Dynamically Generating Callback Summaries for Enhancing Static Analysis

Steven Arzt
Fraunhofer SIT | ATHENE – National Research Center for Applied Cybersecurity, Darmstadt, Germany

Marc Miltenberger
Fraunhofer SIT | ATHENE – National Research Center for Applied Cybersecurity, Darmstadt, Germany

Julius Näumann
TU Darmstadt | ATHENE – National Research Center for Applied Cybersecurity, Darmstadt, Germany

Abstract

Interprocedural static analyses require a complete and precise callgraph. Since third-party libraries are responsible for large portions of the code of an app, a substantial fraction of the effort in callgraph generation is therefore spent on the library code for each app. For analyses that are oblivious to the inner workings of a library and only require the user code to be processed, the library can be replaced with a summary that allows to reconstruct the callbacks from library code back to user code. To improve performance, we propose the automatic generation and use of precise pre-computed callgraph summaries for commonly used libraries. Reflective method calls within libraries and callback-driven APIs pose further challenges for generating precise callgraphs using static analysis. Pre-computed summaries can also help analyses avoid these challenges.

We present CGMiner, an approach for automatically generating callgraph models for library code. It dynamically observes sample apps that use one or more particular target libraries. As we show, CGMiner yields more than 94% of correct edges, whereas existing work only achieves around 33% correct edges. CGMiner avoids the high false positive rate of existing tools. We show that CGMiner integrated into FlowDroid uncovers 40 % more data flows than our baseline without callback summaries.

2012 ACM Subject Classification Software and its engineering → Dynamic analysis

Keywords and phrases dynamic analysis, callback detection, java, android

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.4

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.2.2`
Software (Source Code): `https://github.com/Fraunhofer-SIT/DynamicCallbackSummaries/` archived at `swh:1:dir:774fc1c198c94da21f9d9dc21f9a9721c9ac233c`

Funding This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Min- istry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

1 Introduction

Static analyses are commonly used for checking software for security vulnerabilities, quality defects, privacy leaks, and other properties. The callgraph is a core data structure of interprocedural static analysis. It encodes which statements in the code call which methods. When an analysis encounters a method call, it queries the callgraph for the set of callees in which to continue the analysis. If the callgraph misses edges, the respective callees are not considered and the analysis is incomplete. If the callgraph, on the other hand, contains spurious edges, irrelevant subtrees in the callgraph must be processed. This may not only impact performance and scalability, but may also lead to false positives.

Computing a complete and precise callgraph is non-trivial due to virtual dispatch and exceptional control flow. Approaches such as SPARK [16] already handle polymorphism well. SPARK builds upon the Escape Analysis in Soot [14] to handle exceptional control flow precisely. Nevertheless, several problems remain unsolved. Firstly, software commonly [17] uses third-party libraries, which can contribute significant amounts of code. Large code bases, in turn, pose scalability challenges for callgraph generation. Secondly, libraries can be complex and, e.g., use reflective method calls, contain asynchronous control flow, or manage callbacks in collections. A callgraph analysis must be able to handle all these language features correctly. Some libraries such as the popular OKHTTP3 library have separate callbacks for successful and failed requests. The failure callback is only executed when an exception is thrown within the library code. An approach without support for exceptional flows misses these callbacks. In practice, existing callgraph analyses apply approximations that lead to spurious or missing edges, or are too complex to scale to large programs.

We observe that, from the perspective of most client analyses, only the interface of a library is relevant. Callgraph edges inside the library only need to be computed as a means to obtain edges that cross the library's interface through callbacks. We therefore argue that these API edges can be summarized once as a one-time effort. Only the information which API calls trigger which callbacks must be retained. Intuitively, the summary is a list of such target callbacks. When a client analysis encounters a call to an API method for which a summary is available, it plugs in the summary instead of analyzing the library. More precisely, it adds a callgraph edge to each callback described in the summary. A similar reasoning has already been applied to the data flow behavior of libraries [2].

