, which we report in this section. Most of our results were never reported before by other researchers in the literature; some of the problems detected by Amandroid are in the same category (but different instances) of the previously reported ones (*e.g.*, password leak), while others are completely new categories (*e.g.*, OAuth token leak). As Amandroid is the only tool which tracks data flows through ICC, Amandroid is able to find sophisticated data leak and data injection problems as illustrated by the results.

4.1 Performance and Scalability

We perform our experiments on a machine with 2×2.26 GHz Quad-Core Xeon and 32GB of RAM.

Amandroid gives options for multiple precision levels. For instance, the context length k serves as a parameter to set the trade-off between precision and performance. In our experimentation, we always set $k = 1$, meaning the static analyzer tracks up to one calling context. Amandroid also allows the user to define the scope of the analysis by excluding certain third-party libraries, and in our experiment we excluded a few popular third-party libraries since they are huge in size and could be separately analyzed, summarized, and reused by the analysis of all the apps that include them. Currently, we use the same modeling techniques as explained in Section 3.3.2 for the excluded third-party libraries.

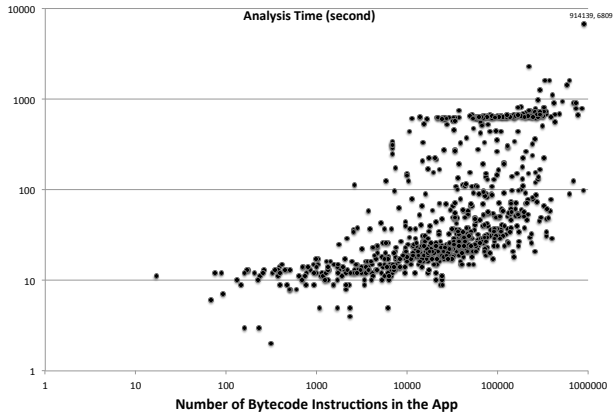


Figure 4: Time to Build *IDFG*

The most computational intensive step in Amandroid is building the *IDFG*. Once the *IDFG* is built, the running times of the subsequent analyses such as building *DDG* and running the specialized analyses using *IDFG* and *DDG* are negligible in comparison. Figure 4 presents the time taken by Amandroid to construct *IDFG* for 853 apps. This measurement is done on the data-sets GPlay and MAL which are all real-world apps. During the experiment, we limited the processing time of *each component* of an app to 10 minutes; Amandroid raised this timeout on 86 of the 853 apps. For each app, which consists of multiple components, the median time is 29 seconds; minimum is 2 seconds; and maximum is 113 minutes and 29 seconds. The scatter plot shows both the running time and the size of the app (in number of byte code instructions).

4.2 Application to Security Analysis

We report experimentation results addressing data leak, data injection, and misuse of APIs (as discussed in Section 3.5). All experiments are done using the GPlay and MAL datasets (real-world apps).

4.2.1 Data Leak

Password Leaks: We used the following policy to vet apps for properly handling user passwords: "password should not be saved in the device (not even when encrypted) and should be transferred to a remote server only via HTTPS." (similar guidelines can be found in, *e.g.*, [13]). Amandroid can be readily used to verify whether the input app obeys such a policy. The only "to-do" task is to identify which variables in the app's code corresponds to a password object (source), and to define the potential leaking sinks.

We find the *TextView* item corresponding to a password (when the *inputType* attribute's value is *textPassword*) in an app's layout file and identify its unique ID. Amandroid then looks for the usage of this particular ID in method call *Context.getViewbyId(x)*, which is done through a standard reaching-definition analysis on the intra-procedural control-flow graph[7]; this method returns an *EditText* object y, and *y.getText()* gives the password object. We can then define

[7]If the app's developer obfuscates the ID through, *e.g.*, mathematical manipulation, our reaching-definition analysis will not be able to return concrete values for the view

this object as the source. We prepare the list of sink APIs by considering the relevant I/O operations (*e.g.*, *Log.i(key, value)*, and *URL.openconnection()*). The rest of the analysis is the straightforward application of *DDG* as explained in Section 3.5.

