EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework

***Abstract*—A wealth of recent research proposes static data flow analysis for the security analysis of Android applications. One of the building blocks that these analysis systems rely upon is the computation of a precise control flow graph. The callback mechanism provided and orchestrated by the Android framework makes the correct generation of the control flow graph a challenging endeavor. From the analysis' point of view, the invocation of a callback is an implicit control flow transition facilitated by the framework. Existing static analysis tools model callbacks either through manually-curated lists or ad-hoc heuristics. This work demonstrates that both approaches are insufficient, and allow malicious applications to evade detection by state-of-the-art analysis systems.**

To address the challenge of implicit control flow transitions (i.e., callbacks) through the Android framework, we are the first to propose, implement, and evaluate a systematic treatment of this aspect. Our implementation, called EDGEMINER, statically analyzes the entire Android framework to automatically generate API summaries that describe implicit control flow transitions through the Android framework. We use EDGEMINER to analyze three major versions of the Android framework. EDGEMINER identified 19,647 callbacks in Android 4.2, suggesting that a manual treatment of this challenge is likely infeasible. Our evaluation demonstrates that the current *insufficient* treatment of callbacks in state-of-the-art analysis tools results in unnecessary imprecision. For example, FlowDroid misses a variety of leaks of privacy sensitive data from benign off-the-shelf Android applications because of its inaccurate handling of callbacks. Of course, malicious applications can also leverage this blind spot in current analysis systems to evade detection at will. The results of our work allow existing tools to comprehensively address the challenge of callbacks and identify previously undetected leakage of privacy sensitive data.

I. INTRODUCTION

Mobile smart devices, such as smart phones, media players, and tablets, have become ubiquitous. Industry reports the total number of sales of Android-powered smart phones at over six hundred million in 2013 alone [6]. The application ecosystem that developed around the mobile platform is a key contributing factor to the huge success of mobile smart devices. Android users can choose from over one million applications (apps) offered through the official Google Play marketplace. Furthermore, a wealth of alternative sources for Android applications is available for users to choose from. These range from curated stores, such as Amazon's Appstore to less legitimate sources that offer pirated content. The sheer number of mobile applications prompted researchers from academia and industry to develop static analysis techniques that scrutinize these applications for vulnerabilities and malicious functionality.

Android applications always execute in the context of the Android framework — a comprehensive collection of functionality that developers can conveniently use from their applications. The prolific use of the framework poses unique challenges for the analysis of Android applications.

State-of-the-art static analysis systems for Android applications (e.g., [9, 11, 17, 18, 23]) reconstruct an application's control flow graph as part of their analysis. However, imprecision in the control flow graph (CFG) permeates throughout the analysis and can cause false alarms as well as missed detections for malware and vulnerability scanners. One common, but insufficiently addressed cause of such imprecision are *callbacks*. A callback is a method implemented by the application but invoked by the Android framework. This implies that the callback method does not have an incoming control flow edge contained in the application itself. Instead, control is implicitly transferred to the callback by the framework.

Intuitively, the existence of a callback must be communicated to the framework before an invocation can happen. In Android, this can be accomplished in two different ways. The first one consists in defining and implementing a well-known Android component, such as an *Activity*. The callbacks associated with these components are strictly related to their life-cycles, as described in the Android documentation. In this case, the developer would *communicate* to the Android framework the existence of the component by specifying it in the so-called *manifest*, an application's main configuration file. Then, when the application is started, the framework parses the manifest and becomes *aware* of the component, and it can thus properly invoke all the defined callbacks.

The second way through which the existence of a callback can be communicated to the framework is by using a so-called *registration* method. A classic example is the `onClick` callback method that applications bind to a user interface element (e.g., a button) with the `setOnClickListener` registration method. Once the user taps the button, the framework automatically invokes the specified `onClick` callback. Similarly, a call to the `sort` registration method of a `Collection` will result in multiple calls to the `compare` callback method implemented by the `Comparator` class (see Section II).[1]

Because these interactions are entirely provided by the framework, an analysis that solely considers the application's code cannot identify these implicit control flow transitions (ICFTs). That is, systems that analyze applications in isolation

[1] As we will discuss later in the paper, certain callbacks (e.g., `onClick`) can be implicitly registered through an XML configuration file. However, this is only possible for a very limited set of callbacks.

from the framework necessarily generate incomplete control flow graphs. As we will show in Section VII, this omission allows malicious applications to evade detection and moreover jeopardizes the correct analysis of benign applications.

The prolific use of these callback mechanisms in real-world Android applications mandates that analysis systems model the corresponding ICFTs accordingly. Existing systems use a variety of techniques in an attempt to address ICFTs. Callbacks related to the application's life-cycle are well documented and understood. Thus, current analysis systems model such callbacks by means of a manually-curated list of configuration entries. For example, the authors of FlowDroid describe how they were able to properly model the life-cycle of the main components after Android's documentation. As these are well-known and well-documented methods, we believe that this manual effort is sufficient to correctly handle such life-cycle related callbacks.

Where existing systems fall short, is in modeling callbacks that are not as well documented as those related to the application life-cycle. These callbacks are modeled either based on manually-curated lists [9, 18] or heuristics [23]. Unfortunately, in this case, neither approach is a sufficient treatment of ICFTs as the resulting control flow graphs lack edges that are not explicitly modeled or captured by the heuristics. In the next section, we provide two concrete examples of this concept, while in Section VII-F we demonstrate how this imprecision directly leads to undetected malicious functionality.

The main goal of our work is to remove the unnecessary imprecision that this second category of callbacks introduce in current analysis systems. We propose a novel static analysis approach and its implementation – EDGEMINER – as the first work to systematically address the challenge of ICFTs. EDGEMINER applies automated program analysis techniques to identify the complete set of callbacks and their registration methods in the Android framework. To this end, we implement a scalable inter-procedural backward data flow analysis that first identifies callbacks and then links these callbacks with their corresponding registration functions. In Section VII-F, we show how the results produced by EDGEMINER (i.e., pairs of registrations and callbacks) can be used to significantly improve state-of-the-art Android analysis systems, such as FlowDroid [9]. In particular, we show that before the integration of our results, FlowDroid fails to detect privacy leaks in applications that use ICFTs (i.e., false negatives).

We used EDGEMINER to analyze the codebase of three major Android versions. Note that EDGEMINER only needs to run once per framework version, as the extracted information pertaining to callbacks and registrations is invariant for all applications executing on a given framework version. This observation also implies that the results of EDGEMINER remain valid regardless of any obfuscation attempts made by individual Android applications.

In summary, this paper makes the following novel contributions:

- We identify the challenge of implicit control flow transfers provided by the Android framework as a source for imprecision in state-of-the-art static analysis systems.
- We design a novel automated analysis (§IV) and corresponding implementation (EDGEMINER §V) to analyze multiple versions of the Android framework for the complete set of callbacks and their registrations. Our results (§VII) indicate that modern Android versions feature nearly 20,000 registration and callback methods.
- To demonstrate that our results can easily be incorporated into existing analysis systems, we extend the popular FlowDroid framework with support for the newly detected ICFTs (§VII-F). This extension allows FlowDroid to detect information leaks that previously eluded the tool's capabilities.

II. MOTIVATING EXAMPLES

In this section we present two examples (that we will use throughout the paper) that illustrate the importance of implicit control flow transitions for the static analysis of Android applications. The main role of these examples is to show how a malicious application could leak private information through the usage of implicit control flow transitions. In particular, as we discuss at the end of this section, these simple examples are sufficient to evade detection from all existing static analysis tools, including FlowDroid [9], the current state-of-the-art taint analysis system.

Our first example (Figure 1) uses a *synchronous* ICFT. An ICFT is synchronous if the callback method is synchronously invoked as soon as its associated registration method is invoked. Our second example (Figure 2) uses an *asynchronous* ICFT. An ICFT is asynchronous if the invocation of the callback method is *delayed* to some time after its associated registration method is invoked. Both our examples consist of two parts: application space code (Figure 1a and Figure 2a), which is written by the application developer, and framework space code (Figure 1b and Figure 2b), which is defined within the Android framework itself.

Let us first focus on the application space code in Figure 1a. At lines 10-17 the `MalComparator` class is defined. This class implements the `Comparator` Java interface and its mandated `compare` method. During execution, this method stores the current GPS coordinates into the `MainClass.value` static field (Line 14). Then, the main method (of the `MainClass` class) is defined (Lines 3-8). This method first creates an instance (`mal`) of the `MalComp` object (Line 4) and then sets the `MainClass.value` static field to an integer constant (Line 5). Subsequently, in Line 6 it invokes the `Collections.sort` method and passes the `mal` instance as a second argument. Finally, `main` transmits the value stored in the `MainClass.value` field to the network (Line 7).