Manually assembling these summaries is inefficient, since many libraries use callbacks. Furthermore, each new version of each library would need to be studied to reflect the changes made to the library API in the model. Consequently, the generation of these callback models must be automated. Static approaches [8] share the shortcomings of static callgraph analyses and require coarse approximations, leading to false positives as we show.

Our approach CGMiner is based on dynamic analysis instead. CGMiner takes a set of programs that use libraries of interest. It statically instruments these programs such that each call to a library function reports dynamic callgraph data to an analyzer. CGMiner executes all instrumented programs and combines the resulting execution traces into a *callback summary* for the respective library. It abstracts away from all program-specific behavior and reduces the summaries to edges between methods in the public interface of the library, omitting calls within the library itself leading eventually to the execution of a callback. If a library, for example, takes a callback as the first argument on an API call and then invokes a method on the object passed as the first argument, there is an edge between this API method and its first parameter. This edge exists regardless of which program uses the API and what the concrete callback implementation is. CGMiner captures such abstract callback semantics on the API level. With its focus on dynamic analysis, CGMiner avoids many of the common challenges in static analysis such as precisely modeling reflective method calls and complex collection types.

In this work, we focus on Android apps. On Android, libraries are compiled into the apps that use them, rather than being shared between apps. Consequently, a wide variety of libraries is used in apps [26], and handling them efficiently is vital for each app analysis. Furthermore, almost 2 million apps are available in the official Play Store. Since for each library, CGMiner requires a sample set of apps that use the respective library, such a freely accessible data source is beneficial. In addition, Android is widely used for dealing with sensitive data and functions, making client analyses that depend on callgraphs for,

e.g., finding data leaks, highly relevant in practice. We show that CGMiner can identify efficiently complex callback edges. While other approaches lead to true positive rates of more than 33% with an additional 20% of edges being incomplete, CGMiner achieves more than 91% correct edges. Note that CGMiner focuses on control flow and is intended to be combined with analysis-specific summaries such as StubDroid [2] for data flow analysis. Even with a simple integration approach, CGMiner summaries lead to 28% more correct flows being discovered than without callback summaries.

The remainder of this paper is structured as follows. Section 2 contains some background information about Android. Section 3 shows a motivating example. Section 4 explains the CGMiner approach. We describe implementation details in Section 5, before explaining CGMiner's limitations in Section 6. Our empirical research questions and evaluation data is contained in Section 7. Finally, we present related work in Section 8 and conclude in Section 9.

2 Android Background

Android applications are written predominately in Java and Kotlin. The compiler translate this code into Dalvik byte code, which is similar to Java byte code used by the JVM. After compilation, the Dalvik byte code is written into one or multiple *classes.dex* files. The dex files as well as the necessary resource files of the app are packaged in an APK file, which is ultimately just a ZIP file. Android apps may use system classes from the `java(x)` and `android` packages. The implementation of these system classes are shared between different apps and reside on the device. On the host computer, the Android SDK installs a stubbed version of this Android system classes, which only contains the method signatures and class hierarchies of the actual implementation. This stub jar is used to link against during compilation, but does not contain the actual implementation code. In contrast to the system implementation, all other third party libraries and their transitive dependencies are compiled to Dalvik byte code as well and placed alongside the application code in the same dex files, so that each application is self-contained.

3 Running Example

Listing 1 shows a program that uses a simplified API for communicating with a remote server once the app's main activity is launched. The library is a slightly adapted version of the OKHttp library. For the sake of brevity, we omit the implementation of the library and instead explain it using the code of the example program.

The library's main class `HttpLibrary` is responsible for communicating with the server. Each request is represented by an instance of the `HttpTask` class. Each task is scheduled for execution using the `schedule` method. Once all requests are scheduled, the program invokes the `runAll` method to run them against the remote server. Once a task is complete, i.e., the server has responded with results or an error, the respective callback is invoked. The implementation of the error callback is omitted in Listing 1 for brevity.