We found several examples of password leakage. Table 1 shows part of the results. We observe a few interesting patterns: (a) the password is logged in clear text (Case 1 in Table 1); (b) the password is reaching a Network API over HTTP channel (Entry 2 in Table 1); and (c) the password is saved in *SharedPreference* (Entry 3 in Table 1). Case 2 stems from a third-party library for Twitter. The *DDG* and *IDFG* shows that the app sends the user's password to `http://api.twitter.com/1` (an HTTP connection). Interestingly, one can see that the URL is not currently working and only responds with a message "SSL is required."

Table 1: Password Leakage Case Study

App name (app source)	App behavior
Case 1: com.datpiff.mobile.apk (GPlay)	Get user password, encode it, then write it into log.
Case 2: com.toystorymusic-.musicapp.* (MAL)	Send password to server via http.
Case 3: com.snappii.angel_investing_news_v10.apk (GPlay)	Write user password into SharedPreference.

OAuth Token Leaks: OAuth 2.0 [21] is a popular authentication protocol which is frequently used for single-sign-on (SSO), social sharing, *etc.* Typically, Google, Facebook, and other popular services are the Identity Provider (*IdP*). Thus, if the OAuth token is stolen, the user's corresponding *IdP* account can be compromised. Similar to password tracking, Amandroid can be used to check whether the input app obeys the OAuth token protection policy. The source of the potential leak is determined using a simple strategy of tracking the string literal "access-token" and marking the related object creation statements as the source. The sinks are the same as in the password leak detection. We found several potential OAuth token leakage cases, some of which are shown in Table 2. We observe a couple of interesting patterns: (a) The implicit intent carrying the token can possibly reach a malicious app. (*e.g.*, Case 1 in Table 2), and (b) A malicious app having Log-read permission can grab the OAuth token (*e.g.*, Case 1 and Case 2 in Table 2). Note that the above type of token leakage is very different from the explicit token discoveries reported in a recent work [33].

Table 2: OAuth Token Leakage Case Study

App name (app source)	App behavior
Case 1: com.skout.android.apk (GPlay)	Send OAuth token via implicit ICC; also write it to *Log.i*().
Case 2: com.keek.apk (GPlay)	Write OAuth token into log using *Log.d*().

4.2.2 Data Injection

We found a variety of intent injection problem in our experiment; Table 3 shows part of the results. We observe a couple interesting patterns: (a) The attacker controls the "url" string in the TwitterLoginActivity. (Case 1), and (b) The destination of an ICC depends on an incoming intent controlled by the attacker (Case 2).

ID. In this case we will conservatively report a possible malicious app, since it is extremely unlikely a benign app will perform such manipulations on a view ID.

Table 3: Intent Injection Case Study

App name (app source)	App behavior
Case 1: com.fcbh.dbp.BibleSocietyOfPhilippines.apk (GPlay)	TwitterLoginActivity retrieves the "url" from incoming intent and sends it to another Activity
Case 2: com.kamagames.notepad.apk (GPlay)	Start an activity by using the *mData* of the incoming intent

Table 4: Crypto API Usage Case Study

App name (app source)	App behavior
Case 1: hu.sanomabp.citromail-.apk (GPlay)	Encrypt OAuth token using AES ECB mode, then store it in SharedPreference.
Case 2: diesel.peko.ninkyodobrowser.apk (GPlay)	Encrypt the password using AES ECB mode.

4.2.3 API Misuse

We found several apps that violate the rule on not using ECB mode for encryption. Table 4 shows part of the results. The apps encrypt the user credential using the AES cipher in ECB mode.

4.2.4 What it Takes to Build a New Analysis

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 "plugin" that leverages the *IDFG* and *DDG* from Amandroid's analysis. Moreover, once the core analysis produces *IDFG* and *DDG* 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 5, which shows the sizes of the plugin in Scala LOC, as well as the average running time of each plugin. 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 5: Code Size and Running Time (Plugins and Core)

Name	Approx. Size (Scala LOC)	Avg. Time
Tracking password plugin	120	50ms
Tracking OAuth Token plugin	120	50ms
Generic Data Leak plugin	60	300ms
Intent Injection plugin	70	50ms
Crypto API check plugin	140	10ms
Core Framework	30,000	60s

4.3 Comparison with Existing Tools

We use three benchmark testsuites[8] to compare Amandroid with two other well-known static analysis tools for Android: FlowDroid [6, 17] and Epicc [25]. 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, allowing for computing metrics such as precisions and recalls. However, one needs to bear in mind that these metrics are not representative of the real

[8]Using real-world apps for comparison will be difficult since there is no easy way to determine the ground truth. For example, we have used the GPlay dataset to compare Amandroid and Epicc and found much less reachable ICC call sites than Epicc. This could indicate that Amandroid is more precise than Epicc (both tools are sound and thus will not miss a path); however, without ground truth in the apps it is impossible to test this hypothesis.

performance of the tool on real-world apps. They can only be used for comparison purposes.