The implicit control flow transition in this example is provided by the framework code illustrated in Figure 1b. The `sort` method implemented in the `Collections` base class (Line 6) implements a variety of sorting algorithms (mergesort, quicksort, and insertion sort). Regardless of the chosen implementation, each algorithm repeatedly invokes the `compare` method (Line 10) of the provided `Comparator` object to assess the ordering of two elements in the collection. Thus, a call to the `sort` method implicitly invokes the `compare` callback method implemented in the application. Consequently, the value transmitted to the Internet at Line 7 of the `main` function corresponds to the sensitive GPS

```
1  class MainClass {
2    static int value = 0;
3    static void main(String[] args) {
4      MalComp mal = new MalComp();
5      MainClass.value = 42;
6      Collections.sort(list, mal);
7      sendToInternet(MainClass.value);
8    }
9  }
10 class MalComp implements Comparator {
11   int compare(
12         Object arg0,
13         Object arg1) {
14     MainClass.value = getGPSCoords();
15     return 0;
16   }
17 }
```

(a) Application Space

```
1  public interface Comparator<T> {
2      public int compare(T lhs, T rhs);
3  }
4
5  public class Collections {
6    public static <T> void sort(
7          List<T> list,
8          Comparator<? super T> comparator) {
9      ...
10     comparator.compare(element1, element2);
11     ...
12   }
13 }
```

(b) Framework Space

Fig. 1: An example that shows that without properly linking the `sort` method (invoked at (a) Line 6) to the `compare` method (defined at (a) Line 11) and invoked at (b) Line 10, existing static analyzers would not detect the privacy leak. Note that the framework space code is simplified for understanding.

```
1  class MainActivity extends Activity {
2    static int value = 0;
3    onCreate(Bundle bundle) {
4      MalListener mal = new MalListener();
5      MainActivity.value = 42;
6      // get a reference to a button GUI widget
7      Button b = [...]
8      b.setOnClickListener(mal);
9    }
10 }
11 class FinalActivity extends Activity {
12   // This activity is reached towards the
13   // end of the application's execution.
14   onCreate(Bundle bundle) {
15     sendToInternet(MainActivity.value);
16   }
17 }
18 class MalListener implements OnClickListener {
19   int onClick(View v) {
20     MainActivity.value = getGPSCoords();
21     return 0;
22   }
23 }
```

(a) Application Space

```
1  public class ViewRootImpl extends Handler {
2    public void handleMessage (Message msg) {
3        switch (msg.what) {
4          case EVENT:
5            mView.mOnClickListener.onClick();
6            ...
7        }
8    }
9  }
10 public class View {
11   OnClickListener mOnClickListener;
12   public void setOnClickListener (EventListener
         li) {
13       mEventListener = li;
14   }
15 interface OnClickListener {
16   void onClick(View v) {
17   }
18 }
```

(b) Framework Space

Fig. 2: An example that shows that without properly linking the `setOnClickListener` method (invoked at (a) Line 8) to the `onClick` method (defined at (a) Line 19) and invoked at (b) Line 5, existing static analyzers would not detect the privacy leak. Note that the framework space code is simplified for understanding.

information as opposed to the constant value. Note that the ICFT in this example is synchronous: in fact, the callback method (`compare`) is invoked as soon as its associated registration method (`sort`) is invoked.

Similarly, the example provided in Figure 2 shows how a malicious application could leak sensitive information through an asynchronous ICFT. In particular, this application registers an `OnClickListener` (by invoking the `setOnClickListener` registration method, Line 8), and it associates it to a specific GUI Button. Once the user clicks on this button, the associated `onClick` method will be invoked, and the current GPS coordinates are stored in the `MainActivity.value` static field. Then, when the `FinalActivity` activity is reached, the recorded GPS coordinates will be leaked.

A static analysis that analyzes applications in isolation from the framework will miss these implicit control flows. This implies that systems that analyze applications for privacy leaks will incorrectly label the examples in Figure 1 and Figure 2 as benign (i.e., false negatives).

To demonstrate that, indeed, these examples constitute false negatives, we augmented the DroidBench [8] benchmark suite with a variety of test cases similar to the presented example. In Section VII-F, we show how the technique that we described in these two examples can be used by a malicious application to evade analysis. We also modified FlowDroid [9] to leverage our results to correctly model ICFTs, and we show how our improvement allows FlowDroid to identify previously-undetected privacy leaks.

Existing Approaches. ICFTs are prolific and static analy-

sis systems cannot afford to ignore them entirely. Although existing static analysis systems acknowledge [9, 18, 23] that callbacks must be handled, they do not address the challenge comprehensively. Instead, existing systems address this challenge with one of the following, incomplete techniques. The majority of the approaches (e.g., [9, 18]) rely on manually compiled lists for implicit control flow transfers. However, the large number of ICFTs (i.e., EDGEMINER identified more than five million) renders manual efforts to identify all ICFTs intractable. Other approaches attempt to solve the ICFT problem based on heuristics. CHEX [23], for example, connects all potential callbacks to the constructor of the containing object. The example in Figure 1 would cause CHEX's data flow analysis to incorrectly conclude that the GPS information is overwritten by the assignment of the constant value in Line 5.

Another approach would be to treat all the non-reached methods (such as the `compare` and the `onClick` methods in our examples) as top-level methods. However, this approach would cause false negatives as well. In fact, if the `compare` or the `onClick` methods are analyzed only after the analysis of the main application's codebase has been analyzed, the static analyzers will miss the information leak.

The only way to properly address this issue is to analyze such callbacks within the right *execution context*. In other words, a static analyzer would need to analyze the `compare` method just after the `sort` method, or analyze the `onClick` method when the user could click on the button (i.e., potentially before the final activity is reached). Clearly, a static analyzer can perform this kind of precise analysis only if it is *aware* of such registration-callbacks implicit control flow transfers. The main goal of our work is to enable existing static analyzers to perform more precise static analysis, and to detect privacy leaks even in the scenarios presented in our motivating examples.

A last, overly conservative, approach would be to add control flow edges to all callbacks at any function callsite in the application. To the best of our knowledge, no existing system adopts this naïve approach. While this heuristic would work for our examples, the CFG of real-world applications would explode in size and negatively impact the precision of any data flow analysis performed on top of such an inflated CFG.

III. OVERVIEW & PROBLEM STATEMENT

Heuristic and manual approaches to tackle the challenge of ICFTs either do not scale or are incomplete. Thus, we propose a novel approach to automatically extract all registration functions and their associated callbacks provided by the Android framework. In this section, we first present an overview of our system, we then provide a precise definition of *registration* and *callback* methods, and, finally, we specify our problem statement.

A. Overview

Figure 3 provides a schematic overview of our work. Our approach takes as input the entire codebase of the Android framework. The output is a list of ICFTs as pairs of *registration* and *callback* methods along with their corresponding type signatures. This list *summarizes* the implicit control flow behavior of the Android framework and can be used to improve the precision of existing Android analysis systems (lower part of Figure 3). Our approach is based on conservative program analysis techniques and produces only *one-sided* errors. That is, while we are willing to tolerate false positive pairs of registrations and callbacks, no false negative ICFTs can occur from our analysis.

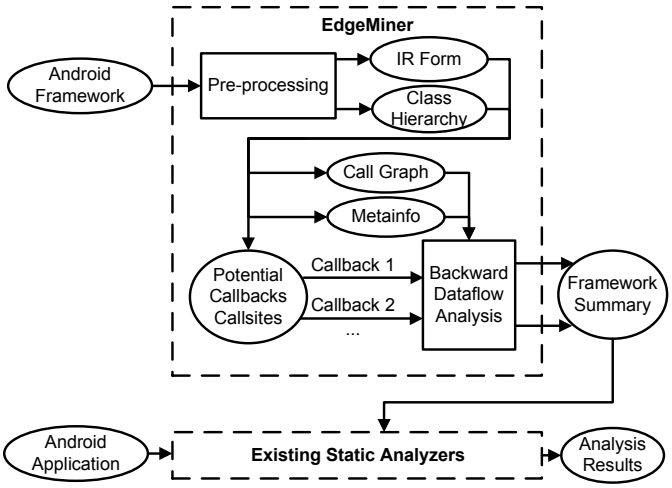

Fig. 3: Overview of EDGEMINER.

B. Problem Statement

In this section we first provide definitions of the terminology we use throughout the paper. Then, we formulate the problem we are trying to solve using this terminology.

Definition 1 *An application callback is a method implemented in application space that can be (implicitly) invoked by the framework.*

From a technical point of view, the callback mechanism in Java and other object-oriented languages (e.g., Objective-C) relies on *method overriding* and the *dynamic dispatch* mechanism: First, an application space method overrides a method defined in framework space. Subsequently, when the framework invokes this method, the dynamic dispatch mechanism invokes the overridden (application-defined) method corresponding to the dynamic (runtime) type of the object. We refer to the framework method that is overridden as a *framework callback*.