For demonstration purposes, we assume the following simplified implementation of the library. The constructor of `HttpTask` stores the callback that it receives as a parameter into a field in `HttpTask`. The `schedule` method adds the `HttpTask` to a list. Still, for not freezing the UI thread, the library is multi-threaded and performs the http request in the background. The `runAll` method spawns a worker thread that regularly polls the scheduled

Listing 1 Motivating Example Code.

```
1  ICompleted onComplete = new ICompleted(){
2    @Override
3    public void onCallback(String results){
4      Log.i("Web", "Results: " + results);
5    }
6  };
7  IHttpFailed onFailed = ...;
8  HttpTask task = new HttpTask("/api/do", onComplete, onFailed);
9  HttpLibrary lib = new HttpLibrary("http://www.company.com");
10 lib.schedule(task);
11 lib.runAll();
```

tasks, takes the task at the top of the worklist, and processes it. The worker thread sends the requests to the server, collects the results, and then invokes the callback. The callbacks are invoked on a different thread than the original call to `schedule` or `runAll`.

Callgraph algorithms traditionally do not model the delayed behavior of the callback and instead insert an edge from the call site that causes the callback to be executed to the callback implementation, e.g., from `Thread.start` to the `run()` method of the thread. CGMiner adopts the same behavior. Its callback summaries model an edge from `runAll` to the `onCallback` method of both completion and error callback. In other words, our model assumes that `runAll` immediately invokes both callbacks. The challenge in this example, which is not handled by existing approaches, arises because the callbacks are not in the scope of the call site for `runAll`. Instead, the summary generator must automatically infer the link to the `HttpTask` instances that were scheduled, and then to the actual callbacks that were passed to the constructor of the `HttpTask`.

Existing approaches such as EdgeMiner [8] model the callgraph edges in Line 8. This reduces the complexity of the example, because the callback is not passed across multiple classes. On the other hand, such a model is incompatible with a flow-sensitive analysis. To illustrate this, we change the example slightly as shown in Listing 2. For this example, suppose that the `source` method call in Line 9 returns sensitive information. Further assume that the parameter of the `sink` method (called in Line 4) is sent to a remote server. This example uses the field `data` to save the sensitive information, which is leaked in the callback method.

Consider we are running a flow-sensitive taint analysis such as FlowDroid [5] to detect this data flow from `source` to `sink`. FlowDroid starts at the source [4] statement in Line 9 and advances forward through the control-flow graph until it reaches a sink. It reports a leak when the sink is reached with a tainted parameter. During the propagation, it keeps track of all variables that may contain sensitive information ("tainted").

Notice that the callback is passed to the library in Line 8, before the `source` method is called. During execution, the call to `runAll` in Line 12 invokes the callback which leaks the data. Nonetheless, EdgeMiner inserts an edge at Line 8 to the `onCallback` method. Because the call to `source` happens after the callback registration, FlowDroid does not encounter the sink statement when using the EdgeMiner summary. Consequently, the leak is missed. In this example, FlowDroid needs an edge from the `runAll` call in Line 12 to the callback method in order to reach the sink and thus detect the dataflow.

Listing 2 Flow-Sensitivity Example.

```
1  ICompleted onComplete = new ICompleted(){
2    @Override
3    public void onCallback(String results){
4      sink(data);
5    }
6  };
7  IHttpFailed = ...;
8  HttpTask task = new HttpTask("/api/do", onComplete, onFailed);
9  data = source();
10 HttpLibrary lib = new HttpLibrary("http://www.company.com");
11 lib.schedule(task);
12 lib.runAll();
```

Such approximations as in existing work have even greater negative consequences. In yet another modification of the example, imagine that the `runAll` method never calls `onFailed`, but throws an exception instead. The `onFailed` callback only exists for a second method `tryRun` that calls `onFailed` in the case of an error and that never throws an exception. In that case, EdgeMiner would still model the edge from the `HttpTask` constructor call in Line 8 to `onFailed`. This edge is clearly invalid if the program never calls `tryRun`. The EdgeMiner model does not contain any notion of `runAll` and `tryRun` and, hence, cannot make this distinction.

