Information-Flow Analysis of Android Applications in DroidSafe

Michael I. Gordon*, Deokhwan Kim*, Jeff Perkins*, Limei Gilham†, Nguyen Nguyen‡, and Martin Rinard*
*Massachusetts Institute of Technology
mgordon@mit.edu, dkim@csail.mit.edu, jhp@csail.mit.edu, rinard@csail.mit.edu
†Kestrel Institute
gilham@kestrel.edu
‡UWIN Software, LLC
nguyen@uwinsoftware.com

***Abstract*—We present DroidSafe, a static information flow analysis tool that reports potential leaks of sensitive information in Android applications. DroidSafe combines a comprehensive, accurate, and precise model of the Android runtime with static analysis design decisions that enable the DroidSafe analyses to scale to analyze this model. This combination is enabled by accurate analysis stubs, a technique that enables the effective analysis of code whose complete semantics lies outside the scope of Java, and by a combination of analyses that together can statically resolve communication targets identified by dynamically constructed values such as strings and class designators.**

Our experimental results demonstrate that 1) DroidSafe achieves unprecedented precision and accuracy for Android information flow analysis (as measured on a standard previously published set of benchmark applications) and 2) DroidSafe detects all malicious information flow leaks inserted into 24 real-world Android applications by three independent, hostile Red-Team organizations. The previous state-of-the art analysis, in contrast, detects less than 10% of these malicious flows.

I. Introduction

Sensitive information leaks, as implemented by malicious or misused code (such as advertising libraries) in Android applications, constitute one of the most prominent security threats to the Android ecosystem [1, 2]. Android currently supports a coarse-grain information security model in which users grant applications the right to access sensitive information [3]. This model has been less than successful at eliminating information leaks [1], in part because many applications need to legitimately access sensitive information, but only for a specific limited purpose — for example, an application may legitimately need to access location information, but only with the right to send the information to authorized mapping servers.

Motivated by this problem, researchers have developed a variety of systems that are designed to analyze or explore the information flows in Android applications. Dynamic analysis frameworks execute instrumented versions of Android applications and observe behaviors [4–6]. Potential downsides of this approach include missed information flows that are not exercised during testing and, in some cases, the ability of the malicious application to detect the testing and modify its behavior to avoid exercising the malicious flow [4]. They also suffer from denial-of-service attacks if malware is activated during application execution and the application is killed or functionality is disabled.

Static analysis frameworks attempt to analyze the application before it executes to discover all potential sensitive flows [7–13]. Standard issues that complicate the construction of such systems are the challenges of 1) scaling to large applications and 2) maintaining precision in the analysis such that it does not report too many flows that do not actually exist in the application. One particularly prominent issue with developing static analyses for Android applications is the size, richness, and complexity of the Android API and runtime, which typically comprises multiple millions of lines of code implemented in multiple programming languages. Because sensitive flows are often generated by complex interactions between the Android application, API, and runtime, any static analysis must work with an accurate model of this runtime to produce acceptably accurate results.

Accuracy is critical for a static analysis seeking to calculate security properties of an application; any inaccuracies in the execution model provide a motivated attacker with the opportunity to insert malicious flows that will not be captured by an analysis. Also, imprecision in a model could lead to results that are unusable due to too many false positives; another target for a motivated attacker. To the best of our knowledge, the difficulty of obtaining an acceptably accurate and precise Android model has significantly limited the ability of previous systems to successfully detect the full range of malicious information flows in Android applications.

DroidSafe

We present a new system, DroidSafe, for accurately and precisely analyzing sensitive explicit information flows in large, real-world Android applications. DroidSafe tracks information flows from sources (Android API calls that inject sensitive information) to sinks (Android API calls that may leak information). We evaluate DroidSafe on 24 complete real-

NDSS '15, 8-11 February 2015, San Diego, CA, USA
Copyright 2015 Internet Society, ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23089

world Android applications that, as part of the DARPA Automated Program Analysis for Cybersecurity (APAC) program, have been augmented with malicious information flow leaks by three hostile Red Team organizations. The goal of these organizations was to develop information leaks that would either evade detection by static analysis tools or overwhelm static analysis tools into producing unacceptable results (by, for example, manipulating the tool into reporting an overwhelming number of false positive flows). DroidSafe accurately detects all of the 69 malicious flows in these applications (while reporting a manageable total number of flows). A current state-of-the-art Android information-flow analysis system, FlowDroid [8] + IccTA [14], in contrast, detects only 6 of the 69 malicious flows, and has a larger ratio of total flows reported to true malicious flows reported.

Additionally, we evaluate DroidSafe on DroidBench, a suite of 94 Android information-flow benchmarks from the developers of FlowDroid and IccTA, and report the highest accuracy (most actual flows reported) and highest precision (fewest false positive flows reported) for this benchmark suite to date, 94.3% and 87.6% respectively. DroidSafe fails to report only the implicit flows in DroidBench. Finally, we evaluate DroidSafe on a suite of 40 Android explicit information-flow benchmarks developed by us to add coverage to DroidBench; DroidSafe achieves 100% accuracy and precision for the suite, compared to FlowDroid + IccTA's 34.9% accuracy and 79.0% precision.

One advantage of working with applications that contain known inserted malicious flows is the ability to characterize the accuracy of our analysis (i.e., measure how many malicious flows DroidSafe was able to detect). As these results illustrate, DroidSafe implements an analysis of unprecedented accuracy and precision. To the best of our knowledge, DroidSafe provides the first usable information-flow analysis for Android applications [7–14].

The Android Model and Analysis Co-design

Given the extensive and complex interactions between the Android execution environment and Android applications, an accurate and precise information-flow analysis for Android applications requires a comprehensive and accurate model of the Android environment. To obtain such a model, we started with the Android Open Source Project (AOSP) [15] implementation, which contains a Java implementation of much of the Android environment. The goal was to maximize accuracy and precision by directly analyzing as comprehensive a model of Android as feasible.

As we worked with AOSP, it quickly became apparent that the size and complexity of the Android environment made it necessary to develop the model and the analysis together as an integrated whole, with the design decisions in the model and the analysis working together synergistically to enable an effective solution to the Android static information-flow analysis problem. The result is the first accurate and precise model of the Android environment and the first analysis capable of analyzing such a model.

Accurate Analysis Stubs: While the AOSP provides an accurate and precise model for much of Android, it is missing critical parts of the Android runtime. And for good reason — it is currently not practical to implement much of the Android runtime in Java. We therefore developed a novel technique, *accurate analysis stubs*, to enable the effective analysis of code whose full semantics lies outside the scope of AOSP. Each stub is written in Java and only incompletely models the runtime behavior of the modeled code. But the semantics of the stub is complete for the abstractions that the analysis deploys (in this case points-to and information-flow analyses). Examples of semantics missing in the AOSP and added via accurate analysis stubs include native methods; event callback initiation with accurate context; component life-cycle events; and hidden state maintained by the Android runtime and accessible to the application only via the Android API.

Accurate analysis stubs simplify the development of the analysis — they eliminate any need to develop a library of method summaries written in a different specification language [16, 17], any need to conservatively hard code policies within the analysis that attempt to compensate for the missing semantics [8], or any need to analyze code written in multiple languages. They also simplify the development of the model — they enable the developers of the model to work flexibly and efficiently within the familiar implementation language. And they support the use of sophisticated language features such as inheritance, polymorphic code reuse, exceptions, and threads, all of which promote effective engineering of stubs that accurately and precisely model key aspects of the Android environment.

In addition to code, accurate analysis stubs also support the use of Java objects to model otherwise hidden state maintained by the Android runtime. Examples of such state include Android `Activity` saved state, the global `Application` object, `Intent`, `Parcel`, shared preferences, and the file system. Accurate analysis stubs enable DroidSafe to be the first analysis to accurately model these key Android features.

The AOSP implementation overlaid with our accurate analysis stubs represents our model of the Android API and runtime. We call this model the *Android Device Implementation* (ADI). Each application is analyzed in the context of the ADI, approximately 1.3 MLOC. For the information-flow analysis, we manually identified and classified 4,051 sensitive source methods and 2,116 sensitive sink methods in the Android API.

Scalable, Precise Points-To Analysis: Both our Android model and Android applications heavily use sophisticated language features (such as inheritance and polymorphic code reuse) that are known to significantly complicate static program analyses. To preserve acceptable precision, DroidSafe therefore deploys a modern global *object-sensitive* points-to analysis specifically designed to analyze code that uses such features [18]. DroidSafe further enhances scalability by identifying classes that are not relevant to the information flow and eliminating object sensitivity for instances of these classes. This Android-specific optimization enables our global points-to analysis to achieve a context depth greater than what was achieved in prior work [18, 19], delivering a precise analysis appropriately tailored for solving Android information-flow problems.

Flow-Insensitive Analyses: DroidSafe employs a flow-insensitive information-flow analysis. Many interactions between Android applications and the Android environment are mediated by asynchronous callbacks. Because our DroidSafe implementation uses flow-insensitive points-to and information-flow analyses, it accurately considers all possible runtime event orderings that asynchronous callbacks can trigger. Developers of flow sensitive analyses, in contrast, have had difficulty obtaining a model that correctly exposes all of these event orderings to a flow sensitive analysis (see Section VIII). Because the analysis does not consider all event orderings, it may miss sensitive flows.

Critically, flow insensitivity also enables the analysis to scale to analyze an accurate and precise Android model and therefore to accurately and precisely track information flows through the Android environment. Scalability issues restrict flow-sensitive analyses to significantly less accurate and precise Android models characterized by imprecise conservative flow summaries and/or blanket policies for Android API methods [8, 9, 11]. Our results show that the ability to analyze an accurate and precise model more than makes up for any loss of precision caused by flow insensitivity (see Section VIII).

Static Communication Target Resolution: Information flows in Android apps may involve inter-component (between application components) and inter-application (between separate installed apps) communication; communication targets are identified by dynamically constructed values (such as String, `Uri`, and class designators) packaged in an `Intent` object.

To precisely analyze such flows, DroidSafe combines 1) accurate analysis stubs, 2) a internal representation of all defined `IntentFilter` registrations, 3) an analysis of operations that construct strings; this analysis delivers regular expressions that accurately summarize the strings that the application will construct when it runs, 4) a novel points-to analysis that precisely tracks Strings, and 5) algorithms that rewrite the DroidSafe intermediate representation to directly invoke resolved targets. Because DroidSafe works with a comprehensive model of the Android environment, it supports precise resolution of communication targets whose identification (typically via an `Intent`) involves significant interactions with the Android API.