Definition 2 *A framework callback is a method defined in the framework space that can be overridden by an application space method (i.e., an application callback), in a way that the overriding method can be (implicitly) invoked by the framework.*

The two callback definitions correspond to two different views (application and framework) on the same concept. The difference is that application callbacks are dependent on the concrete implementation of individual applications, whereas framework callbacks encompass all possible callbacks any application can use. As our goal is to identify all ICFTs in the Android framework, we use the framework callback definition

as the working definition for this paper. In our example, the definition of the `Comparator` interface's `compare` method is the framework callback, and the concrete implementation of of the `compare` method in the `MalComp` class (Lines 11-16) corresponds to the application callback.

Before an application callback can be invoked, the framework must be made aware of the callback's existence. This is accomplished through a so-called *registration* method.

Definition 3 *A registration method is a method implemented in framework space that communicates the availability of an application callback to the framework itself.*

Intuitively, registration methods are the methods through which an application space object implementing an application callback flows to the framework space. Clearly, these registration methods are necessary for the realization of the callback mechanism. In our example, the `sort` method in Figure 1a (Line 6) is a registration method, because it passes the `MalComparator` instance with its `compare` application callback to the framework. Another popular pair is the `setOnClickListener` registration method (Figure 2a, Line 8) and an object implementing the `OnClickListener` interface with its `onClick` callback (Figure 2a, Lines 18–23).

Based on the above definitions we want to automatically determine all pairs of registration and callback methods that result in implicit control flow transfers. For an ICFT to occur, the object on which a callback is invoked must be the same object that is communicated to the framework in a previous registration call. Thus, a final requirement is that registrations and callbacks are connected through a data flow of the corresponding object. Clearly such a data flow is present in the examples in Figure 1 and Figure 2 facilitated by the respective `mal` objects. However, passing an object that implements a callback to the framework through a method other than a registration, does not result in the required data flow. For example, adding an object that implements the `OnClickListener` interface to a container (e.g., a Vector, `java.util.Vector.addElement(mal)`) does not result in an ICFT. The reason is that the Android framework does not contain code that invokes the `onClick` method of objects stored in a `Vector`.

IV. Approach

As explained in the previous section, our approach takes as input the Android framework codebase, and it extracts all implicit control flow transitions throughout its codebase. The output of the system is a list of registration-callback pairs.

To identify these pairs of method calls, we first extract a list of potential callbacks and identify all callsites in the framework that can invoke these potential callbacks. Subsequently, for each identified callsite, we perform an inter-procedural backward data flow analysis to trace the origin of the object used at the callsite. If this analysis reveals that the object is passed to the framework as a parameter from the application space, we successfully identified a registration-callback pair.

In the remainder of this section, we explain the details of our approach. In particular, as Figure 3 shows, the analysis needs to perform several preprocessing steps, before the actual backward data flow analysis identifies the resulting registration-callback pairs.

A. Preprocessing

Prior to performing the actual data flow analysis, our approach performs a series of preprocessing steps. In a first step, we transform each method of the framework into an intermediate representation that is particularly well suited for the data flow analysis at the core of our system. We also extract the class hierarchy and interface definitions from the Android framework. Subsequently, we extract an over-approximation of the framework's call graph. At the same time we label all potential callbacks in the generated call graph. The final preprocessing step collects meta-information pertaining to field accesses (i.e., read and write) in the framework.

Intermediate Representation (IR). The first preprocessing step consists of transforming the individual methods implemented in the Android framework into Static Single Assignment [12] (SSA) form. Code in SSA form has the property that each variable is assigned exactly once. Different assignments to the same variable are represented as distinct *versions*. Representing the methods of the framework in SSA significantly eases the implementation of the backward data flow analysis, which we describe in detail in the next section.

Class Hierarchy. The class hierarchy is a tree data structure that represents the inheritance relationships between individual classes defined in the framework. Similar to the Java class hierarchy, the Android framework class hierarchy is rooted at the `Object` node. That is all classes defined and used by the Android framework inherit from `Object`. The class hierarchy also stores information about interfaces and attributes classes to the interfaces they implement.

As this information is needed for generating an over-approximation of the call graph, our approach reconstructs the class hierarchy of all classes and interfaces defined in the Android framework. This is done by retrieving, for each class and interface, which class they directly extend and which interfaces they directly implement, if any. Then, we compute the transitive closure to have the complete information readily available.

Call Graph Construction. The goal of this step is to construct the framework's call graph, which is necessary to implement the data flow analysis inter-procedurally. We use the following conservative approach to extract an over-approximation of the framework's call graph. For each *invoke* instruction (i.e., method call), the analysis determines all possible targets. This is achieved by combining the information of the static type of the receiving object and information available in the class hierarchy. As common for object-oriented languages, the target of a method call depends on the dynamic type of the receiving object. At runtime, the dynamic dispatch mechanism is responsible to identify and call the correct implementation of a polymorphic method. Unfortunately, it is an undecidable problem, in general, to identify the precise dynamic type at a callsite statically. To address this problem, our analysis conservatively identifies all potential methods (i.e., callees) that can be invoked at a callsite as follows. We first retrieve the static type T of the receiving object, which is available from the framework's code directly. Then, since polymorphism as

implemented by Dalvik mandates that the dynamic type S of an object is equal to T or any of its subtype (i.e., $S \leq: T$), we can process the subtree of the class hierarchy that is rooted at T to identify all possible method implementations. For each method defined in a non-abstract class in that subtree, we then check whether it matches the name and type signature of the method at the callsite. All methods that match these criteria are connected to the callsite with an edge in the call graph. Furthermore, if the static type of the receiving object at the callsite (i.e., T) is an interface, we repeat the previous steps for all classes in the framework that implement this interface.

This technique will result in an over-approximation of the call graph for regular Dalvik code. Note that instead of using this approach, a more sophisticated algorithm could be used to generate a more precise call graph. However, the size of the Android framework codebase (i.e., more than 8M LOC) places stringent requirements on scalability and performance of the employed algorithms. Thus, expensive fine-grained alias analyses can quickly become prohibitively costly. Another challenging point is that the Android framework codebase mixes Dalvik code with components compiled to native (binary executable) code. Also, the above-mentioned technique does not take reflective method calls into account when constructing the call graph. Imprecision due to native code or reflective calls threatens the completeness of our approach. Thus, we analyzed the prevalence of these techniques in the Android framework. In particular, as we will discuss in Section V, we found that the framework only contains a small number of reflective or native method invocations (36 and 46 respectively). As we can reliably enumerate these cases, we modeled the edges through manual annotations in the resulting call graph.

The remaining mechanism that could render our over-approximation incorrect is exceptions. It is well understood that exceptions are notoriously hard to model correctly. While we would expect malicious applications to use exceptions to evade detection, we do not expect the Android framework to use exceptions to implement the callback mechanism at the focus of our analysis.

Potential Callback Callsites. Definition 2 defines framework callbacks. However, to automatically identify callbacks, the textual description must be translated into a description that can be checked through automated analysis. Although we cannot easily translate the precise definition, we can translate Definition 2 into a sequence of automatically checkable necessary (but not sufficient) properties. Thus, evaluating these properties on all callsites in the framework will result in a superset of all framework callbacks (i.e., the set of *potential callbacks*).

Definition 4 *A potential callback is a framework method that can be overridden by an application space method. To this end, a method must satisfy the following necessary conditions:*

- *The method is public or protected.*
- *The class in which the method is declared has a public or protected modifier.*
- *The method is not final or static.*
- *The class in which the method is declared does not have the final modifier.*
- *The class in which the method is declared is an interface or has at least one explicitly or implicitly declared, public or protected constructor.*

The first four criteria are necessary so that the specified method can be overridden. The fifth criterion states that a class implementing the callback in application space can be instantiated.

We perform the evaluation of the above five properties during the generation of the call graph to avoid duplication of effort (e.g., identifying `invoke` instructions, consulting the class hierarchy). The resulting list of callsites constitutes all possible locations in the framework where a callback can be invoked. Thus, these are precisely the locations where we start our backwards data flow analysis to confirm whether the object at a given callsite is passed to the framework as an argument to a registration method.

Metainformation Collection. To aid the subsequent data flow analysis, this step extracts additional meta-information about the framework. In particular, we extract accesses (read and write) to fields and their corresponding classes. To this end, we map each field to all its *definition* sites in the framework (i.e., `put_field` instructions). Similarly, we map all fields to `get_field` instructions which retrieve the value of a given field. Because fields are defined in classes, we address the challenge of dynamic subtype polymorphism analogously to the above discussion in the call graph paragraph. In addition, we leverage the insights that methods in an inner class can access private fields in the outer class, and methods of a class can access package private fields defined by other classes in the same package.