The first benchmark is DroidBench, a benchmark testsuite provided by the FlowDroid team that consists of Android apps for evaluating information-flow analysis. The version we used contains 39 apps, including test cases for static analysis challenges as well as Android-specific challenges. As DroidBench does not test any ICC-related capability, we added another testsuite called ICC-Bench which contains 16 apps for testing various ICC reasoning capabilities. Each test app has two components where one component sends an Intent to the other one. The sender component involves a source API while the receiver contains a sink API. The test apps are categorized in two parts: 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 information flow paths do or do not exist. The list of ICC-Bench apps can be found in the 1st column of Table 8. The third testsuite consists of four specially designed test apps (each with a single component) to test the data injection detection capability. While the latter two testsuites were designed by us and not a third party, the apps in these testsuites are not crafted to favor a particular tool. They represent common scenarios one will find when reasoning about the relevent security issues. We plan to make both these testsuites publicly available.

4.3.1 ICC Test

Table 6 summarizes the result of ICC testing using Part A of ICC-Bench. As discussed in Section 3.3.3, to completely handle ICC an analysis tool needs to carry out three steps. Amandroid is able to successfully pass all the tests for the three steps. Epicc only addresses Step 1 thus did not pass the tests on Step 2 and 3 (shown as "N/A" in the table). FlowDroid does not address ICC thus did not pass any of the tests. For Step 1, Amandroid is able to handle all types of intents: explicit, implicit, and mixed (either explicit or implicit depending on execution paths). As discussed in Section 3.3.3, an implicit intent can behave in multiple ways depending on which field (*mAction*, *mCategories*, *mData* (*i.e.*, *Uri*[9]), or *mType*) is used. Amandroid can track each such field and hence can handle all types of implicit ICC. Epicc is the only existing tool which attempts to solve ICC parameters. However, Epicc does not handle *mData* or *mType* field of an intent, and thus failed those tests.

4.3.2 Data Leak

We compare the effectiveness of Amandroid's data leak detection with the other tools on two benchmarks: DroidBench and ICC-Bench. The tool is run on each test app to see if the tool can report the correct data leak paths. 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. Only FlowDroid and Amandroid can perform static taint analysis to find those leak paths; Epicc only outputs information based on ICC call parameters and thus cannot find the actual leak paths.

[9] *Uri* consists of multiple parts, such as *scheme*, *host*, *port*, *path*; a tool must be able to determine the values of each part to pass the test.

Table 7: DroidBench Test Results. O = True Positive, * = False Positive, X = False Negative.

App Name	FlowDroid	Amandroid	Epicc
Arrays and Lists			
ArrayAccess1	*	*	N/A
ArrayAccess2	*	*	
ListAccess1	*	*	
Callbacks			
AnonymousClass1	O	O	N/A
Button1	O	O	
Button2	OOO*	OOO	
LocationLeak1	OO	OO	
LocationLeak2	OO	OO	
MethodOverride1	O	O*	
Field and Object Sensitivity			
FieldSensitivity1			N/A
FieldSensitivity2			
FieldSensitivity3	O	O	
FieldSensitivity4			
InheritedObjects1	O	O	
ObjectSensitivity1			
ObjectSensitivity2			
FieldSensitivity1			
Inter-App Communication			
IntentSink1	X	O	N/A
IntentSink2	O	O	
ActivityCommunication1	O	X	
Lifecycle			
BroadcastReceiverLifecycle1	O	O	N/A
ActivityLifecycle1	O	O	
ActivityLifecycle2	O	O	
ActivityLifecycle3	O	O	
ActivityLifecycle4	O	O	
ServiceLifecycle1	O	O	
General Java			
Loop1	O	O	N/A
Loop2	O	O	
SourceCodeSpecific1	O	O	
StaticInitialization1	X	O	
UnreachableCode			
Miscellaneous Android-Specific			
PrivateDataLeak1	O	O	N/A
PrivateDataLeak2	O	O	
DirectLeak1	O	O	
InactiveActivity			
LogNoLeak			
Implicit Flows			
ImplicitFlow1	XX	XX	N/A
ImplicitFlow2	XX	XX	
ImplicitFlow3	XX	XX	
ImplicitFlow4	XX	XX	
Sum, Precision and Recall — DroidBench			
O, higher is better	26	27	N/A
*, lower is better	4	4	
X, lower is better	10	9	
Precision p = O/(O + *)	86%	87%	
Recall r = O/(O + X)	72%	75%	
F-measure 2pr/(p + r)	0.78	0.81	