These techniques enable DroidSafe to precisely analyze calls that start `Activity` components; start, stop, and bind `Service` components; invoke RPC calls on `Service` components; send and receive `Service` messages; broadcast messages to `BroadcastReceiver` components; and perform operations on `ContentProvider` components (shared databases). For `Intent`-based resolution, DroidSafe incorporates `IntentFilter` registrations defined both in the Android manifest and those defined programmatically in app code. We are aware of no other analysis that can provide comparable or even usable levels of accuracy or precision for all of these critical Android communication mechanisms.

Contributions

This paper identifies and implements, for the first time, an effective point in the overall Android information-flow design space. Our overarching contribution is the identification of this design point and the resulting DroidSafe implementation. We attribute the ability of DroidSafe to operate at this design point to: 1) the identification of a set of techniques that work well together, 2) new implementations of known program analysis techniques that enable DroidSafe to deliver an analysis of unprecedented scalability, accuracy, and robustness, 3) a set of new mechanisms that enable these techniques to work together to provide a comprehensive, accurate, and precise information-flow analysis for Android applications, and 4) significant engineering effort that delivers a comprehensive model of the Android runtime. Specific contributions include:

- **Accurate Analysis Stubs**: A novel technique that enables the rapid and accurate development of semantics missing from a source code base. Each stub is written in the language of the implementation of the API model, simplifying analysis. Stubs augment the implementation with semantics possibly incomplete for the full runtime behavior, but complete for the analysis abstractions.
- **Android Device Implementation**: A comprehensive and precise model of the Android API and runtime system implemented in Java that accurately captures the semantics of life-cycle events, callback context, external resources, and inter-component communication. The core of the ADI includes 550 manually-verified Android API classes which cover over 98% of API calls in deployed Android applications. The ADI currently models Android 4.4.3, because updating the model for Android updates is not overly burdensome. Independent analysis tools can readily employ this model.
- **Static Analysis Design Decisions**: Our analysis occupies a new design point for information-flow analysis of Android: deep object sensitivity and flow insensitivity. Flow insensistivity enables DroidSafe to accurately consider all possible event orderings. It also enables DroidSafe to scale to analyze an accurate and precise model of the Android environment, which is critical for the overall success of DroidSafe. Any loss of precision due to flow-insensitivity is more than compensated for by the analysis's ability to scale to analyze our accurate and precise Android model.
- **Static Communication Target Resolution**: A comprehensive and precise model of inter-component communication resolution in Android that links data flows between sender and target. DroidSafe includes a global `Intent` and `Uri` value resolution analysis, `IntentFilter` reasoning, and coverage of all common forms of communication. To our knowledge it is the most complete such model to date.
- **Experimental Evaluation**: An evaluation demonstrating that 1) DroidSafe achieves unprecedented precision and accuracy for the information-flow analysis of Android and 2) DroidSafe can detect malicious sensitive information leaks inserted by sophisticated, independent hostile organizations, where a current state-of-the-art information-flow analysis largely fails.
- **Full Implementation**: A full open-source implementation of DroidSafe and our ADI available upon request.

II. Background and Problem

Android applications are implemented in Java on top of the Android API. The implementation of an application specifies handlers for the dynamic events that may occur during the execution of the application. Thus, Android applications are dynamic and event-driven by nature. Applications have multiple entry points, and interact heavily with the Android API via utility and resource access classes. The package for an Android application represents an incomplete program; the source package alone is not appropriate for analysis without an accompanying model of the Android API and runtime semantics to exercise all possible semantics in the application.

The Android API version 4.4.3 includes over 3,500 classes visible to an application developer. Analyzing the complete source code for the API is exceedingly difficult because it is implemented over multiple languages and some of the implementation is device-specific. Thus, static analysis frameworks rely on modeling the Android API semantics. Manually producing summaries for all of the application-visible methods of the Android API is daunting task that is potentially error prone. For a high-precision analysis, it is also exceedingly difficult to model all semantics of the implementation regarding memory allocation, data flows, and aliasing. A blanket policy for generating flows for all API methods would risk being too imprecise and inaccurate.

A. Event Dispatch and Ordering

An accurate model of the event dispatch and ordering must represent all valid event orderings so that a static analysis can accurately capture possible runtime behavior. Otherwise, an attacker can hide flows in semantics not covered by the model. Android applications are composed of multiple components, each implementing one of four classes: `Activity`, `Service`, `BroadcastReceiver`, and `ContentProvider`. Each of these components has its own life-cycle defined with events for which a callback implementation can be provided. For example, Figure 1(a) provides an example of a single `Activity` that defines two life-cycle events. These events have the potential to run in many orders, and they are not called directly in application code. There exists a leak of sensitive information in one possible ordering, if `onCreate` is dispatched after `onStop`. This is possible if the activity is placed in the background by user interaction, and not reclaimed by the system before it is reactivated by the user.

In addition to life-cycle state orderings, components can have different launching modes that specify whether a single object should handle all activations or if a separate object is spawned for each activation. Thus memory could be shared across separate activations of a component.

B. Callback Context

An Android application defines callback handler methods that are called for dynamically-dispatched runtime events. Many event handler methods include arguments passed by the runtime to the application for processing. These arguments are generated by the runtime and could include data from the application (including tainted data), depending on the execution sequence prior to the event. We call the arguments to a callback handler its *callback context*. Figure 1(b) gives example of a flow through callback context. This example employs an `Activity`'s ability to save state when it is paused, and restore that state when resumed. An accurate model must represent these possible flow connections (of which there are possibly thousands). Policies such as injecting taint for *all* callback handler arguments or connecting callback argument flows conservatively risk generating an overwhelming number of false positives (see Section VIII).

C. Inter-component Communication (ICC)

The Android framework relies heavily on inter-component communication (ICC) to allow individual components to be independent and to better manage resources. Components initiate and connect to other components via `android.content.Intent` objects (which can themselves contain a payload). The resolution of `Intent` destination is complex [20]. An `Intent` can specify a class explicitly, or implicitly allow the Android system to select a destination based on a `Uri` and string fields. Components register for implicit `Intent` delivery programmatically or via their application's XML manifest. Service components additionally allow one to send and receive messages and perform remote procedure calls.

An accurate model of Android must represent the possible flows via ICC mechanisms. Figure 1(c) gives an example of three components that communicate via `Intent` objects and Service messages. In the example, there is a flow through ICC from `ICCSource` through `ICCService` to `ICCSink`. In addition to representing the communication, a model must consider all possible orderings of component activations.

A blanket conservative policy to deliver `Intent` objects and messages to all possible targets may not be acceptable because applications are typically constructed of many components. However, statically calculating the destination of each `Intent` requires resolution of `Intent` values such as `Uri` strings and action strings, and reasoning about components' implicit `IntentFilter` registration.

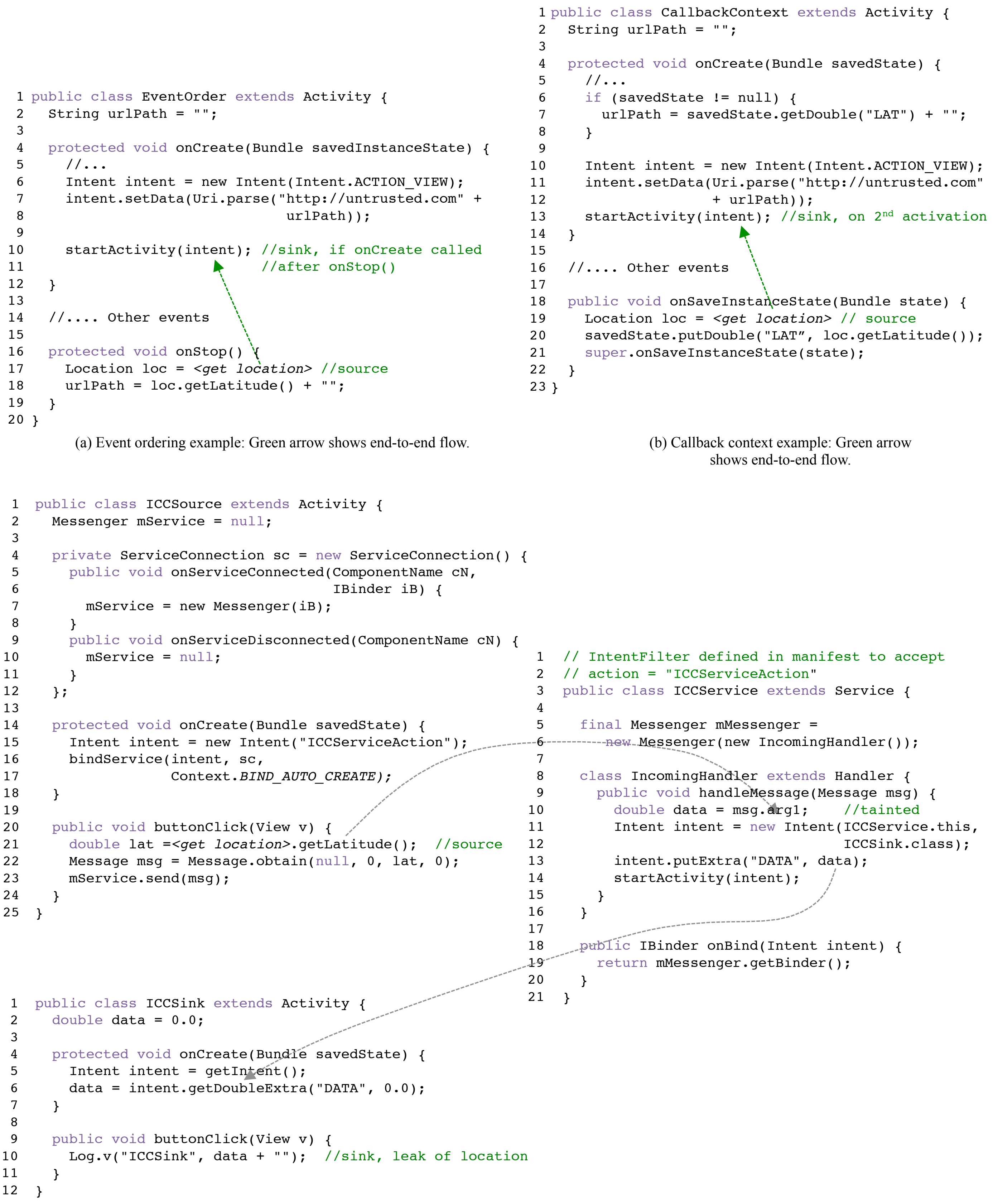
III. Threat Model and Limitations

In our scenario the application developer (or re-packager) is malicious. This attacker seeks to exfiltrate *sensitive data* from a mobile device to her servers or to an area on the device that is unprotected so that a colluding application can perform the exfiltration.