B. Backward Data Flow Analysis

Based on the information extracted by the preprocessing above, we can now describe how we use our inter-procedural backward data flow analysis to identify registration-callback pairs. We start the analysis at each callsite to a potential callback. Intuitively, the analysis determines whether the object used at the callsite could originate from application space (i.e., whether the object is passed to the framework as a method argument). To this end, the analysis leverages the fact that use-def chains are explicitly captured in the SSA representation of the framework. Thus, starting at a callsite to a potential callback, the analysis recursively traverses the use-def chains backwards until either of the stopping conditions (discussed in the next paragraph) are met. While traversing the use-def chains within a method (i.e., intra-procedurally), four instruction types influence how the analysis proceeds.

- **Method Parameter Passing.** If the analysis reaches the start of the current method (*M*) and the use-def chain references a parameter (*P*) to *M*, the analysis consults the call graph and recursively continues at all callsites that can invoke *M*. When continuing at each callsite, the analysis will track the argument that maps to parameter *P* in method *M*.
- **Method Call.** If the analysis encounters an `invoke` instruction (i.e., a method call) while traversing the use-def chain, it consults the call graph and recursively continues at the `return` instruction for all possible callees.

- **Field Access.** If the analysis encounters a `get_field` instruction, the meta-information is consulted. The analysis continues recursively from all `put_field` instructions that define the specified field (Section IV-A details how we address the challenge of polymorphic subtyping for field accesses).
  Note how it would be possible, at least in principle, to use a more precise analysis to handle data flows through fields. However, scalability considerations dictate a tradeoff between precision and analysis target size (the Android framework codebase has over eight million LOC). Furthermore, our results indicate that this conservative choice does not result in prohibitively many false positives. More precisely, 90% of all fields accessed through `get_field` have at most three subtype-compatible `put_field` statements in the entire framework.
- **Static Definitions.** Two classes of instructions stop the processing along the current branch of the recursion. The first one is the `new_instance`, which creates a new object of the given type. Similarly, the `move_const` family of instructions stores a constant value in the target operand. These instructions unconditionally overwrite any previous values stored in the target operand with values solely determined by the framework. Thus, a use-def chain that includes such instructions can never link a registration to a callback method.

Outputting Criteria. Our analysis outputs a registration-callback pair if a flow is found from the receiving object at a callback's callsite to a parameter *P* of a potential registration. A potential registration is a method implemented by the framework that satisfies the following two conditions: 1) The method can be invoked from application space; 2) The class of parameter *P* defines a method that corresponds to a framework callback (i.e., the method is signature and type compatible to a framework callback and can be overridden).

Stopping Criteria. The recursive backward analysis stops when one of the two following conditions is met: 1) The analysis reaches the entry node of a method with no callers (i.e., the call graph has no incoming edge); 2) As discussed before, the receiving object of the callback is defined, within the framework, by a `new-instance` or a `move-const` instruction.

Handling Data Flow Loops. For our backward analysis we unroll loops once. Note how this choice does not affect the results of our analysis, as multiple iterations of the same loop do not influence the presence of a data flow from a registration to a callback.

Type Checking. At each step of the backward tracing, our system checks that the occurring types are compatible. More precisely, if the type of the object at the callsite where the backward tracing started is not in a subtype relation with the operand considered at the current step, this incompatible operand does not need to be traced further.

C. Completeness

All analyses presented above, including method calls through the reflective and native API, are conservative. Thus, we expect that our results exclusively contain one-sided errors, where we anticipate false positives due to the conservative analysis. However, we do not expect any false negatives. This expectation is supported by a large-scale empirical evaluation presented in Section VII.

D. Results Utility

In this section we discuss how the results of our analysis can be used by existing static analyzers. Our results (i.e., the pairs of registration and callback methods) are directly applicable to improve the generation of Android application control flow graphs. Note, however, that our definition of registration method does not specify how this information should be included in an application's control flow graph. In particular, our definition does not specify if a callback associated with a given registration method is invoked immediately (i.e., synchronously), or in a delayed (i.e., asynchronous) manner.

For example, the aforementioned `Collections.sort` example synchronously invokes the `Comparator.compare` method. In this (and similar cases), a static analyzer can directly link the registration to its associated callback in the control flow graph. However, in the popular `setOnClickListener – onClick` registration-callback pair the `onClick` callback is only invoked once the user taps the corresponding user interface element (e.g., button). Existing Android application analysis systems (e.g., [9]) model asynchronous callbacks by randomly invoking the callbacks from their manually curated lists whenever the application is in *idle* state. While this approach is intuitive, we acknowledge that application analysis systems can leverage our results in a variety of different ways.

To allow consumers of our results to handle the identified callbacks as precisely as possible, our analysis differentiates between synchronous and asynchronous ICFTs. To this end, each registration-callback pair in our results is annotated with the category for that pair. In Section VII-F we demonstrate that our results can be readily used to improve the precision of the FlowDroid [9] state-of-the-art Android application analysis framework with minimal code changes and minimal performance impact.

V. Implementation

We implemented the approach presented in the previous section in a system called EdgeMiner. All analysis steps implemented in EdgeMiner operate on the ROP intermediate representation (IR). ROP is the IR that Google's `dx` compiler uses internally. `dx` is developed by Google as part of the Android Open Source Project (AOSP) [2], and it forms one of the core components of the Android SDK. In particular, `dx` is responsible for translating Java bytecode to Dalvik bytecode. Thus, every one of the thousands of Android developers who uses the Android SDK to compile her application uses `dx`. Similarly, system images for Android, whether they are provided by Google, phone manufacturers, or third party after market versions, are all compiled with `dx` too. As a consequence, we are confident that the `dx` compiler, and its ROP intermediate representation are thoroughly battle-tested and reliable. Moreover, as the `dx` tool performs optimization-related analyses, its ROP IR, is well-suited to perform data

flow analysis. For example, ROP is in SSA form, and most of the information required for our analysis (e.g., use-def registers for each instruction), is readily available in the provided data structures.

The backward data flow analysis is implemented as a recursive procedure. The analysis proceeds backward from each callsite of a potential callback, and traces the origin of the object that receives the method call (i.e., the object that actually provides a concrete implementation of the callback). Intuitively, this step of the analysis aims to establish whether an object can flow, through an invocation of a registration method, from application space to the framework space. In other words, it aims to understand whether an application can *register* code within the framework, which is later called as a consequence. The analysis works on top of the over-approximation of the call graph built during preprocessing, and it consults the meta-information related to the fields, as mentioned in the previous section. While recursively traversing the over approximation of the call graph backwards is straightforward, there are a number of aspects that we needed to take into consideration to guarantee the completeness of our results. We describe these aspects in the remainder of this section.

Call Graph Construction. As a first step, we focus on method invocations that are performed through the `invoke` instruction. Similar to Dalvik, ROP provides an `invoke` instruction with the same semantics (i.e., dispatch a method call). As discussed previously, an over-approximation of the call graph for regular (i.e., managed) code can be determined with the help of the class hierarchy. However, the Android framework codebase has several aspects that need to be taken into account: reflective calls and native code components. Although these mechanisms are rarely used in the Android framework, they threaten to render the over-approximation incorrect if not handled appropriately. We discuss these aspects in the following two paragraphs.

Reflective Calls. Identically to Java, reflection in Android is implemented by the reflection API. Reflective method calls are exclusively handled by the `java.lang.reflect.Method.invoke` API call. By analyzing the Android framework source code we can reliably detect all reflective method calls based on this method's signature. Note how all reflection-related calls that lead to a method invocation must use this interface. Our analysis of the Android framework codebase revealed 36 invocations of this method. In all cases, it was possible to determine the *target* method, as its class name and method name are hardcoded in the source code. By adding the corresponding edges after the automated call graph construction, but before the data flow analysis, we preserve the call graph's property of being an over-approximation.

Native Code Components. We now describe how we handled native code in the Android framework. The Android framework consists of a mix of components authored in Java (which are then compiled to Dalvik bytecode), and components authored in a lower-level languages (such as C and C++), which form the so-called native code components. In the context of our work, the challenge is that a native code method can invoke regular methods implemented in the Android framework. This, in turn, can affect the completeness of the over-approximation of the call graph. In fact, as the `dx` compiler, and thus our analysis, solely considers Java code, our analysis would miss native method calls which would render the generated call graph incomplete.

To properly handle this aspect, we investigate how and why native code invokes methods implemented in the managed Dalvik environment. We found that native code can invoke regular Android methods only through the `dvmCallMethod` (and similar) C++ functions. We manually analyzed the native code components of the Android framework, and identified 46 callsites where native code calls regular Android methods. We took into considerations all these instances and we augmented the generated call graph accordingly to preserve the over-approximation. Furthermore, we perform an investigation to categorize the functionality of these native-to-managed code calls. Our findings are summarized by the following list.

- The `<init>` and `finalize` methods of a class are implicitly invoked upon the execution of a `new` and `destroy` operation, respectively. In particular, after the native code allocates/releases memory for a new object, the native code will call the object's constructor/destructor (i.e., its `<init>` and `finalize` methods).
- A custom implementation of the `loadClass` method (if provided) is invoked during class loading.
- The `Thread.start` method invokes the `Thread.run` method of a thread.
- Network packet arrival is signalled to a private `dispatch` method.
- A customized implementation of the `printStateTrace`, `getMessage`, and `incaughtException` methods (if provided) can be invoked when an exception is raised.