The detailed comparison of the performance of FlowDroid and Amandroid on DroidBench and ICC-Bench is available in Table 7 and 8. Not surprisingly, Amandroid outperforms FlowDroid on the ICC-Bench since FlowDroid does not handle ICC. The two perform similarly on the DroidBench test suite.

4.3.3 Data Injection

Table 9 compares Amandroid, FlowDroid and Epicc in the context of data injection detection performance, using the third testsuite. Since this datasuite only consists of apps with a single component, FlowDroid is able to handle most

Table 6: ICC Test Result

Tools	Step 1: Solve ICC call parameters						Step 2: Find the target component(s)	Step 3: Track the ICC data flow
	Explicit ICC	Implicit ICC				Mixed		
		mAction	mCategories	mData	mType			
Epicc	✓	✓	✓	✗	✗	✓	N/A	N/A
Amandroid	✓	✓	✓	✓	✓	✓	✓	✓
FlowdDroid	N/A						N/A	N/A

Table 8: Results on ICC-Bench. O = True Positive, * = False Positive, X = False Negative.

App Name	FlowDroid	Amandroid	Epicc
Part A — Testing ICC Addressing			
ICC_Explicit1	*X	O	N/A
ICC_Implicit_Action	OX	OO	
ICC_Implicit_Category	OX	OO	
ICC_Implicit_Data1	OX	OO	
ICC_Implicit_Data2	OX	OO	
ICC_Implicit_Mix1	OXX	OOO	
ICC_Implicit_Mix2	OX	OO	
ICC_DynRegisteredReceiver1	OX	OO	
Part B — Testing ICC Data Flow Tracking			
ICC_Explicit_NoSrc_NoSink			N/A
ICC_Explicit_NoSrc_Sink			
ICC_Explicit_Src_NoSink	*		
ICC_Explicit_Src_Sink	*X	O	
ICC_Implicit_NoSrc_NoSink			
ICC_Implicit_NoSrc_Sink			
ICC_Implicit_Src_NoSink	O	O	
ICC_Implicit_Src_Sink	OX	OO	
Sum, Precision and Recall — ICC-Bench			
O, higher is better	9	20	N/A
*, lower is better	3	0	
X, lower is better	11	0	
Precision p = O/(O + *)	75%	100%	
Recall r = O/(O + X)	45%	100%	
F-measure 2pr/(p + r)	0.56	1.00	

Table 9: Data Injection Detection Comparison. FP = False Positive, FN = False Negative.

App Feature		FlowDroid	Epicc	Amandroid
Public Comp	data reach sink	✓	✓	✓
	not reach sink	✓	FP	✓
Private Comp	data reach sink	FP	✓	✓
	not reach sink	✓	✓	✓

of them, execpt for one case where it raised a false alarm due to not being aware of a component's *exported* status. Although not included in the testsuite, FlowDroid would have False Negative (FN) when the app involves ICC (*e.g.*, "next intent" vulnerability [34]). On the other hand, Epicc takes a simple worst-case approach to detect a data injection problem, which assumes that any public component can have such vulnerability. However, this conservative approach will cause false alarms where there is no data flow path from the public component to the sensitive operation sink. Amandroid can correctly handle all the cases.

5. DISCUSSIONS

Amandroid has limited capability to handle exceptions. If an app has a security issue where the code of an exception handler plays a role, Amandroid may not detect it. We will address this limitation in future work. Amandroid does not currently handle reflections and concurrency. Adding support for reflections is similar to handling ICC in Amandroid, which already has some preliminary string analysis capability.

An app may have multiple components and they may run concurrently. There could be security problems that only manifest when multiple components interleave in certain ways. Handling concurrency in a general way in static analysis is nontrivial; like in other prior works we leave this for future research. For example, we could follow the approaches that have been developed from prior research [10].