Our definition of sensitive data includes unique device ID, sensor data (location, acceleration, etc.), file data, image data and meta-data, email and SMS messages, passwords, network traffic, and screen-shots. All of these data items are retrieved or stored via the Android API; we define sources of sensitive data as flows initiated from calls to Android API methods that we have identified (see Section IV).

The attacking developer intentionally routes sensitive data to a destination (on or off the device) that is not authorized by the user. We define sinks as Android API calls that may exfiltrate data beyond the application boundaries. Sinks include network, NFC, file system, email or SMS message, or directly to a colluding application via ICC or RPC. All of these sinks are guarded by the Android API. Sinks are identified as described in Section IV.

DroidSafe protects against explicit sensitive information exfiltration by tracking sensitive source to sink flows present in the application. DroidSafe analyzes an applications before it is placed on an app store or before device install. Not every flow reported by DroidSafe is malicious; maliciousness depends on the intent of the developer and the security policies of the user or organization. Thus, the user or a trusted entity reviews the information flows for malicious leaks.



```
public class EventOrder extends Activity {
  String urlPath = "";

  protected void onCreate(Bundle savedInstanceState) {
    //...
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setData(Uri.parse("http://untrusted.com" +
                             urlPath));

    startActivity(intent); //sink, if onCreate called
                           //after onStop()
  }

  //.... Other events

  protected void onStop() {
    Location loc = <get location> //source
    urlPath = loc.getLatitude() + "";
  }
}
```

(a) Event ordering example: Green arrow shows end-to-end flow.

```
public class CallbackContext extends Activity {
  String urlPath = "";

  protected void onCreate(Bundle savedState) {
    //...
    if (savedState != null) {
      urlPath = savedState.getDouble("LAT") + "";
    }

    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setData(Uri.parse("http://untrusted.com"
                   + urlPath));
    startActivity(intent); //sink, on 2nd activation
  }

  //.... Other events

  public void onSaveInstanceState(Bundle state) {
    Location loc = <get location> // source
    savedState.putDouble("LAT”, loc.getLatitude());
    super.onSaveInstanceState(state);
  }
}
```

(b) Callback context example: Green arrow shows end-to-end flow.

```
public class ICCSource extends Activity {
  Messenger mService = null;

  private ServiceConnection sc = new ServiceConnection() {
    public void onServiceConnected(ComponentName cN,
                                   IBinder iB) {
      mService = new Messenger(iB);
    }
    public void onServiceDisconnected(ComponentName cN) {
      mService = null;
    }
  };

  protected void onCreate(Bundle savedState) {
    Intent intent = new Intent("ICCServiceAction");
    bindService(intent, sc,
                Context.BIND_AUTO_CREATE);
  }

  public void buttonClick(View v) {
    double lat =<get location>.getLatitude();  //source
    Message msg = Message.obtain(null, 0, lat, 0);
    mService.send(msg);
  }
}
```

```
// IntentFilter defined in manifest to accept
// action = "ICCServiceAction"
public class ICCService extends Service {

  final Messenger mMessenger =
     new Messenger(new IncomingHandler());

  class IncomingHandler extends Handler {
    public void handleMessage(Message msg) {
      double data = msg.arg1;    //tainted
      Intent intent = new Intent(ICCService.this,
                                 ICCSink.class);
      intent.putExtra("DATA", data);
      startActivity(intent);
    }
  }

  public IBinder onBind(Intent intent) {
    return mMessenger.getBinder();
  }
}
```

```
public class ICCSink extends Activity {
  double data = 0.0;

  protected void onCreate(Bundle savedState) {
    Intent intent = getIntent();
    data = intent.getDoubleExtra("DATA", 0.0);
  }

  public void buttonClick(View v) {
    Log.v("ICCSink", data + "");  //sink, leak of location
  }
}
```

(c) Android ICC Example: ICCSource sends message with tainted data to ICCService. ICCService forward data to ICCSink. Intermediate flows shown in grey arrows.

Fig. 1. Examples of the challenges present in the static information flow analysis of Android applications.

A. *Limitations*

We assume the device has not been rooted, and dynamic code loading is not present in the application. We do not aspire to detect leaks of sensitive data via side channels or implicit flows [21]. Our trusted computing base on a device is the Linux kernel and libraries, the Android framework, and the Dalvik VM.

DroidSafe's reporting is defined by the source and sink calls identified in the Android API. An attacker could exfiltrate API-injected information that is not considered sensitive by DroidSafe, or via a call that is not considered a sink; and it would not be reported.

Our analysis does not have a fully sound handling of Java native methods, dynamic class loading, and reflection. However, we compensate for these idioms with aggressive best-effort policies and analyses. Our analysis has a blanket flow policy for native methods of an Android application, but an application could inject a sensitive flow in a native method, and DroidSafe would not report it. We attempt to aggressively resolve reflection targets in a fashion similar to [22] and [23], but if a *reflected invoke* cannot be resolved, we inject a special REFLECTION taint on the method's arguments and return value (injected by DroidSafe). Thus we could miss a flow injected in an unresolved reflected call.

Finally, we intend the DroidSafe ADI to accurately reflect the runtime semantics of Android with respect to the information flow and points-to information. While we believe we largely cover the semantics, given the size of the Android runtime and API we acknowledge that there may be some methods whose semantics the current ADI does not fully reflect. Different versions exist of Android, and we analyze an application in the context of Android 4.4.3.

IV. DroidSafe's Android Device Implementation

To our knowledge, our model of Android represents the most complete, accurate, and precise Android execution model suitable for static analysis. We accurately and precisely model complexities such as callback context, life-cycle events, data flows, heap object instantiations, native methods, and aliasing. Our Android model is expressed in a single language, standard Java, matching the source language of Android applications, and is appropriate for many existing analysis techniques. One could think of our Android model as a software implementation of an Android device; along with the application and harness. Thus we call our model the *Android Device Implementation* (ADI).

The ADI is a simplified (thus easier to analyze) model of the actual Android system that, with respect to our analysis, represents a best-effort over approximation of the possible behaviors of the real system. The combination of an application, our ADI core, our harness, the semantic transformations for ICC (see Section VI), and resources (unique for each application), creates a closed application for analysis of an Android application.

A. *ADI Core*

We seeded our ADI with the Java implementation of the Android API available from the Android Open Source Project (AOSP) [15], version 4.0.3, along with additional open-source libraries upon which the AOSP implementation depends. This created a code base with no missing dependencies that could be compiled. This code base was approximately 1.3 MLOC, however it was missing substantial portions of the semantics of the Android API and runtime such as native methods, event firings, callback initiation, and component life-cycle events. Furthermore, many commonly used classes included Java implementations that present difficulties for a static analysis.

We therefore developed a novel technique, *accurate analysis stubs*, to enable the effective analysis of code whose full semantics lies outside the scope of AOSP. Each stub is written in Java and only incompletely models the runtime behavior of the modeled code. But the semantics of the stub is complete for the abstractions that the analysis deploys (in this case points-to and information-flow analyses).

We added accurate analysis stubs for 3,176 native methods to model the data flow, object instantiation and aliasing of the missing native code. This was accomplished through a combination of automated and manual means, though all methods were reviewed manually. We developed concrete implementations of 45 classes for which concrete implementations are left to closed-source, commercial libraries.

We simplified the implementation of 117 classes in the Java standard library and Android library to increase precision and decrease analysis time. Examples include container classes, component classes, I/O classes, primitive wrapper classes, strings, and threading classes. We attempted to faithfully maintain the semantics of the original code with respect to its contract with an Android application and a flow-insensitive analysis. The base AOSP plus our additions and modifications enable our ADI to accurately and precisely track flows through the API.

B. *Event and Callback Dispatch*

We created a runtime implementation hooked into the API implementation that models component creation, shared and saved state, life-cycle event firing and argument context, and callback event firing and argument context.

For callback handlers, we implement the callback registration method to invoke the application's callback handler method with the appropriate arguments. For example, Android defines the ability for a component to register to be notified if a database has changed, and handle this change in a given method in a new thread. The application will define a callback handler object, and register this to be notified of database changes. The ADI implements this registration method via a stub that creates the thread directly, and calls the callback method on the registered database. Since our analysis is flow insensitive and our harness is wrapped in a loop (see below), DroidSafe considers all event orderings even though the stub API method invokes the callback handler method directly from the callback registration method (with the appropriate context).

For arguments to callback handlers that are generated by the runtime system, our model creates a new object and passes it to the registered callback handler in the app. For example, to model a key press, our runtime system will create a new object to represent the key press, and call the callback handler with this object on each component.

We developed a separate package for implementing component creation and life-cycle event modeling. This package contains stubs for registration methods for each Android component type: `Activity`, `Service`, `BroadcastReceiver`, and `ContentProvider`. The harness (discussed below) instantiates each application component and passes the component object to the appropriate registration method. The registration methods model shared preferences, saved state, global context classes, and device configuration. This context is passed accurately to the life-cycle events of components.

Since our runtime system makes explicit calls to all life-cycle events of each component, a flow-insensitive analysis can capture the flow between the two life-cycle events in the component in Figure 1(a). Also, since we accurately model saved state through the API and back into a callback handler, our model enables an analysis to report the flow in Figure 1(b).

C. Identifying and Classifying Sources and Sinks

We manually identified 4,051 sensitive source methods and 2,116 sensitive sink methods in the Android API. We also classified each source and sink with a high-level classification (e.g., location, device ID, file, network, and database) so that analysis results can be grouped for verification or consumption by a human. For example, a flow reported by the tool might be: "Location data can flow to the network."

Initially, we tested SuSi, a tool that automatically identifies sink and source methods in the Android API [24]. The automatically identified sources and sinks were incomplete. We compared our identifications of sources and sinks with the results of SuSi.[1] and found that SuSi is missing hundreds of important sources. For example, SuSi did not identify 53% of source calls as "sensitive sources" and 32% of sink calls as "sinks" for the malicious flows in the APAC malicious applications (see Section VIII). These missing sources and sinks indicate the challenge of automatic identification.