Note how two characteristic features of reflective calls and native calls make this manual assessment possible. First, the native and reflective APIs are coherent enough that *all* occurrences can be easily enumerated. Second, the number of calls (i.e., 36 and 46 respectively) is small enough to warrant the one-time manual analysis effort. It is worth noting that, for our analysis, the important aspect is not the number of native code components in the Android framework. Instead we only need to identify all transition points from native code to managed code. These observations and findings give us strong confidence that the call graph on top of which the data flow analysis step operates is indeed an over-approximation.

VI. Discussion

In this section, we discuss some aspects of the Android framework that could threaten the completeness of our analysis. In particular, we discuss how we handle the fact that callbacks can be (implicitly) registered through XML configuration files, and we discuss what is the role played by the Android Looper and the Binder IPC mechanism, two aspects that are known to be complicated to be handled correctly.

Callbacks registered through XML Resources. EdgeMiner detects pairs of registration and callback method calls. So far, we discussed callbacks that are registered explicitly through registration methods. However, Android applications can also register callbacks through XML resource files. For example, a developer could associate an `OnClickListener` to a specific Button, by setting the `android:onClick`

attribute in the associated layout XML resource file. These callbacks are implicitly attributed to their corresponding object (e.g., a user interface button) when the framework renders the element described by the resource file. As this technique of specifying a callback does not follow the traditional registration-callback pattern, our data flow analysis would not find these *registration* methods. For this reason, we conducted a manual investigation on how this mechanism works, and determined that this feature of specifying a callback method through XML attributes is only available for a very limited set of well-documented callbacks. Intuitively, this makes sense. In fact, for each of these callbacks, the Android framework codebase must explicitly implement support for parsing the appropriate XML attribute and invoking, on behalf of the application, the associated registration method. For this reason, the developers of the Android framework are fully aware of each of these special callbacks, and are therefore able to fully and properly document the usage of each of these callbacks. Furthermore, we note that existing analysis approaches, such as FlowDroid [9] or CHEX [23] already handle callbacks defined in resource files. In summary, resource files must precisely define the involved callbacks, and, therefore, no additional analysis of the framework is necessary to support this functionality.

Android Looper. One of the well-known components that, in principle, makes the analysis of the Android framework challenging, is the Android Looper. This component is in charge of processing all the asynchronous events received by the framework. For example, when the user clicks on a button, the OnClick event is generated by the framework, and inserted in a queue. The Android Looper is implemented as an infinite loop (hence, the term *looper*) that waits for such events and processes them as soon as they arrive. The main Looper's method that implements this functionality is the `handleMessage` method, which acts as a generic dispatcher of events. In particular, this method dispatches events to their corresponding handlers according to the value of their `msg.what` field. For the events for which a callback is defined, the `handleMessage` method is in charge of retrieving the correct object (which is stored in a field), and invoking the method corresponding to the event. In these cases, the object has previously been *set* (i.e., stored in the field) by means of one of the registration method.

Clearly, the precise modeling of the Looper component is a challenging task [20]. However, we found that our current data flow analysis successfully determines registration-callback pairs, even for those cases where the Android Looper is involved (e.g., `onClick`). We now explain why this is the case.

The *event* message that is received by the Android Looper only contains information about the event's type (e.g., OnClick). The Looper will then parse this message, and, depending on the event, will retrieve a reference to the object that implements the callback, which is then invoked. As no code is specified in the event itself, the Looper retrieves the reference to the appropriate object from a field (e.g., the `mOnClickListener` field contains the listener for each object that inherits from View). To make use of a click listener, this field must be properly set by one of the registration methods. The key observation is the following: the data flow from the registration method to the callback invocation is through a field. Since our analysis already conservatively handles data flows through fields, a precise modeling of the Looper class is not required to have complete results.

Binder IPC Mechanism. Another well-known aspect of the Android framework that is known to be challenging to model is the Binder IPC mechanism. The Binder is used both for communicating within the same application, and to communicate between distinct apps.

To understand to which extent this component interferes with our analysis, we manually investigated how Binder is used by the Android framework. We determined that Binder can cause a set of callbacks to be implicitly invoked. Unsurprisingly, these callbacks are all related to intra- and inter-application communication, and are related to the life-cycle of application components. For example, when an Activity A invokes the `startActivity` method to start Activity B, a Binder IPC transaction is generated and, as a consequence, B's methods related to its life-cycle (such as `onCreate`, `onStart`, etc.) are implicitly invoked. Our data flow analysis does not detect this kind of implicit control flow transfers. This is because these flows do not follow the usual registration-callback pattern.

However, while ideally a more sophisticated analysis could be performed to detect this category of edges, we believe this is not necessary. In fact, we found that these implicit transfers are all related to the application's life-cycle. State-of-the-art static analysis tools, such as FlowDroid [9], already model the application life cycle based on the thorough Android documentation [4], which precisely defines the state machine that prescribes an application's life cycle and all involved callbacks. In summary, we believe that this aspect of the framework is already properly modeled by existing state-of-the-art systems, and that a manual effort is enough, as the number of these callbacks is limited and well-documented. In other words, even if we do not model the inherently dynamic parts of the Binder, we believe this aspect does not threaten the completeness of the registration-callback pairs our analysis discovers.

VII. Evaluation

In this section we discuss and analyze the results we obtained by evaluating our implementation of EdgeMiner. First, in Section VII-A, we describe the experimental setup. Then, in Section VII-B, we present an overview of the results we obtained by running EdgeMiner on three different versions of the Android framework. In Section VII-C we present several insights related to the registration-callback pairs we found. Next, we evaluate EdgeMiner's performance (Section VII-D) and accuracy (Section VII-E). In Section VII-F, we present a case study where we demonstrate how our results can be used to improve FlowDroid [9], a state-of-the-art static analyzer for Android applications. In particular, we first show how incomplete support of ICFTs leads to undetected malicious functionality (i.e., false negatives). Then, we show how FlowDroid can be easily extended to incorporate the results that EdgeMiner generates, to detect the malicious functionality that leverages ICFTs. Finally, in Section VII-G, we discuss how frequent ICFTs are used by real-world applications, and we show how the original version of FlowDroid

Android Version	# Registrations	# Callbacks	# Pairs
2.3 (API 10)	10,998	11,044	1,926,543
3.0 (API 11)	12,019	13,391	2,606,763
4.2 (API 17)	21,388	19,647	5,125,472

TABLE I: Number of registrations, callbacks, and pairs extracted by EDGEMINER in three Android framework version.

misses several privacy leaks. These leaks are detected as a consequence of our modifications to FlowDroid that leverage the results produced by EDGEMINER.

A. Experiment Setup

Before we can evaluate EDGEMINER, we have to precisely define the input for the analysis, and answer the question: What constitutes the Android framework? The Android Open Source Project (AOSP) contains implementations for 53,094 classes. However, not all of these classes are packaged up to form the Android framework. Instead, the BOOTCLASSPATH variable in the build environment specifies a set of 10 JAR archives that contain the 24,089 classes which ultimately comprise the Android framework.

Interestingly, we found that standard Android development tools (e.g., Eclipse + Android SDK) rely on a mock *Android.jar*, which does not export all the APIs defined in the above-mentioned JAR archives. In particular, we found that methods that are marked with the $@hide$ attribute in the Android source code are not included in the mock *Android.jar*. One might assume that because applications cannot invoke the *hidden* methods, this mock *Android.jar* is a suitable alternative definition for the Android framework. Unfortunately, this is incorrect, as applications can either use reflection or be compiled with non-standard tool-chains (e.g., with a complete *Android.jar*) to invoke the hidden methods.

For this evaluation, we thus adopt the correct definition of the Android framework to comprise *all* archives, classes, and methods that are specified by the BOOTCLASSPATH variable, and evaluate EDGEMINER on all 24,089 classes.

B. Analysis Results

We evaluated EDGEMINER on three different versions (2.3, 3.0, and 4.2) of the Android framework. Table I summarizes the results for each framework version. In particular, we report the number of registrations, callbacks, and registration-callback pairs that EDGEMINER identified for each version. In Android 4.2, for example, EDGEMINER found 21,388 registration and 19,647 callback methods, for a total of 5,125,472 registration-callback pairs. Note how the number of detected pairs increases in successive framework versions. This is not surprising, since additional functionality introduced in newer Android versions is frequently realized with additional packages, classes, and methods. These results indicate that a manual annotation of the ICFTs in the Android framework is likely an intractable endeavor. Due to space limitations, the remainder of this evaluation focuses on the results for the Android 4.2 framework.