Amandroid's data and control flow analysis depends on the faithfulness of the models, including the models of the Android environment and its APIs. Due to the size of the library, it still remains a challenge to develop a precise and sound model for every library API.

6. RELATED WORK

There has been a long line of works on applying static analysis for Android security problems [6, 8, 11, 14, 17, 20, 23, 25]. 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, 17] (*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 [32], 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 [28, 29] 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 [17]. Amandroid calculates all objects' points-to information in a both context- and flow-sensitive way, with reasonable computing cost (ref. Section 4.1). This enables us to build the generic framework supporting multiple security analyses.

Epicc [25] computes Android ICC 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 ICC parameter analysis result to resolve the ICC call targets in the general case, and has not used the result to perform inter-component dataflow analysis. Amandroid's approach to deriving ICC parameters is to simply use the flow and context-sensitive points-to information (including that for string objects) already computed in the *IDFG*, without the need for a separate data flow analysis just for ICC. Aman-

droid also uses the ICC call parameter information to link ICC call sites to call targets, resulting in an *IDFG* that includes data flow paths both within and across components.

Lu *et al.* [23] 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 [16]. 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 *IDFG* and *DDG* for the complete app. CHEX does not have the provision to track data flow through the ICC channel, which Amandroid does.

Chin *et al.* [8] first systematically studied the attack surface related to ICC. 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. ComDroid performs flow-sensitive, intraprocedural static analysis, and the paper states that there is a limited interprocedural analysis that "follows method invocations to a depth of one method call." Amandroid performs a full-fledged interprocedural 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 (`http://www.comdroid.org`) is not working. We contacted the authors for obtaining a copy of the tool and dataset used for evaluation, but have yet to receive the information.

There has been a large body of work reporting Android app security issues [36, 37], some of which use static analysis techniques [11, 14, 18, 19]. 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 [9, 26, 35] investigated the root security problems in the Android system and proposed augmented infrastructures to enforce the given security policy. Recently, SEAndroid [30] 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 [12] 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 [27] that Google's Bouncer [3] 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.* [31] 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.

7. CONCLUSION

We presented Amandroid – a general static analysis framework for security analysis of Android applications. Amandroid can precisely track the control and data flow of an app across multiple components, and can compute an abstraction of the app's behavior in the forms of an inter-component 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 and can be readily applied to effectively address those specialized security problems, and out-performs existing static analysis tools for Android apps.

Acknowledgment

We express our gratitude to Eric Bodden and Steven Arzt for helping us understand the FlowDroid work, to Patrick McDaniel and Damien Octeau for sharing with us the dataset used in Epicc, and helping us better understand the work, and to Marc Eisenbarth and Arbor Networks for sharing with us the Android malware samples. We also thank Gang Tan who provided valuable feedback for our wrok. Venkatesh Prasad Ranganath contributed many ideas in the discussions we had together. This work was partially supported by the U.S. National Science Foundation under grant no. 0644288, 0954138 and 1018703, 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.

8. REFERENCES

[1] Android documentation: Activity. http://developer.android.com/reference/android/app/Activity.html.

[2] Android documentation: Intent and Intent Filter. http://developer.android.com/guide/components/intents-filters.html.

[3] Google Bouncer. http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[4] WALA documentation: CallGraph. http://wala.sourceforge.net/wiki/index.php/UserGuide:CallGraph.

[5] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. l. Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the ACM PLDI*, 2014.

[7] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android permission specification. In *Proceedings of the ACM CCS*, 2012.

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the ACM Mobisys*, 2011.

[9] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich. CRePE: A system for enforcing fine-grained context-related policies on Android. *Information Forensics and Security, IEEE Transactions on*, 7(5):1426–1438, 2012.

[10] M. B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, Robby, and T. Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Proceedings of the TACAS*, 2006.

[11] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the ACM CCS*, 2013.

[12] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX OSDI*, 2010.

[13] ENISA. Smartphone secure development guidelines. http://www.enisa.europa.eu/activities/Resilience-and-CIIP/critical-applications/smartphone-security-1/smartphone-secure-development-guidelines, 2011.

[14] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *Proceedings of the ACM CCS*, 2012.

[15] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011.

[16] S. Fink and J. Dolby. WALA–The TJ Watson Libraries for Analysis. http://wala.sf.net/, 2012.