D. ADI Coverage and Keeping Current with Android Updates

The core of our model includes 550 commonly-used Android API implementation classes. We manually reviewed, added accurate analysis stubs, and verified these 550 classes. For verification, we manually confirmed that the class implementation is not missing semantics for data flow, object instantiation, and aliasing; and that event callbacks defined in the classes are called explicitly by our model (with the proper context). For classes not in our core set, we still maintain high accuracy because we analyze the actual Java implementation (with accurate analysis stubs), however we may experience a higher level of imprecision for these classes if their implementation is complex.

To measure the coverage of the ADI, we acquired a list of Android API method call frequencies accumulated over 95,910 Android applications downloaded from the Google Play Store. This list reports the number of invoke expressions to each Android API method over all the applications. Calls to the core 550 verified classes account for 98.1% of the total calls over these applications.

[1]We used the source and sink lists for Android 4.2 in the SuSi public repository under the directory `SourceSinkLists/Android 4.2/SourcesSinks`.

We initially seeded the ADI with the AOSP version 4.0.3, and verified the core based on this version. We have since upgraded our model to Android 4.4.3. This process included reviewing changes to classes in our core 550 classes between these versions; and accounting for and verifying any changes in the ADI. This process required one person-week of work, for an experienced Java and Android developer. For the update, the rest of the ADI classes were copied over from AOSP 4.4.3, and accurate analysis stubs were created for native methods. This process required another person-week. We expect the update process for our ADI to continue to be relatively fast since there are few changes to the core of Android between version updates; historically new implementation has been contained in new packages.

E. Harness

Each analyzed application must be *hooked* into the ADI via a harness. In its first pass, DroidSafe generates this harness method automatically. DroidSafe scans the application source code for all classes that subclass one of Android's four component types. It instantiates objects in the harness for all such classes found. We cannot rely solely on the Android manifest for the complete list of components, since the manifest is required to list only components that are exported and available to other applications. We represent each component with a single heap object to account for the complexities of the Android component memory model (see Section II-A). In our harness, each instantiated component object is passed to the appropriate runtime method to exercise all of its life-cycle events. The harness method is wrapped in a loop; the loop is present to capture all possible orderings for callbacks that are called in the harness.

Though details are beyond the scope of this paper, the harness also includes instantiation of GUI objects defined in XML resources (including programmatically setting their attributes, and registering event handlers, such as *onClick*), and the incorporation of String values defined in XML resources.

V. Object-Sensitive Points-To Analysis

Points-to analysis (PTA) is a foundational static program analysis that computes a static abstraction of all the heap locations that a pointer (reference) variable may point to during program execution. In addition to the points-to relation, points-to analysis also constructs a call graph as modern languages require points-to results to calculate targets for dynamic dispatch and functional lambda calculations. Our goal was to employ much of the AOSP Android API implementation without modification, and achieve precise results for our client analyses. However, as with many static analyses there is a trade-off between scalability and precision; appropriate control-flow and data-flow abstractions must be chosen to avoid intractability and to calculate acceptably precise results.

Let us consider the difficulties of analyzing complex Java code with precision. Figure 2 lists simplified ADI source code for two commonly used classes in the Android API: `android.os.Bundle` and `java.util.HashMap`. `Bundle` allocates a `HashMap`. The example also provides relevant code for two Android activities that each create a `Bundle` and store values to their `Bundle`; `Activity1` puts non-sensitive data in its

Android Device Implementation (ADI)

```
1  package android.os;
2  public class Bundle ... {
3    private Map<String,Object> mMap =
4                  new HashMap<String,Object>(); Ⓗ
5
6    public void put(String k, Object v) {
7      mMap.put(k,v);
8    }
9
10   public Object get(String k) {
11     return mMap.get(k);
12   }
13 }
```

```
1  package java.util;
2  public class HashMap<K,V>... {
3    private Entry[] table = new Entry[size]; Ⓣ
4
5    public void put(K key, V value) {
6      ...
7      table[index] = new Entry<K,V>(key, value); Ⓔ
8    }
9
10   public V get(Object key) {
11     ...
12     e = table[indexFor(hash, table.length)];
13     ...
14     return e;
15   }
16 }
```

Android Application Source Code

```
1 public class Activity1 extends Activity {
2   ...
3   Bundle bundle1 = new Bundle(); Ⓝ
4   bundle1.put("data", <notSensitive>);
5   ...
6   sink(bundle1);  //not a sensitive flow
7 }
```

```
1 public class Activity2 extends Activity {
2   ...
3   double sensitive = location.getLatitude(); //source
4   Bundle bundle2 = new Bundle(); Ⓢ
5   bundle2.put("data", sensitive);
6   ...
7   sink(bundle2);  //flow of sensitive -> sink
8 }
```

Fig. 2. Example source code for our ADI and two `Activity` objects illustrating the challenges of points-to and information flow analysis.

`Bundle` while `Activity2` puts sensitive data in its `Bundle`. Consider the difficulty presented to an analysis given this code. To precisely separate the two `Bundle` objects created ($\textcircled{N}$ and $\textcircled{S}$), a PTA must separate multiple levels of allocations started at each `Bundle` allocation (`Bundle` allocates a `HashMap`, $\textcircled{H}$, which allocates an array to store entries, $\textcircled{T}$). In other words, an analysis must be able separate analysis facts between the array of entries created in the two `HashMaps` objects of the two `Bundle` objects in this code. Otherwise, sensitive data put in one `Bundle` is conflated with data that can be retrieved from the other `Bundle`, decreasing precision.

Our PTA algorithm is based on a whole-program, flow-insensitive, subset-based foundation [25] for Java on which we have added *context sensitivity*. Context sensitivity is a general approach where a PTA is able to separate analysis facts for a method m that arise at multiple call sites of m. There are multiple choices for context, and the DroidSafe PTA implements *object sensitivity*. Accumulating evidence demonstrates that object sensitivity is the best choice for object-oriented languages [18, 26–28].

Object sensitivity is notoriously difficult to understand and implement [18]. Here we give the reader an intuitive description of object sensitivity and its scalability challenges. For a rigorous description of object sensitivity see [18] (note that our PTA implements a 3Full+2Heap analysis modified as described below). An object-sensitive analysis uses object allocation sites (`new` expressions in Java) as context elements. In our analysis, a heap object, o, is represented by the allocation site of o, plus the allocation site of the object that allocated o, and so on, to a parameterized depth, k. For a given method, our analysis is able to separate facts depending on the heap object of the receiver on which the method is called.

Considering again the example in Figure 2, our analysis is able to separate analysis facts calculated for the array `table` of two `HashMap` objects allocated from the two `Bundle` objects. When the `Bundle` objects are created on line 3 and line 4 of `Activity1` and `Activity2`, respectively, a series of allocations is performed via constructors. The object-sensitive heap abstraction will have two separate elements representing the two `table` arrays, with context being their allocation history:

$$\langle \textcircled{T} \leftarrow \textcircled{H} \leftarrow \textcircled{N} \rangle$$
$$\langle \textcircled{T} \leftarrow \textcircled{H} \leftarrow \textcircled{S} \rangle$$

Where $a \leftarrow b$ denotes "a allocated in b".

The `new` expression on line 3 of `Activity1` creates a heap object $\langle \textcircled{S} \rangle$, and the cascading allocations from the constructors of `Bundle` and `HashMap` create $\langle \textcircled{H} \leftarrow \textcircled{S} \rangle$ and $\langle \textcircled{T} \leftarrow \textcircled{H} \leftarrow \textcircled{S} \rangle$, respectively.

When the `Bundle.put(...)` method of `Activity2` is called (line 5), the method context (receiver) for the call is $\langle \textcircled{S} \rangle$, this triggers a call to `HashMap.put(...)` (line 7 of `HashMap`) with context $\langle \textcircled{H} \leftarrow \textcircled{S} \rangle$. In this context, the points-to set result for the reference to the field `table` on line 7 of `HashMap` is the array object $\langle \textcircled{T} \leftarrow \textcircled{H} \leftarrow \textcircled{S} \rangle$. Any elements placed in this array via the assignment of line 7 will only be reflected in this array heap object.

Thus, for the example, our PTA is able to separate analysis facts between the two `Bundle` objects in the two Activities via deep object-sensitivity. For this example, the analysis requires a heap context depth of 3 (to distinguish $\langle \textcircled{T} \leftarrow \textcircled{H} \leftarrow \textcircled{N} \rangle$ from $\langle \textcircled{T} \leftarrow \textcircled{H} \leftarrow \textcircled{S} \rangle$).

Object sensitivity has powerful precision but scalability presents a challenge. Our example requires a depth of 3, and other commonly used classes require deeper depths, e.g., in the AOSP implementation, `Intent` allocates a `Bundle` (that allocates a `HashMap`...) requiring a depth of 4 to disambiguate the items placed in the `Bundle` of two `Intent` objects. We tested other whole-program object-sensitive frameworks, but found they could not scale to the required context depth [29] or did not maintain program information required for our client [18].

To solve this scalability challenge, our points-to analysis implementation operates on the pointer assignment graph (PAG) representation of the program [30], an explicit representation of the program. In the past, while explicit implementations provided the fastest running times, they would typically exhaust main memory for large programs [29, 31]. However, today, with the large and increasing size of available main memory, we found that an explicit implementation can now scale to large programs.

Furthermore, our implementation is flexible and parameterized. This flexibility enables us to implement a series of client analysis-specific and Android-specific optimizations of our object-sensitive points to analysis. Without our optimizations 3 of the 24 APAC applications (see Section VIII) could not finish analysis in DroidSafe given a limit of 64GB of heap memory. With optimizations, all applications now run in under 34GB of heap memory. The optimizations provide a savings of 5.1x in total analysis time. We highlight the important optimizations here.

A. Selectively Applying Context

Typically, a points-to analysis keeps the same base context depth for all allocated objects. Initially, we tried this policy, at a depth of 3, for the applications in our APAC suite. However, our analysis would fail on 3 of our APAC applications because it exhausted 64GB of allocated heap memory. Instead we implement a targeted approach: we add context for an abstract heap location at the minimum depth that is required to achieve precision for one of our client analyses. The depth of context on abstract heap objects varies from 0 to 4. The context depth is calculated based on the following.