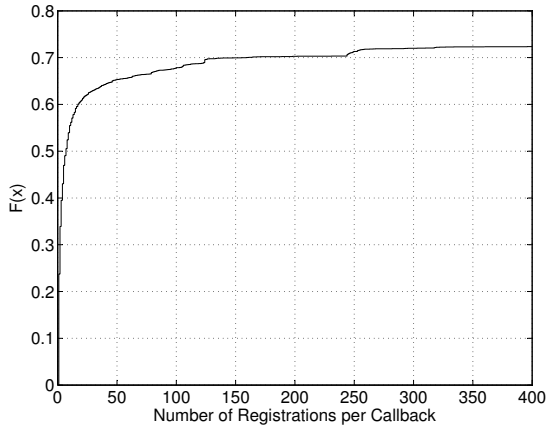

Fig. 4: Cumulative Distribution Function (CDF) of Number of Registrations per Callback (The CDF does not reach 1.0 because we trim the x-axis at a value of 400).

C. Callbacks and Registrations

In this section we discuss our insights for the registration-callback pairs that EDGEMINER extracted. Figure 4 shows the cumulative distribution function (CDF) of the number of registrations per callback. To better illustrate the curve for smaller values along the x-axis, we cut off the long tail. As the figure shows, most callbacks (>60%) have less than 20 registrations. A small number of registrations per callback is intuitive. A large number of registrations could be an indicator that the analysis is overly conservative. However, this is not necessarily the case. Consider, for example, the `onClick` callback mandated by the `OnClickListener` interface. One of the classes that provides a `setOnClickListener` registration for this type of callback is the `android.view.View` class. Virtually all user interface elements, such as buttons, text fields, or check boxes, extend or inherit from this base class. Android's online documentation for the `View` class lists 75 direct and indirect subclasses[2]. EDGEMINER will output one callback-registration pair for each of these subclasses of `View`. Similarly, the `toString` or `hashCode` callbacks, both of which are defined by `java.lang.Object`, are connected with a variety of registration methods. Because these callbacks are defined by the root node of the class hierarchy (i.e., `Object`) and overridden by dozens of subclasses they form a large number of pairs with different registration methods.

To further analyze the pairs of registrations and callbacks, we categorized the results according to their corresponding method names.

set- and on-. The first cluster consists of registration-callback pairs for which the registration method starts with *set-*, while its associated callback method starts with *on-*. This is an interesting cluster as the registration and callback methods' names correspond to the intuitive understanding of how callbacks are used in Android applications. We use regular expression to identify 77,982 pairs that match these patterns.

As an additional experiment, we identify another smaller cluster where the registration and callback methods satisfy the naming convention *setOn-Listener* and *on-*, respectively. For example, the classic *setOnClickListener* registration and

[2]http://developer.android.com/reference/android/view/View.html

onClick callback methods falls into this category. In total, our results contain 672 pairs that satisfy this naming convention. As expected, all *setOn-Listener* methods implemented in the framework are detected by EDGEMINER as registration methods. Note how existing approaches heuristically identify callbacks based on such naming conventions. This experiment shows that such heuristics lead to vastly incomplete results, as the number of pairs that conform to this naming convention is several orders of magnitude smaller than the number of pairs found by EDGEMINER.

D. Performance Evaluation

We evaluated EDGEMINER on a quad core 2.80GHz Intel(R) Xeon(R) X5560 CPU and 36GB of memory. To analyze Android 4.2, the system requires 8GB of memory to load the intermediate representation of the whole Android framework, and it then requires 4GB of additional memory to compute the framework call graph and perform the backward data flow analysis. In total this framework version contains 24,089 classes and 196,252 methods. Furthermore, the resulting call graph contains 161,229 vertices and 4,519,965 edges. The analysis of the entire Android framework completes in under 4 hours. Note that a specific version of the Android framework remains invariant for all applications running on top of it. Thus, EDGEMINER only needs to be run once per Android version, as the generated results, too, remain valid for that framework version.

E. Accuracy Evaluation

One key challenge of our work is to evaluate the correctness of our findings. Since, to the best of our knowledge, we are the first to systematically approach the challenge of implicit control flow transitions in Android, we lack authoritative ground truth to compare our results against. Moreover, the large number of registration-callback pairs found by EDGEMINER renders the manual analysis of all results impractical. Thus, we discuss in this section how we assessed false positives and false negatives among our results.

False Positives. To estimate false positives, we performed manual analysis of a random sample of 200 detected registration-callback pairs. For each of these 200 pairs returned by EDGEMINER, we manually verified whether it was possible to register a given callback through an invocation of its associated registration method. We found that 192 pairs out of 200 are indeed true positives. That is, for the 96% of such pairs, it was possible to *register* a given callback by invoking its associated registration method. In the remaining eight cases (i.e., 4%) manual analysis did not confirm the findings of EDGEMINER. We consider these eight samples false positives.

To increase confidence in our manual assessment, we chose 10 registration-callback pairs and embedded each in a distinct toy Android application. Executing the application in the Android emulator revealed that all callbacks are implicitly invoked by the framework as expected. We limited this analysis to 10 pairs, as the manual effort to satisfy all requirements to trigger the callback can be significant. The reason is that many classes that provide callbacks implement interfaces or extend abstract classes. This implies that the developer must provide implementations for additional methods that are not directly related to the callback functionality.

We also manually investigated the source of the false positives. We determined that all eight false positives are due to the fact that our analysis is based on an over-approximation of the call graph. This is expected, as the algorithms and data structures used by EDGEMINER are designed conservatively to strictly favor false positives over false negatives. However, a more precise call graph that still maintains the over-approximation property could help reduce false positives further.

Originally, we anticipated containers (e.g., sets, maps, etc.) to be a source for false positives. In fact, containers are well-known to pose significant challenges for static analysis. Interestingly, a detailed analysis of our results revealed that *no* containers are used in any of the data flows that connect registrations to callbacks. At first, this was surprising, as we expected the Android framework to coalesce a variety of callbacks in container data structures. However, an analysis of the framework revealed that callbacks are rather directly attributed to the specific element they operate on. For example, the `OnClickListener` object implementing an `onClick` callback is stored in a dedicated field for the `View` class.

False Negatives. In this section we describe how we evaluated EDGEMINER for false negatives. In particular, we try to answer the following question: Does EDGEMINER detect all registration-callback pairs? As we lack authoritative ground truth, answering this question conclusively is challenging.

Thus, for this evaluation, we designed an experiment based on dynamic analysis. Dynamic analysis has the advantage that the gained results are precise. That is, every ICFT identified dynamically, is guaranteed to be correct and witnessed by the application that produced it. However, the disadvantage of limited coverage implies that dynamic analysis results are also incomplete. Nonetheless, our confidence in EDGEMINER rises if we can show that a large-scale dynamic analysis experiment with real-world Android applications does not result in *any* ICFTs that EDGEMINER did not detect.

To dynamically extract ICFTs used by an application, we instrumented the Dalvik virtual machine to output a detailed execution trace of all the methods that are invoked during the execution of a given application. These traces contain information about the called methods, as well as the arguments and return values. Similar to our approach described in Section IV, we label potential registrations and potential callbacks. Note that in a dynamic analysis setting precisely labeling method calls that cross the application-framework boundary is trivially possible. We identify an ICFT in a dynamic trace iff: 1) A potential callback (i.e., method implemented in application space) is directly invoked by the framework, and 2) the object implementing the callback was passed to the framework in a previous registration call. We collected the execution traces for 8,195 randomly selected real-world Android applications by running each in an Android emulator for 120 seconds. During this time we simulated user interaction with the help of *The Monkey* [5]. Analyzing the resulting execution traces revealed 6,906 distinct registration-callback pairs.

We then compared this dynamically-generated list of pairs against the pairs extracted by EDGEMINER: All pairs that we found dynamically were already included in EDGEMINER's output. In other words, according to this experiment, our

approach is not affected by false negatives. Although we acknowledge that this is not a conclusive answer to the above stated question, this experiment instills significant confidence in EDGEMINER's results, especially because this experiment has been conducted completely independently from the backward data flow analysis. Moreover, this is not surprising as EDGEMINER is designed to produce one-sided errors (i.e., false positives) only.

An additional interesting aspect of this experiment is that the 8,195 applications we dynamically made use of *only* 6,906 distinct registration-callback pairs, out of the more than five million identified by EDGEMINER. This is due to the fact the vast majority of ICFTs identified by our approach are *not* documented, and, as a consequence, benign apps are likely not to use them. However, such non-documented ICFTs could be easily used by malicious applications for evasive purposes. For this reason, it is important that static analysis tools take all the ICFTs detected by EDGEMINER into account, even if only a small subset of them is actually used by real-world, benign applications.

F. Case Study - FlowDroid

EDGEMINER is motivated by the observation that existing static Android application analysis systems do not address the challenge of implicit control flow transitions systematically. Thus, the success of our work can be judged by the suitability of the results to improve existing analysis systems. To this end, we conducted the following case study.