[17] C. Fritz, S. Arzt, S. Rasthofer, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. Highly precise taint analysis for Android application. Technical report, EC SPRIDE, 2013.

[18] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Proceedings of the International Conference on Trust and Trustworthy Computing*, 2012.

[19] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock Android smartphones. In *Proceedings of the NDSS*, 2012.

[20] M. C. Grace, W. Zhou, X. Jiang, and A. R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[21] D. Hardt. The OAuth 2.0 authorization framework. 2012.

[22] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Proceedings of the Compiler Construction*, 2003.

[23] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically vetting Android apps for component hijacking vulnerabilities. In *Proceedings of the ACM CCS*, 2012.

[24] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[25] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in Android with Epicc: An essential step towards holistic security analysis. In *Proceedings of the USENIX Security Symposium*, 2013.

[26] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in Android. *Security and Communication Networks*, 5(6):658–673, 2012.

[27] N. J. Percoco and S. Schulte. Adventures in Bouncerland. *Black Hat USA*, 2012.

[28] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1995.

[29] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1):131–170, 1996.

[30] S. Smalley and R. Craig. Security enhanced (SE) Android: Bringing flexible MAC to Android. In *Proceedings of the NDSS*, 2013.

[31] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan. SMV-HUNTER: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *Proceedings of the NDSS*, 2014.

[32] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of the Compiler Construction*, 2000.

[33] N. Viennot, E. Garcia, and J. Nieh. A measurement study of Google Play. In *Proceedings of the ACM SIGMETRICS*, 2014.

[34] R. Wang, L. Xing, X. Wang, and S. Chen. Unauthorized origin crossing on mobile platforms: Threats and mitigation. In *Proceedings of the 2013 ACM CCS*, 2013.

[35] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for Android applications. In *Proceedings of the USENIX Security Symposium*, 2012.

[36] Y. Zhou and X. Jiang. Dissecting Android malware: Characterization and evolution. In *Proceedings of the IEEE SP*, 2012.

[37] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the NDSS*, 2012.

9. APPENDIX

The Basic *IDFG* 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

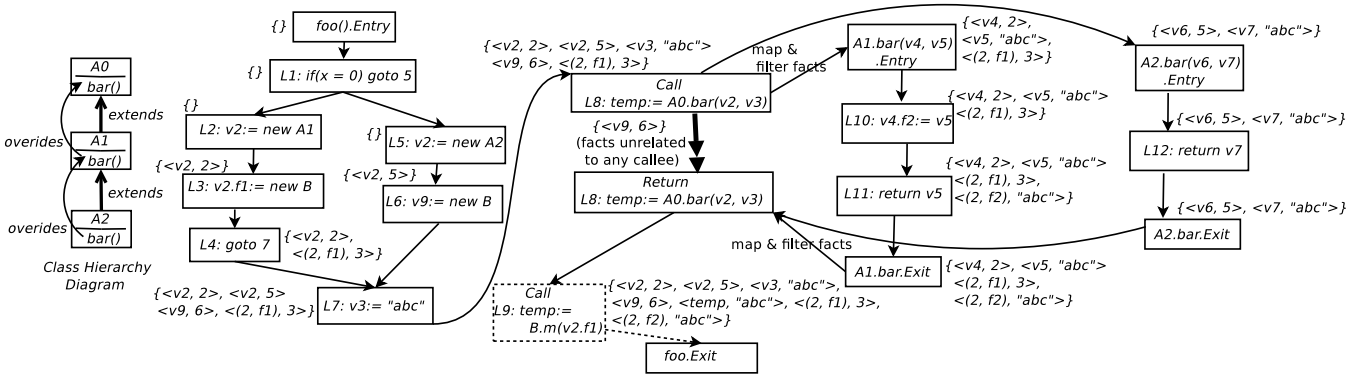


Figure 5: Building the *IDFG* for *foo*: The intra-procedural *CFG* of *foo* is extended to a callee, *bar*.

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 *IDFG* 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 *IDFG*s. Formally, let C be a component, the *IDFG* from C is denoted $IDFG(E_C)$ where E_C is the environment method of C, and is a tuple defined as the following.

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

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 $IDFG(E_C)$ 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 *IDFG* 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).

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 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 5, 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 $\langle v9,\ 6\rangle$ 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 $\langle v2,\ 5\rangle$). 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.