By analyzing a suite of 211 Android applications for which we had source, we learned which API classes from our ADI could be analyzed content-insensitively without significant loss of precision for clients of the PTA. We performed our points-to analysis on our suite, and determined API classes that could never reach (via a series of local or field references) a String value our `Intent` resolution analysis was tracking, and could never reach a value that was tainted with sensitive information flow (across all Android applications in our suite).

This set, S, contains 1489 Android API classes. Most of these classes are Android GUI objects and libraries through which sensitive data should never flow. For all $c \in S$, our clients analyses will calculate the same results regardless of whether an object of c in the heap abstraction of our PTA has context or not (for the 211 Android applications analyzed). This set represents 26% of the total classes of our ADI. We extrapolate that context-insensitivity analyzing the classes of S in our will give us an acceptable loss of precision across *all* Android applications.

We modified our PTA to never attach context to an allocation of or a method call on an object of a class in S. This means that for a method m called on $c \in S$, the information-flow analysis analyzes m without context, conflating the analysis results of all calls to m on objects of c. For information-flow analysis, this relaxation means that, in m, if a sensitive taint flows to a field f of an object of a class c, the taint will (imprecisely) flow to f for all heap objects of class c.

Conversely with a maximum context depth of 3, our points-to analysis is unable to disambiguate many important analysis facts. For example, we could not disambiguate the `Bundle` between separate `Intent` objects (as discussed above). To address this issue, we automatically increase the context depth to 4 for all heap objects of Array type. In the example of Figure 2, this means that the array object allocated at line 3 of `HashMap` has a depth of 4, giving it enough context to disambiguate the `Intent` object that allocated the `Bundle` that allocated the `HashMap` that allocated the array. This general strategy works across containers in the Java and Android API packages.

VI. Improving the Precision of ICC Modeling

Inter-component communication (ICC) is common in Android applications and must be modeled both accurately and precisely. However, the AOSP implementation of ICC-related classes is incomplete (relying on native methods for target resolution and payload delivery). To achieve precision, we implement our own model of ICC via accurate analysis stubs, aggressively resolve dynamic program values, and transform application code to increase precision.

`Intent` objects describe ICC destinations. Values involved in the resolution include both `java.lang.String` and `java .lang.Class` objects. We resolve these values statically to guide precision-enhancing transformations. Our ADI provides us with an accurate and precise model for resolution analysis of values involved in `Intent` resolution.

Strings are an essential base value type of many of the `Intent` fields. To resolve string values given the myriad operations performed on strings, we employ the JSA String Analyzer (JSA) [32]. JSA is a flow-sensitive and context-insensitive static analysis that includes a model of common operations on Java's `String` type. For a given `String` reference, the analysis computes a multi-level automaton representing all possible string values. As a first pass, we run JSA (on only the application source) to resolve values for string references that are arguments to Android API calls. We convert each resolved automaton to a regular expression that represents the possible values of the string value.

After JSA is run, we replace resolved string values in the application code with constants representing their computed regular expression, and perform a pass of our points-to analysis such that these values can be propagated globally. We run our points-to analysis and store the results of this analysis for all string references in the program, such that later we can query the resolved regular expressions representing values for all string references in application code.

A. Resolving Explicit `Intent` Destinations

Explicit `Intent` objects are initiated with the destination component's fully-qualified class name or class constant object. Before the PTA is run, each class constant passed to a

Source Method	Target Method Call Injected
`Context: void send*Broadcast(Intent, ...)` [*6 variants*]	`BroadcastReceiver: void onReceive(Intent)`
`Activity: void startActivit*(Intent, ...)` [*6 variants*]	`Activity: void setIntent(Intent)`
`Context: void bindService(Intent, Connection)`	`Service: void droidSafeOnBind(Intent, Connection)`
`Context: void startService(Intent)`	`Service: void onStartCommant(Intent, ...)`
`ContentResolver: insert, query, delete, update`	`ContentProvider: insert, query, delete, update`

Fig. 3. DroidSafe's ICC source to target methods transformations.

method of `Intent` is converted into a component name string constant representing the class. To determine the destinations of an `Intent` object in our abstract heap, we query the points-to information for the fields of component name. If all of the strings objects in the points-to set are constants, we consider the `Intent` object resolved.

B. Resolving Implicit `Intent` *Destinations*

Implicit `Intent` objects are `Intent` objects for which a component name is not specified; in our analysis these are `Intent` objects for which the component name field is null. Implicit `Intent` objects do not directly reference a destination but instead leave it to the Android system to deliver them to the appropriate destination(s). A component registers as a destination of implicit `Intent` objects by declaring `IntentFilter` elements in the manifest or programmatically installing `IntentFilter` objects on components. `IntentFilter` registrations specify string constants that the component will accept for the action, category, data type, and uniform resource identifier (`Uri`) fields of a dispatched `Intent`. We parse the Android manifest, and keep a map of implicit `Intent` registrations, i.e., for each component the implicit `Intent` field values it accepts. We also support updating this map with programmatic registrations for `BroadcastReceiver` objects, by modeling `IntentFilter`.

An implicit `Intent` object in our abstract heap is resolved if our PTA concludes that the points-to set for one of the action, category, data type, or `Uri` fields reference only constants (or is empty).

For a resolved implicit `Intent`, i, we build i's list of in-app targets by comparing to each component's intent filter. For component c, we test the action, category, data type, and `Uri` fields in sequence. For each field, if i's field is unresolved, then the test passes. If the field of i is resolved, then if any of its string constants are in the set of strings accepts by c's intent field for the field, then it is a match. All fields of i have to pass the test against the respective fields of c's intent filter for i to be able to target c.

There is additional complexity for programmatic intent filters, as DroidSafe may not be able to resolve all fields of an intent filter to constants. An intent filter field that cannot be resolved matches the respective field for all Intents.

C. Transforming ICC calls to Improve Precision

ICC initiation calls are methods that pass an `Intent` to Android's runtime system to perform inter-component communication or binding. Our strategy for improving precision for ICC is to transform ICC initiation calls into appropriate method calls at the destination(s), thus linking the data flows between source and destination.

Figure 3 presents a list of the most common ICC initiation calls, and the linkage calls that are inserted by DroidSafe to improve precision and accuracy. For example, for an invoke of `startActivity(Intent)`, we transform this call into calls of the destination activities' `setIntent(Intent)`, linking the source and targets. Thus when a target Activity calls `getIntent()`, all `Intent` objects that could possibly be sent to the Activity are calculated by our PTA (we update the PTA result after all ICC transformations are completed).

For ICC initiation calls on resolved `Intent` objects, we link the ICC initiation call to only the destination components that are specified by the `Intent`. This is achieved by calling the appropriate linkage method on the heap object allocated in our harness for the destination component. For unresolved `Intent` objects, we insert linkage calls to all components of the appropriate type. For example, a `startActivity(Intent)` call with an unresolved `Intent` is delivered to all Activity components.

D. Android Services

Android Service components require additional sophistication because in addition to `Intent`-mediating communication, messaging and RPCs can be performed. We illustrate our Service transformations via the examples in Figure 1(c). In this example, the Activity `ICCSource` binds and sends a messages to the Service `ICCService`. The important steps performed by DroidSafe to resolve this flow are as follows:

1) Our manifest parser maps the `Intent` action string "ICCServiceAction" to `ICCService`.
2) DroidSafe resolves the `Intent` object on line 15 as an Implicit `Intent` with action string "ICCServiceAction". Consulting the implicit `IntentFilter` registration map, we see the `Intent`'s destination is `ICCService`.
3) The call to `bindService(...)` on line 16 is transformed to a linkage call to the harness object representing `ICCService`. This linkage call is a new method we define in our ADI for Service, `droidSafeOnBind(Intent, ServiceConnection)`. The linkage method performs the following (some details omitted):
   a) Invoke `ICCService`'s (the receiver's) `onBind(Intent)` method to retrieve the `Binder` object. The ADI model for `android.os.Messenger .getBinder()` creates a `Binder` that references the `Messenger` object which created it.
   b) Invoke `onServiceConnected(ComponentName, Binder)` on the passed `ServiceConnection` object, passing the `android.os.Binder` returned from `ICCService`'s `onBind()` method.

With the linkage methods called, the `Binder` object used to create the `Messenger` in `ICCSource` line 7 is connected to the `IncomingHandler` of line 8 of `ICCService`. The method `android.os.Messenger.send(Message)` has a stub to call the `handleMessage(Message)` method of its `Handler` object. Thus, the call `mService.send(msg)` on line 23 of `ICCSource` will deliver the message to the `handleMessage(Message)` method of `ICCService`'s `Messenger` object.

This is just one example of binding and message communication in Android that we support. The DroidSafe ICC model supports precision increasing transformations for common forms of ICC, and handles uncommon cases conservatively.

VII. Information-Flow Analysis

Our information-flow analysis computes an over-approximation of all the memory states that occur during the execution of a program. The analysis is designed as a forward data-flow analysis. For each type of statement, we define a transfer function in terms of how it changes the state of memory.

We divide memory into four separate areas that store local variables, instance fields, static fields, and arrays, reflecting the semantics of the Java programming language:

$$\begin{aligned}
\text{Memory} &= \text{Local} \times \text{Instance} \times \text{Static} \times \text{Array} \\
\text{Local} &= \text{Ctx} \times \text{Var} \to \text{InfoVal} \\
\text{Instance} &= \text{Loc} \times \text{Field} \to \text{InfoVal} \\
\text{Static} &= \text{Class} \times \text{Field} \to \text{InfoVal} \\
\text{Array} &= \text{Loc} \to \text{InfoVal} \\
\text{Ctx}, \text{Loc} &= \text{AllocSite}^k
\end{aligned}$$

Each of the memory areas is modeled as a function whose codomain consists of a set of *information value*s. An information value is a tuple of the type of information and the source code location where the information was first injected. Our analysis can identify not only the kind of information being exfiltrated but the code location of the source.

In the local variable area, Local, each method's local variables are parameterized by their calling contexts (i.e. the heap location of a receiver object), so the precision of the analysis does not decrease when a method is called in various contexts. In other words, our information-flow analysis analyzes local variables in a flow-insensitive and object-sensitive fashion.

The instance field area, Instance, is a function that takes as its arguments an abstract heap location and an instance field. The return value is information values that flow into the instance field of objects at the heap location. Note that an abstract heap location consists of a series of allocation sites (see Section V). This area corresponds to what is colloquially called "context-sensitive (or object-sensitive) heap" or "heap cloning" in the literature [18]. Each static field of each class has an entry in the static field area, Static. Unlike the memory area for instance fields, the static field area is not parameterized by heap locations because, in Java, all objects of each class share the static fields of the class.