FlowDroid [9] is an open source, state-of-the-art, static Android application analysis framework. The system performs static taint analysis of off-the-shelf Android applications to identify leaks of privacy sensitive information. The same authors also release DroidBench [8], a suite of benchmark applications to evaluate static Android analysis systems, especially those that perform data flow analysis.

In a first step of this case study, we extended the DroidBench benchmark suite with six additional sample applications. Three applications use asynchronous pairs to leak privacy sensitive information (e.g., GPS positional information) to the network, and three samples use synchronous registration-callback pairs (e.g., the pair of `Collection.sort` and `Comparator.compare`). Once we verified that the sample applications correctly transmit the sensitive information, we evaluated whether FlowDroid could detect the privacy leaks in any of the samples. Unsurprisingly, FlowDroid did not detect any privacy leaks.

As a second step of this case study, we show how existing static analysis tools would directly benefit from our results. In particular, this study has two goals: we show that FlowDroid fails to detect the privacy leak because of its incomplete modeling of the callback mechanism; we show how our results can be used from a practical point of view. We first investigated how FlowDroid supports the callback mechanism. In particular, FlowDroid ships with a configuration file (called `AndroidCallbacks.txt`) that holds a list of 181 callbacks. We verified that EDGEMINER automatically found all 181 callbacks. This further reinforces our belief that EDGEMINER is not affected by false negatives.

Pattern	# FlowDroid	# EDGEMINER
Listener	155	576
Callback	19	319
On	3	509
None of the above	4	18,243
Total	181	19,647

TABLE II: Patterns of callbacks used by FlowDroid and identified by EDGEMINER

Table II presents a breakdown of the 181 callbacks used in FlowDroid with respect to common naming patterns. The table also shows the number of callbacks identified by EDGEMINER for the same patterns. This indicates, once again, that manual or heuristic approaches to identify and handle callbacks are insufficient. In the remainder of this section, we describe how we integrated our findings with FlowDroid.

Integration of Asynchronous Callbacks. Augmenting FlowDroid with the information about asynchronous callbacks was straightforward, as it did not require any source code modification. We only needed to append the 19,647 callbacks identified by EDGEMINER to FlowDroid's configuration file. We then analyze the three applications that leak sensitive information through an asynchronous callback. With our modifications, FlowDroid successfully identified privacy leaks in all three applications. This clearly shows that the missing detection was indeed caused by the incomplete modeling of the callback mechanism.

Furthermore, we noticed two additional imprecisions in FlowDroid's modeling of the callback mechanism. First, FlowDroid invokes callbacks regardless of whether a previous registration occurred or not. In other words, FlowDroid will incorrectly analyze a callback (e.g., `onClick`), if it is implemented by an application but never registered with the framework. Second, FlowDroid exclusively relies on the method name to match callbacks to entries in the configuration file. These imprecisions can lead to false positives, as well as false negatives. We note that EDGEMINER outputs the pairs of registration and callback methods along with their type signatures, making a more precise handling of the callback mechanism possible. In summary, FlowDroid could prevent false negatives resulting from ICFTs, and reduce false positives by taking the additional information of the registration method into account.

Integration of Synchronous Callbacks. Once synchronous callbacks are identified with the help of EDGEMINER, integrating them with FlowDroid is straight forward. It is sufficient to augment the call graph of an application with a new edge for each registration and callback pair detected by EDGEMINER. To this end, we modified FlowDroid's source code in two ways. First, we add a routine at the beginning of the analysis that parses the registration-callback pairs extracted by EDGEMINER into a map data structure. This functionality was implemented by adding 36 lines of code to the FlowDroid codebase. The second modification augments the call graph generated by FlowDroid with the corresponding edges identified by EDGEMINER. More precisely, the call

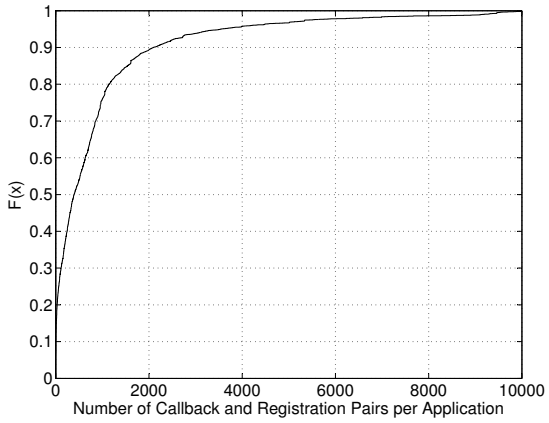

Fig. 5: CDF of the number of Callback and Registration pairs per application. 50% of the applications use less than 393 pairs. 90% of them use less than 2,000.

graph is traversed and for each callsite that type-checks (see Section IV-A) with a registration call, we add an additional edge to the corresponding callback method. This functionality is implemented with 28 lines of code.

We then used this modified version of FlowDroid to analyze our benchmark applications that evade detection by the unmodified FlowDroid through the use of synchronous ICFTs. In all cases, the modified version of FlowDroid successfully detected the privacy leak. This clearly shows how the results from EDGEMINER improve state-of-the-art static analysis frameworks by improving their precision and removing the broad class of ICFT-related false negatives.

G. ICFTs in Real-World Android Apps

We envision that the results of EDGEMINER will be used by existing static analysis systems. In the previous section we showed, as an example, how this can be easily achieved to improve FlowDroid. These systems would use our results and augment the call graphs of the applications they analyze. The large number of ICFTs identified by EDGEMINER begs the question whether using our results would create unusually dense call graphs for analyzed applications. Under the assumption that a denser call graph negatively affects scalability and precision of a static analysis system, we performed two different experiments to study to what extent our results would have that impact.

As the first experiment, we investigated to what extent real-world Android applications make use of the registration-callback pairs identified by EDGEMINER. In particular, we analyzed 18,672 randomly selected real-world Android applications to check how many registration-callback pairs they contain. To this end, we used apktool [1] to disassemble the applications, and retrieve the method names and type signatures for all `invoke` statements. Subsequently, we created the intersection of the identified methods with EDGEMINER's results and counted the matching entries in all the registration-callback pairs. Note that the intersection operation was not merely a simple set intersection. Because of subtype polymorphism (see Section IV-A) we also matched methods if any of their subtypes were contained in results obtained by EDGEMINER.

Tool	FlowDroid	FlowDroid + EDGEMINER
# Apps with at least one privacy leak	285	294
# Apps with no privacy leak	167	158
# Apps timeout	48	48
# Privacy leaks (in total)	46,586	51,418

TABLE III: Results obtained by analyzing 500 apps with both versions of FlowDroid.

Figure 5 shows the results of this experiment. The figure shows the CDF of the number of pairs used by the applications. We found that the 3% of the applications do not make use of any callbacks. A manual investigation revealed that all of these applications only provide a `WebView` for user interactions, and indeed do not make use of the callback mechanism. Furthermore, the majority of applications only use a small number of pairs. The figure shows how 50% of the applications use less than 393 pairs, and 90% of them use less than 2,000 pairs. Moreover, no application in our dataset uses more than 9,178 pairs.

The second experiment we conducted illustrates how the incomplete treatment of ICFTs in FlowDroid results in false negatives when analyzing benign off-the-shelf applications. To this end, we analyzed 500 randomly selected real-world (i.e., from Google's Play Store) Android applications with the current development version of FlowDroid[3]. We compared these results with the results of our modified version of FlowDroid that incorporates the additional information of ICFTs (as described in Section VII-F). This comparison is especially interesting along two different performance dimensions — analysis speed and precision. Thus, we answer the following two questions: 1) Does the large number of registration-callback pairs identified by EDGEMINER negatively impact the analysis' execution speed? and 2) Does this systematic approach to the ICFT challenge enhance the analysis' precision in terms of detecting privacy leaks?

Table III reports the results we obtained when conducting this experiment on the same hardware that we used to evaluate EDGEMINER on (see Section VII-D) with a five minute timeout. In particular, the table reports the number of applications for which at least one privacy leak was detected, for which no privacy leak was detected, and for which the analysis encountered a timeout. Moreover, the table also reports the total number of privacy leaks detected for the entire dataset.