The analysis collects all information values that are assigned to the elements of an array and stores the result at a single heap location of the array area, Array. That is, we analyze arrays in an array-index-insensitive fashion.

An information value is injected into an appropriate memory area when a source API method is invoked from application code. More specifically, for the statement `r = o.source(a)` where `r` is of primitive type, the analysis puts an information value into an entry in the local variable area that corresponds to the `r` variable in the current calling context; the stored information value consists of the statement's location and the type of information associated with the `source` method. If `r` is a reference, the information value is stored in the special `taint` field of an instance that `r` refers to in the current calling context.

For each `o.sink(a, ...)` statement that invokes a sink method, DroidSafe reports the information values for *accessed memory addresses* in the sink. For each argument (and the receiver `o`), DroidSafe reports all the information values that are attached to memory addresses (and their `taint` field) read during the execution of the body of the `sink` method (among memory addresses reachable from the argument).

In Figure 2, the analysis injects an information value into the `sensitive` variable when `Location.getLatitude()`, a source API method, is invoked (line 3 of `Activity2`). The information value consists of the line number and the type of information (for this case, a user's location). For the invocation of `Bundle.put()` method (line 5 of `Activity2`), the transfer functions of the statements in the body of `Bundle .put()` convey the information value from the `sensitive` variable to the `value` field of an `Entry` object whose heap location is $\langle Ⓔ \leftarrow Ⓗ \leftarrow Ⓢ \rangle$. At the call to a sink API method (line 7 of `Activity2`), the analysis reports 1) a user's location information is reachable from the `bundle2` argument, 2) the information was generated first at line 3, and 3) whether the body of the sink method actually uses the information by reading the `Entry.value` field. On the other hand, for the call to a sink API method in `Activity1` (line 6 of `Activity1`), the analysis correctly reports that a user's location information is not reachable from the `bundle1` argument. That is, even though the `Entity` objects in `Activity1` and `Activity2` are both created at the same program location Ⓔ, the analysis properly distinguishes information flows into them because each of them has its own heap location ($\langle Ⓔ \leftarrow Ⓗ \leftarrow Ⓝ \rangle$ and $\langle Ⓔ \leftarrow Ⓗ \leftarrow Ⓢ \rangle$, respectively).

VIII. Evaluation

This section presents experimental results that characterize the effectiveness of DroidSafe's information-flow analysis. Our results indicate:

1) DroidSafe achieves both higher precision and accuracy than FlowDroid [8] + IccTA [14] a current state-of-the-art Android information-flow analysis. [8] and [14] demonstrate that FlowDroid + IccTA achieve both higher precision and accuracy than commercially available tools such as IBM's AppScan Source [33] (which was specifically designed to analyze Android apps) and HP's FortifySCA [34].
2) DroidSafe successfully reports all malicious leaks of sensitive information in a suite of malicious Android applications developed by independent, motivated, and sophisticated attackers from three hostile Red Team organizations.
3) DroidSafe successfully scales to analyze large Android applications analyzed in the context of our ADI.

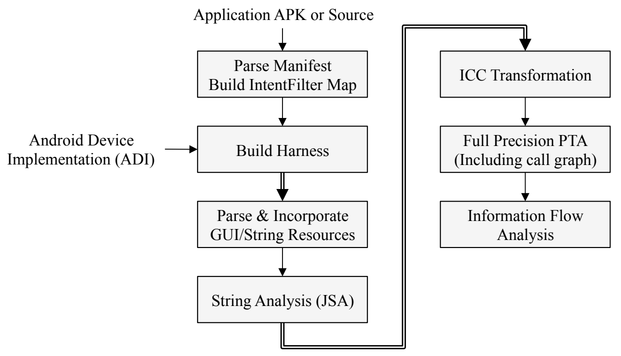


Fig. 4. Phases of the DroidSafe Tool. Double lines denote an update of the PTA result is calculated for the next phase.

Tool	Missed Flows Explicit / Implicit	Accuracy	False Positives	Precision
DroidSafe	0/6	93.9%	13	87.6%
FlowDroid	12/7	80.6%	30	72.5%

Fig. 5. DROIDBENCH results for DroidSafe and FlowDroid.

A. Methodology

DroidSafe is developed on top of the Soot Java Analysis Framework [35]. We implemented our object-sensitive points-to analysis on top of Soot's SPARK PTA framework [30]. DroidSafe comprises approximately 70K lines of Java code. Figure 4 presents the architecture of DroidSafe (previous sections discuss the different phases of DroidSafe). DroidSafe can analyze an APK and Java source code.

We compare DroidSafe's information-flow results to FlowDroid + IccTA [8, 14], a flow-sensitive and object-sensitive static information-flow analysis system developed by academic researchers. This system incorporates Epicc [36] to resolve values for `Intent` objects used in ICC calls. For the remainder of this section we refer to this system as *FlowDroid* since FlowDroid is the underlying information-flow analysis. We run FlowDroid with our list of sources and sinks described in Section IV. Both tools report flows at a fine granularity as the generating source method call expression and the sink method call expression (Soot IR representation of the analyzed application).

We executed FlowDroid and DroidSafe on an Intel® Xeon® CPU E5-2690 v2 @ 3.00GHz running Ubuntu 12.04.5 with 64GB of heap memory for the JVM. We executed on a single core of the processor. We compare results for three application sets:

DROIDBENCH: DROIDBENCH[8] is an evolving set of Android micro-applications designed by the authors of FlowDroid to test precision and accuracy (recall) of static information-flow analyses. For our evaluation of DroidSafe, we employ DROIDBENCH version 1.2 plus the benchmarks developed for IccTA [14], which includes 94 benchmarks. One of the goals of DROIDBENCH is to exercise potentially problematic Java idioms such as exceptions, callbacks, reflection[2], Android Inter-Component Communication (ICC), static initializers, and arrays. It is also designed to test the flow-, field-, and object sensitivity of the analysis. The largest DROIDBENCH application is under 200 lines of code. Reachable code including our analyzed Android ADI is always at least 80,000 lines of code. The maximum FlowDroid analysis time across the benchmark suite is 11 seconds. Despite the fact that it analyzes substantially more code than FlowDroid, the maximum DroidSafe analysis time is 222 seconds.

Additional Android Micro-Applications: To get better test coverage of Android, we developed our own set of 40 information-flow Android micro-applications. These applications test common Java and Android idioms for which coverage is missing in the current version of DROIDBENCH. The largest app is 255 lines of code. Benchmarks include tests of: `Parcel`, `Activity` saved state, `Fragment`, asynchronous event orderings, global `Application` object, String to char conversion, callback context, `SharedPreferences`, and dynamic dispatch precision. Fourteen (14) of the benchmarks test Android ICC mechanism coverage such as: components not in manifest, event ordering between components, programmatic `IntentFilter` registration, `Intent` passed through API objects, and various mechanisms for creating explicit `Intents`. We are working with the developers of DROIDBENCH to incorporate our benchmarks into the DROIDBENCH suite.

APAC: The APAC applications are complete real-world Android applications that, as part of the DARPA Automated Program Analysis for Cybersecurity (APAC) program, have been augmented with information-flow leaks by three independent hostile Red Team organizations. The goal of these organizations was to develop information leaks that would either evade detection by static analysis tools or overwhelm static analysis tools into producing unacceptable results (by, for example, manipulating the tool into reporting an overwhelming number of false positive flows). The APAC applications are unique because they are aggressive malware for which the ground truth for malicious information flows is defined. Furthermore, unlike much of the current generation of malware in the wild, which is delivered in over-privileged applications [37], many of the APAC applications contain malicious leaks to unauthorized destinations of sensitive data that the application is authorized to access based on its stated functionality.

The APAC applications comprise 24 Android applications. The application sizes range from 200 to 80,000 lines of Java code (not including library or Android run-time code). Reachable code including our analyzed Android ADI ranges from at least 80,000 to 180,000 lines. DroidSafe analysis times for the APAC applications range from 4.4 to 27.5 minutes.

B. DroidBench and Additional Micro-Applications

Figure 5 presents results from the DroidSafe and FlowDroid analysis of the DROIDBENCH suite. The second column contains entries of the form X/Y, where X is the number of missed explicit flows across all 94 applications and Y is the number of missed implicit flows. Implicit flows have no direct flow of data from the source to the sink; the flow is instead created via control flow that depends on sensitive data. DroidSafe detects all of the 90 explicit flows in the benchmark suite. FlowDroid detects 78 of 90 explicit flows. Inaccuracies

[2]DroidSafe handles reflection using its String and points-to analysis to replace reflective calls in application code with direct calls to the target method, when applicable. The mechanisms of this transformation are similar to ICC transformation discussed in Section VI.

Application	Lines of Code	Malicious Flow	
		Source	Sink
AgentSmith	1,481	Clipboard	Network
AndroidGame	63,755	Image Metadata	Network
AndroidMap	8,491	Location	Network
AndroidsFortune	14,621	Device ID	Network
AudioSidekick	2,444	Mic	Network
AWeather	1,837	Network	Network
BatteryIndicator	5,319	Image	Network
Butane	2,506	SMS	Network
CalcF	861	User Input	Network
DeviceAdmin2	2,289	System Info	Network
FillInFun	82,602	Contact	SMS
KitteyKittey	962	Image Metadata	Network
PicViewer	221	Image Metadata	Network
Quickdroid	6,155	Contact, Bookmark	IPC
RunningApp	1,785	User Input	NFC
ShareLoc	372	Location	Network
ShyGuyCRM	3,811	Contact	Email
SmartWebCam	1,176	Camera	AIDL
SMSBackup	387	SMS, Image, Browser	File
SMSBlocker	3,775	SMS	Network
SMSPopup	17,953	SMS	SMS
SnapshotShare	13,461	Screenshot	Network
SourceViewer	208	Device ID	Network
UltraCoolMap	2,658	Location	Network

Fig. 6. APAC Information-Flow Applications: Size and malicious flows details.

in the FlowDroid tool cause it to miss the 10 flows: static initializers not modeled properly with respect to flow sensitivity, String to character conversions with unlinked flows, life-cycle event ordering inaccuracies, inaccurate fragments modeling, missing `ContentProvider` flows, and unresolved flows linked through reflected calls.

The fourth column of Figure 5 presents the number of false positives — i.e., the number of reported flows that do not actually exist in the 94 benchmark programs. Despite its greater accuracy, DroidSafe exhibits a lower false positive rate than FlowDroid. The final column presents the corresponding precision numbers for these false positives.

DroidSafe reports 13 false positives. DroidSafe reports two false flows due to conflating all elements of an array with tainted and untainted elements and two false flows due to accesses of a container (Map and List) with tainted and untainted elements. DroidSafe also reports four false flows due to flow insensitivity. We reports two false positives because of our conservative (though accurate) model of life-cycle event orderings; in one benchmark the flow was connected by an order that could never occur at runtime (`onCreate()` called after `onDestroy()`). Two false flows are caused by conflating accesses of the `Intent` state map based on a string key.

Eighteen (18) of the 30 FlowDroid false positives are due to a conservative "Callback" source type that FlowDroid always injects for callback handler arguments because it does not model callback context accurately. Conversely, DroidSafe accurately and precisely models callback context in the ADI, and does not have to conservatively inject these flows. The remaining 12 false positives in FlowDroid are due to various causes such as conflating array and container elements, imprecisions in analyzing callback de-registration, and flow insensitivity on field assignments.

We now present results for the forty additional Android micro-applications developed by the authors. There are 42 leaks of sensitive information in the 40 applications. DroidSafe achieves 100% precision and accuracy for the suite. DroidSafe's comprehensive and precise Android model, the ADI, enables DroidSafe to capture all flows. FlowDroid (plus IccTA) achieves 34.88% accuracy and 79.0% precision. The low accuracy for FlowDroid is caused by the incomplete model of the Android environment that is included in the system. Examples of common Android idioms that FlowDroid currently does not model accurately nor conservatively include: API calls that access state (such as `SharedPreferences` and `Application`); taint transferred from an array index; String to `char` conversion; accounting for all legal asynchronous event orderings; callback context modeling (flows through API callbacks); accounting for components not defined in manifest; event ordering between components; `Service` binding and messages; and programmatic `IntentFilter` registrations.

C. APAC Malicious Applications

We next present analysis results for 24 real-world applications with malicious flows. Figure 6 presents details on the source lines of code and the malicious flows. The most common type of sink is the network (when the flow exfiltrates sensitive information via the network). The developers of the APAC applications attempted to intentionally hide malicious flows. Some of the patterns they used include flows through raised exceptions, application native methods, reflection, and character and string manipulation. They also employ Android API methods that are difficult to model precisely and accurately such as `Object.clone`, `System.arraycopy`, and primitive to string (and vice versa) conversion. One application (SmartWebCam) leaks sensitive information (camera) via an AIDL-defined RPC call that could be called by a colluding application. Android-specific implementation decisions that complicate information-flow analysis include:

1) **ICC**: In 17 of the applications, malicious flows cross components via `Intents`, messages, or RPCs.
2) **Callbacks**: In 17 of the applications, either a) sources or sink calls are reachable via control flow initiated via a callback, or b) malicious flows are linked through callback arguments passed via the runtime.
3) **API Inter-method flows**: In 6 of the applications malicious flows are linked through API method combinations with flows that are challenging to model. More specifically, method a is called to assign a reachable field or static memory location to a tainted value, then method b is called that accesses this memory via static memory or a path of fields from its arguments. `SharedPreferences` are an example: one component could write taint to a preference, and another component could read from that preference.
4) **Non-argument sink flows**: In 7 of the applications, a malicious flow is not linked via the arguments of the sink call. Instead the source taint is accessed beginning with a) memory reachable from a field of the receiver, or b) memory reachable via a static access.

Figure 7 presents information-flow results. The column "Malicious Flows" gives the total number of malicious flows in the application. Many apps leak multiple sources of sensitive data; for example UltraCoolMap leaks location information from four sources to the network.

		DroidSafe						FlowDroid		
Application	**Malicious Flows**	**Reachable Lines (including ADI)**	**Analysis Time (sec)**	**Reachable Source Calls**	**Reachable Sink Calls**	**Total Flows**	**Missed Malicious Flows**	**Analysis Time (sec)**	**Total Flows**	**Missed Malicious Flows**
AgentSmith	1	123,881	434	53	60	167	0	60	123	1
AndroidGame	1	82,170	499	11	18	37	0	Did not complete		1
AndroidMap	2	102,236	698	78	41	132	0	54	25	2
AndroidsFortune	1	130,003	752	72	183	304	0	159	208	0
AudioSidekick	2	126,223	507	62	50	89	0	41	28	2
AWeather	1	126,218	491	35	30	72	0	116	57	1
BatteryIndicator	1	122,132	846	64	135	113	0	106	176	1
Butane	4	173,934	625	73	102	392	0	68	109	2
CalcF	2	117,414	374	11	21	11	0	33	5	0
DeviceAdmin2	2	137,046	358	17	33	5	0	47	6	2
FillInFun	2	123,016	601	22	64	14	0	75	25	1
KitteyKittey	1	110,584	271	4	1	2	0	47	1	1
PicViewer	3	118,019	360	7	3	8	0	20	0	3
Quickdroid	19	119,427	399	103	65	278	0	64	231	19
RunningApp	1	126,629	579	51	34	59	0	75	94	1
ShareLoc	4	119,771	1,051	6	4	7	0	28	7	4
ShyGuyCRM	1	177,853	1,255	105	99	463	0	78	82	1
SmartWebCam	1	126,029	1,649	101	267	21	0	50	30	1
SMSBackup	10	108,317	269	25	7	26	0	20	0	10
SMSBlocker	1	125,531	419	12	105	23	0	42	12	1
SMSPopup	3	149,824	1,477	180	182	918	0	298	304	3
SnapshotShare	1	130,111	590	89	29	108	0	92	71	1
SourceViewer	1	118,943	384	13	11	8	0	23	4	1
UltraCoolMap	4	121,507	407	14	9	12	0	34	42	4
Total	**69**					**3,269**	**0**		**1,640**	**63**

Fig. 7. APAC Information-Flow Applications: DroidSafe and FlowDroid evaluation results.

DroidSafe achieves an accuracy of 100%, reporting all 69 of the malicious flows in the APAC apps. The high accuracy of our Android model enabled this result; the malicious flows in these applications often attempt to exploit Android behavior that is difficult to model. For DroidSafe, the ratio of reported malicious flows to total flows is 2.1%.

FlowDroid, in contrast, misses 63 of the 69 malicious flows, for an accuracy of 8.7%. Reasons for the low accuracy include missing ICC modeling, not considering all valid event orderings (exacerbated by FlowDroid's flow-sensitive analysis), inaccurate modeling of many Android API calls (causing disconnection of flows), and inaccurate callback context modeling. FlowDroid's ratio of found malicious flows to total flows is 0.3%. FlowDroid timed out after 2 hours for the AndroidGame application.

The "Total Flows" column in Figure 7 reports the total number of flows from sources to sinks that the tools report. Even though these flows involve sensitive information, the majority are not malicious as the flow is part of the intended functionality of the application. For example, some legitimate flows in UltraCoolMap send location information via the network to a trusted server. The malicious flows, in contrast, send the location via the network to an untrusted destination.

The APAC applications, as with all unknown applications, do not come with specifications that would enable DroidSafe or FlowDroid to distinguish malicious from legitimate flows. Instead, someone must understand the intended functionality of the application and make a subjective determination of which flows are malicious. Because all of the malicious flows that we report were inserted by the hostile Red Team, there is no doubt that at least these flows are malicious. We did not attempt to analyze all of the remaining reported flows to determine if the flows actually exist in the application or not.

D. `Intent` Resolution in APAC Applications

Section VI presents our ICC modeling and defines *resolved* `Intent` objects. A resolved `Intent` is an `Intent` for which an analysis concludes that at least one of its field values is only a set of constants; resolved values can be used to reduce the number of true target components. Over all of the APAC applications, DroidSafe calculates that there are 213 `Intent`-based communication calls. DroidSafe resolves `Intent` objects in 95.8% of the calls. Of the calls with resolved Intents, 59.2% of calls employ explicit `Intent` objects, and 40.8% calls employ implicit `Intent` objects. Of the resolved `Intent`-based calls, 74.0% target at least one component of the app; DroidSafe concludes that the other 26.0% cannot target a component of the app. On average, across the APAC apps, calls with resolved `Intent` objects that target an in-application component resolve to 1.03 destination components.

FlowDroid relies on Epicc to resolve values for `Intent`. We inspected the output of Epicc to determine the percentage of `Intent` objects that are resolved based on our definition. We ignore the `Uri` data field since Epicc does not reason about `Uri` objects. Epicc finds 177 `Intent`-based ICC calls in the APAC applications, Epicc resolves the `Intent` object arguments for 85.9% of the calls (versus DroidSafe's 95.8%). The lower number of total calls versus DroidSafe may be because of Epicc's model of Android callbacks and reachable code. Causes for unresolved `Intent` objects include lack of support for some explicit `Intent` construction mechanisms and `Intent` objects passed through API methods.

Across the APAC applications, there are 131 `ContentProvider` operations. Of the operations, 66.4% use `Uri` objects that DroidSafe resolves. Of the resolved operations, 35.6% target a components of the application, and each resolved operation targets 1.0 components. Epicc does not resolve `Uri` values, and consequently FlowDroid does not link flows through `ContentProvider` operations.

IX. Related Work

Object-Sensitive Points-To Analysis: For robustness and flexibility, typical whole-program object-sensitive analysis implementations reduce program facts into representations appropriate for general solvers; examples include logic relations [18], constraints [31], and binary decision diagrams [29, 38]. Our implementation differs from these systems in that it operates directly on the pointer assignment graph (PAG) representation of the program [30], an explicit representation of the program. Previous work has demonstrated that direct implementations of points-to analysis problems, when they fit in memory, are typically faster than general solvers [29, 31]. Today main memory sizes are large enough to accommodate our direct implementation of a context sensitive analysis of large programs.

Tuning context-sensitivity of an analysis for precision and scalability has also received much work. *Hybrid* context sensitivity treats virtual and static method calls differently, and in addition to object sensitivity, attempts to emulate call-site sensitivity for static calls [39]. Our analysis implements hybrid context sensitivity by cloning static method calls for calls to application methods, and certain API factory methods. *Type sensitivity* is a form of object sensitivity that merges contexts based on types [18]. We tried type sensitivity for our client, but it did not provide adequate precision. An *introspective analysis* drops context sensitivity from program elements that could blow-up the analysis [19], without regard for precision of the client. In client-driven approaches [40], a client analysis asks for more precision from the points-to analysis when needed. In contrast, our technique pre-calculates the set of classes (and thus allocations and methods calls) for which precision is historically not helpful for our problem.