First, the number of applications for which the analysis times out does not change. This indicates that the integration of our results with existing tools does not introduce scalability issues. Second, note how the improved version of FlowDroid detects privacy leaks in 9 applications that were previously determined to be free of privacy leaks. Of course, the applications identified to leak sensitive information by the modified version of FlowDroid are a strict superset of the original results. To assess whether these newly detected leaks are true positives, we performed an additional experiment: we executed these 9

[3]To ensure maximum comparability, we asked the developers of FlowDroid for access to their original evaluation dataset. Legal reasons prevented them from sharing their data.

applications within TaintDroid [13], a dynamic taint analysis system designed to detect privacy leaks. The rationale is that if TaintDroid can verify a privacy leak, we have a witness execution and know for a fact that the leak exists (i.e., it is a true positive). Out of the 9 applications, TaintDroid confirmed a privacy leak in four (all these apps were leaking IMEI), for three apps no privacy leak was detected, and TaintDroid crashed for the remaining two. This confirms that, at least for the four applications where TaintDroid found a leak, the original version of FlowDroid is affected by false negatives. Augmenting FlowDroid with the results from EDGEMINER successfully identified these privacy leaks. Also, note that the fact that TaintDroid did not identify any privacy leaks in three apps, does not imply that the statically-identified privacy leaks are false positives. Because of its dynamic nature, TaintDroid also suffers from limited coverage. With advanced dynamic testing based approaches achieving roughly 33% coverage [26], it is possible and even likely that the corresponding privacy leak was not triggered in this experiment. In summary, we showed how the incomplete handling for ICFTs is the root cause of false negatives when detecting privacy leaks in real-world applications. As another interesting data point, note how the absolute number of detected privacy leaks increased by roughly 10% (from 46,586 to 51,418) after we integrated EDGEMINER's results.

Finally, we measured the impact of our work on the performance of the FlowDroid. We found that, on average, our modification requires 34.7 seconds to load EDGEMINER's results in the map data structure on startup. As this step simply parses a configuration file (that is invariant from the application to be analyzed), the cost of this step can be amortized by loading the data only once when processing multiple applications. We also measured how the analysis time changed as a result of augmenting the call graph generated by FlowDroid. We found that the modified version of FlowDroid is 1.85% ($\pm$ 0.3%) slower than the original unmodified version. This clearly shows how the overhead introduced by augmenting FlowDroid with information related to ICFTs is negligible. Furthermore, this corroborates the intuition that, even if the number of pairs we found is really high, real-world applications only use a small number of them.

VIII. RELATED WORK

In this section we discuss scientific publications that are related to our work. Most relevant are the works that aim to summarize different aspects of the Android framework and other platforms, such as the Java framework. Furthermore, we discuss works that apply static analysis techniques to analyze Android applications.

Automatic Extraction of Library Summaries. Recent works in Android security proposed approaches to summarize certain aspects of the Android framework. Felt et al. [16] develop Stowaway, where the authors aim to extract a mapping between Android APIs and the permissions they require to be executed. In their work, the authors use an approach based on dynamic analysis to extract these mappings, and they then use the results to discover over privileged applications. Au et al. developed PScout [10] to achieve the same goal, by applying static analysis techniques (e.g., reachability analysis) to the Android framework codebase. More recently, Rasthofer et al. developed SuSi [25], a tool that analyzes the Android framework codebase to automatically identify sources and sinks in the Android framework. In particular, they use a combination of machine learning and static analysis techniques.

Other works that relate to ours are those that applied similar techniques to the codebase of other libraries, such as the JDK library, or the Scala standard libraries. For example, Yan et al. [27] use the SOOT [22] framework to analyze JDK library for alias analysis, while Probst [24] developed a tool to extract a summary of the control flow graphs of the JDK library. Similarly to our work, Zhang et al. [28] perform static analysis on the Java standard library to model callbacks. However, their work is limited to handling synchronous callbacks, while asynchronous callbacks are ignored. As shown in Figure 2 of Section II, proper modeling of these callbacks is required: otherwise, malware can evade the analysis. Furthermore, Zhang et al. only match type signatures of callbacks and disregard the data flow dependency between registration and callback. First, this simple approach can lead to a significant number of false positives. Second, as we already mentioned, this approach is not sufficient to properly extract asynchronous registration-callback pairs: in fact, such callbacks are often implemented by storing an object in a specific field (through a registration method), and by then retrieving it, later on, to invoke one of its methods (i.e., a callback). Clearly, proper data flow analysis is required to handle these cases.

Our work is complementary to all these works. In fact, while we share the same goal of summarizing the Android framework (or a different library) to allow existing static analysis tools to perform a better analysis, we focused on a different security-related problem – the complete modeling of implicit control flow transitions through the Android framework.

Static Analysis on Android Applications. Enck et al. [14] developed ded, a tool to re-target Android applications from Dalvik to Java bytecode. Then, they used commercial off-the-shelf control-flow and data flow analysis to find mis-use of phone or personal identifiers and establish the prevalence of advertising networks among Android applications. Several other works apply a combination of techniques based on static analysis, machine learning, and heuristics to detect malicious Android applications [7, 15, 19, 30]. Jeon et al. [21] propose RefineDroid, a static analysis to infer the fine-grained permissions suitable for existing Android applications, and then enforcing them without affecting the applications' original functionality. Others have applied static analysis techniques to discover vulnerabilities in Android applications. For example, Zhou et al. [29] developed ContentScope to automatically find *content leak* vulnerabilities, while Grace et al. [18] developed Woodpecker, to detect *capability leaks*. Dexter [3] is a static analysis tool that can decompile and dissemble Android applications.

Finally, FlowDroid [9] leverages on-demand algorithms to perform static taint analysis on Android applications. The goal of FlowDroid is to identify applications that incidentally or maliciously leak privacy sensitive information. FlowDroid's website states that FlowDroid "needs a complete modeling of Android's callbacks". While the configuration distributed with FlowDroid cannot claim to be comprehensive, the conservative analysis employed by EDGEMINER identifies all callback and registration pairs in the Android framework. The results of

EDGEMINER can directly be used by the above-mentioned analysis systems to create more precise Android application control flow graphs.

IX. CONCLUSION

In this paper, we designed and implemented a novel analysis tool, called EDGEMINER, that automatically generates API summaries that describe implicit control flow transitions through the Android framework. Our approach works by statically analyzing the Android framework codebase. In particular, our approach performs inter-procedural backward data flow analysis to extract a list of registration-callback pairs throughout the framework.

We evaluated EDGEMINER on several versions of the Android framework, and we automatically reconstructed several million implicit control flow transitions. Moreover, we show how these implicit transitions can be used by malicious applications to evade static analysis tools, including FlowDroid – a state-of-the-art static analysis tool for Android applications. Our evaluation also demonstrates, how existing analysis tools can easily take advantage of our results to increase their precision and eradicate ICFTs as the root cause for a whole class of false negatives.

REFERENCES

[1] Android-apktool: a tool for reverse engineering android apk files. https://code.google.com/p/android-apktool/.

[2] Android Open Source Project (AOSP). https://source.android.com/.

[3] Dexter is a static android application analysis tool. http://dexter.dexlabs.org/.

[4] Documentation related to the Activity component. http://developer.android.com/reference/android/app/Activity.html.

[5] UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html.

[6] Worldwide smartphone shipments top one billion units for the first time. https://www.idc.com/getdoc.jsp?containerId=prUS24645514.

[7] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS: the Annual Symposium on Network and Distributed System Security*, 2014.

[8] S. Arzt. Droidbench. http://sseblog.ec-spride.de/tools/droidbench/.

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2014.

[10] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: analyzing the android permission specification. In *CCS: the ACM conference on Computer and communications security*, 2012.

[11] P. P. Chan, L. C. Hui, and S. M. Yiu. Droidchecker: analyzing android applications for capability leak. In *WISEC: the ACM conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

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

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI: the USENIX conference on Operating systems design and implementation*, 2010.

[14] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX Security Symposium*, 2011.

[15] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *ACM Conference on Computer and Communication Security (CCS)*, 2009.

[16] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *CCS: the ACM conference on Computer and communications security*, 2011.

[17] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated security certification of Android applications. *Technical Report, University of Maryland*, 2009.

[18] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *NDSS: the Annual Symposium on Network and Distributed System Security*, 2012.

[19] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[20] C.-H. Hsiao, C. L. Pereira, J. Yu, G. a. Pokam, S. Narayanasamy, P. M. Chen, Z. Kong, and J. Flinn. Race Detection for Event-Driven Mobile Applications. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[21] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *SPSM: the ACM workshop on Security and privacy in smartphones and mobile devices*, 2012.

[22] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop*, 2011.

[23] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: statically vetting android apps for component hijacking vulnerabilities. In *CCS: ACM conference on Computer and communications security*, 2012.

[24] C. W. Probst. Modular control flow analysis for libraries. In *SAS: the International Symposium on Static Analysis*, 2002.

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

[26] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 209–220, New York, NY, USA, 2013. ACM.

[27] D. Yan, G. Xu, and A. Rountev. Rethinking soot for summary-based whole-program analysis. In *SOAP: the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, 2012.

[28] W. Zhang and B. G. Ryder. Constructing Accurate Application Call Graphs For Java To Model Library Callbacks. In *SCAM: the IEEE International Workshop on Source Code Analysis and Manipulation*, 2006.

[29] Y. Zhou and X. Jiang. Detecting passive content leaks and pollution in android applications. In *NDSS: the Annual Symposium on Network and Distributed System Security*, 2013.

[30] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS: the Annual Symposium on Network and Distributed System Security*, 2012.