Information-Flow Security Analysis: DroidSafe follows a long history of information-flow analysis (sometimes called taint analysis) systems for security. Livshits and Lam [41] present an approach for taint analysis of Java EE applications that is demand-driven, uses call-site context sensitivity, and shallow object sensitivity via inlining. TAJ [42] focuses on Java web application and employs a program slicing technique combined with a selective object-sensitive analysis. F4F [16] is a taint analysis for Java applications built on web frameworks that uses a specification language to describe the semantics of the underlying framework.

Focusing on information-flow analysis for Android, FlowDroid [8] is a sophisticated, open-source static information flow analysis for Android applications. FlowDroid's analysis is flow-sensitive, and thus, is more precise than DroidSafe, however the FlowDroid model of Android is not nearly as complete as DroidSafe's. FlowDroid attempts to compensate with inaccurate blanket flow policies on unmodeled API methods. From testing, we discovered that FlowDroid does not accurately model all possible combinations of life-cycle or callback events, demonstrating the difficulty of modeling Android execution in a flow-sensitive system. FlowDroid's analysis is on-demand and flow-sensitive as opposed to DroidSafe. However, each instantiation of the analysis is expensive; in preliminary experiments running FlowDroid with our ADI, the analysis completed only 7 of 24 applications given a 2 hour timeout for each application.

Epicc [36] is a tool that resolves `Intent` destinations in an application. Epicc developed a model of commonly-used classes and methods involved in the Android `Intent` implementation. Their analysis is on-demand and flow-sensitive. The DroidSafe system includes a more comprehensive model of classes and mechanisms used in inter-component and inter-application communication (for example `Uri` and `Service` messages). DroidSafe's resolution can also reason about values created in and passed through API methods.

IccTA [14] combines FlowDroid with Epicc and seeks to identify sensitive inter-component and inter-application information flows. DidFail [43] also combines FlowDroid and Epicc to discover sensitive flows across applications. Though not discussed here, DroidSafe includes an analysis to capture inter-application flows via a database of previously resolved Intent values and reachable source flows. This database is consulted and appropriate flows are injected before information analysis.

There are other many other examples of static information flow analyses for Android. CHEX [7] detects information flow vulnerabilities between components. ScanDal [9] is a static analysis implemented as an abstract interpretation of Dalvik bytecode. CHEX and ScanDal employ analysis with $k = 1$ call-site context sensitivity. SCanDroid [11] resolves data flows between components using a limited model of Android, and conservative flow policies for API methods. LeakMiner [12] tracks flows with a context-insensitive analysis. AndroidLeaks [13] combines both context-sensitive and context insensitive analyses, but models flows through API methods with a blanket policy that reduces precision. DroidSafe includes a more precise analysis and has a more accurate and precise model of the Android API than these other tools.

Dynamic testing and monitoring approaches engender different tradeoffs compared to static analysis. Examples include the sophisticated dynamic taint-tracking tool TaintDroid [5], and Tripp and Rubin [44] who describe an approach for classifying information leakages by considering values that flow through sources and sinks. They do not have issues with reflection and dynamic class loading. But, if employed for triage, they require adequate test coverage. If used for dynamic monitoring they are susceptible to denial-of-service attacks if malware is activated during execution and the application is killed or functionality is disabled. This might be unacceptable for mission-critical applications. Similar to static analysis, they require user-mediated judgment for reported sensitive flows.

DroidSafe's list of sources and sinks was compiled manually. SuSi [24] employs supervised machine learning to automatically designate source and sink methods in the Android API. Merlin [45] is a probabilistic approach that employs a potentially incomplete list of sources, sinks, and sanitizers to calculate a more comprehensive list. Merlin automatically infers an information flow specification for an application from its *propagation graph* using probabilistic inference rules.

While SuSi's list proved incomplete for the APAC applications, Merlin's technique is complementary to ours and a possible next step for helping the results of DroidSafe.

X. Conclusion

Malicious leaks of sensitive information pose a significant threat to the security of Android applications. Static analysis techniques offer one way to detect and eliminate such flows. The complexity of modern application frameworks, however, can pose a major challenge to the ability of static analyses to deliver acceptably accurate and precise analysis results.

Our experience developing DroidSafe shows that 1) there is no substitute for an accurate and precise model of the application environment, and 2) using the model to drive the design decisions behind the analysis and supporting techniques (such as accurate analysis stubs) is one effective but (inevitably) labor-intensive way to obtain an acceptably precise and accurate analysis. As long as there are complex application frameworks, we anticipate that making an appropriate set of design decisions (such as the use of a scalable flow insensitive analysis) to successfully navigate the trade-off space that the application framework implicitly presents will be a necessary prerequisite for obtaining acceptable accuracy and precision.

Our results indicate that the final DroidSafe system, with its combination of a comprehensive model of the Android runtime and an effective set of analyses and techniques tailored for that model, takes a significant step towards the final goal of an information flow analysis that can eliminate malicious information leaks in Android applications.

Acknowledgment

We would like to thank our shepherd, Ben Livshits, and our anonymous reviewers for their insightful comments. This material is based on research sponsored by DARPA under agreement number FA8750-12-X-0110. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] A. P. Felt *et al.*, "A survey of mobile malware in the wild," *Security*, vol. 55, p. 3, 2011.

[2] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *WISEC*, 2012.

[3] A. P. Felt *et al.*, "Android Permissions Demystified," *CCS*, 2011.

[4] N. J. Percoco and S. Schulte, "Adventures in Bouncerland," 2012.

[5] W. Enck, P. Gilbert, B. Chun, and L. Cox, "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones," in *OSDI*, 2010.

[6] A. Reina, A. Fattori, and L. Cavallaro, "A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors," in *EuroSec*, 2013.

[7] L. Lu *et al.*, "CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities," in *CCS*, 2012.

[8] S. Arzt *et al.*, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *PLDI*, 2014.

[9] J. Kim, Y. Yoon, K. Yi, and J. Shin, "Scandal: Static Analyzer for Detecting Privacy Leaks in Android Applications," in *MoST*, 2012.

[10] E. Chin, A. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *MobiSys*, 2011.

[11] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "ScanDroid: Automated Security Certification of Android Applications," Tech. Rep., 2010.

[12] Z. Yang and M. Yang, "Leakminer: Detect information leakage on android with static taint analysis," in *WCSE*, 2012, p. 104.

[13] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale," *Trust and Trustworthy Computing*, 2012.

[14] L. Li *et al.*, "I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis," *CoRR*, 2014.

[15] Google, "Android Open Source Project." [Online]. Available: https://source.android.com/

[16] M. Sridharan *et al.*, "F4F: taint analysis of framework-based web applications," in *OOPSLA*, 2011.

[17] K. Z. Chen *et al.*, "Contextual Policy Enforcement in Android Applications with Permission Event Graphs," in *NDSS*, 2013.

[18] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick Your Contexts Well: Understanding Object-Sensitivity," in *POPL*, 2011.

[19] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: context-sensitivity, across the board," in *PLDI*, 2014.

[20] Google, "Intent and Intent Filters." [Online]. Available: http://developer.android.com/guide/components/intents-filters.html

[21] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'Em, can't live without 'Em," in *ICISS*, 2008.

[22] Y. Smaragdakis, G. Kastrinis, G. Balatsouras, and M. Bravenboer, "More Sound Static Handling of Java Reflection," Tech. Rep., 2014.

[23] B. Livshits, J. Whaley, and M. S. Lam, "Reflection Analysis for Java," in *APLAS*, 2005.

[24] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks," *NDSS*, 2014.

[25] L. O. Andersen, "Program Analysis and Specialization for the C Programming Language," Ph.D. dissertation, U. of Copenhagen, 1994.

[26] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *OOPSLA*.

[27] O. Lhotak, "Program analysis using binary decision diagrams," Ph.D. dissertation, McGill University, Montreal, 2006.

[28] M. Sridharan *et al.*, *Aliasing in Object-Oriented Programming*. Springer Berlin Heidelberg, 2000.

[29] M. Berndl *et al.*, "Points-to analysis using BDDs," *PLDI*, 2003.

[30] O. Lhotak, "SPARK: A Flexible Points-To Analysis Framework for Java," Ph.D. dissertation, McGill University, Montreal, 2002.

[31] J. Kodumal and A. Aiken, "Banshee: A scalable constraint-based analysis toolkit," in *SAS*, 2005.

[32] A. S. Christensen, A. Mø ller, and M. I. Schwartzbach, "Precise Analysis of String Expressions Static Analysis," in *SAS*, 2003.

[33] IBM, "IBM Security AppScan." [Online]. Available: http://www-03.ibm.com/software/products/de/appscan

[34] HP, "Enterprise Security Intelligence." [Online]. Available: http://www8.hp.com/us/en/software-solutions/enterprise-security.html

[35] R. Vallée-Rai, E. Gagnon, and L. Hendren, "Optimizing Java bytecode using the Soot framework: Is it feasible?" *CC*, 2000.

[36] D. Octeau *et al.*, "Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis," in *Usenix Security*, Washington D.C., USA, 2013.

[37] A. Jordan, A. Gladd, and A. Abramov, "Android Malware Survey," Raytheon BBN Technologies, Tech. Rep. April, 2012.

[38] P. Liang and M. Naik, "Scaling abstraction refinement via pruning," *ACM SIGPLAN Notices*, vol. 47, no. 6, p. 590, 2012.

[39] G. Kastrinis and Y. Smaragdakis, "Hybrid Context-Sensitivity for Points-To Analysis," in *PLDI*, 2013.

[40] S. Guyer and C. Lin, "Client-driven pointer analysis," *SAS*, 2003.

[41] V. B. Livshits and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," in *USENIX Security*, 2005.

[42] O. Tripp *et al.*, "TAJ: Effective Taint Analysis of Web Applications," in *PLDI*, 2009.

[43] W. Klieber *et al.*, "Android taint flow analysis for app sets," in *SOAP*, 2014.

[44] O. Tripp and J. Rubin, "A Bayesian Approach to Privacy Enforcement in Smartphones," in *USENIX Security*, 2013.

[45] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification Inference for Explicit Information Flow Problems," in *PLDI*, 2009.