4 Approach

To avoid the inherent challenges of static analysis described in Section 1, CGMiner relies on dynamic analysis for inferring the callgraph summaries on libraries. Figure 1 shows the architecture of the analysis. CGMiner takes as input the original APK file and a list of classes that correspond to libraries. These apps are then instrumented with three analyses: a general-purpose dynamic callgraph analysis, a general-purpose dynamic taint analysis and a specialized callback analysis into the app. Since libraries (except for the Android Framework) are part of the application code in Android, library code can be instrumented as well.

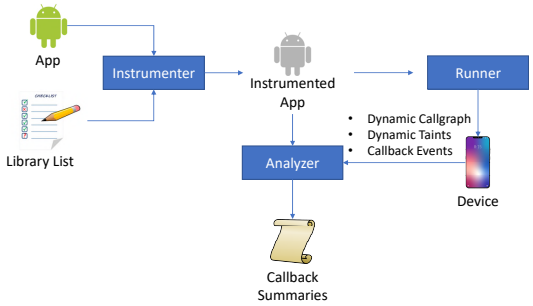

Figure 1 CGMiner Approach. The app is instrumented and then executed on a device, with runtime events being routed to the callgraph analysis. Events include dynamic callgraph, dynamic taint tracking, and specific callback analysis events.

The instrumented app is passed to the runner, which installs it on a device, and establishes a communication channel with the app. It forwards all events sent by the analysis code injected into the app to the analyzer. The analyzer is responsible for processing the events

and for inferring the callback summaries while the app is running. In the remainder of this section, we explain how the analyzer derives the callback summaries and how the different components (dynamic callgraph, dynamic taint analysis, callback analyzer) interact.

4.1 General Idea

CGMiner needs to identify which statements trigger which callbacks. The app is instrumented with event tracing that reports back to the analysis computer whenever a potential callback method is invoked on the phone. When such a method is invoked, CGMiner uses a combination of dynamic control flow analysis and dynamic taint tracking to identify all API calls between the point where the callback class was originally passed to the library (constructor of `HttpTask` in the example) and the callback method.

When arriving inside a callback, CGMiner must find the API call that triggered the callback (method `runAll` in the example). Intuitively, this can be done by searching backwards through the dynamic control flow graph. This approach, however, does not identify the necessary state changes to the `HttpLibrary` object. Recall from Section 3 that the `runAll` method would not invoke any callback unless the callback has previously been registered with the library using the `schedule` method. In other words, the inner state of the `HttpTask` object must be changed before calling `runAll`. More generally, the callback is passed through several objects along the way, and CGMiner must identify the API calls that lead to these state changes, i.e., that copy the callback around. In the example, the constructor of `HttpTask` assigns the callback to a field and the `schedule` method adds the task to the list, which is finally processed by `runAll` (see Section 3). We call such methods *transfer methods*. The user code must call this method for the callback to be registered, and it must therefore be part of the callback summary.

Intuitively, when an API method stores a callback in a field or collection inside of some object, the transfer method is the last API call that transitively lead to the assignment. To find the relevant assignments, CGMiner relies on dynamic data flow analysis. CGMiner taints each callback object when it is first passed to an API call (constructor of `HttpTask` in the example), i.e., each callback object is considered a source for the dynamic taint tracking algorithm. It then follows this taint through the library, until it arrives at the `this` object inside a callback method. The start of a potential callback method is considered a sink.

The taint state on the device is always mirrored to the analysis computer. When the analyzer observes that a callback has been called, it retrieves the taint paths, i.e., all statements that have passed the taint from one object to another on the path between the callback registration and the invocation of the callback method. Whenever a new object is tainted, e.g., through an assignment to a field, CGMiner searches the dynamic callgraph backwards to find the API call that triggered this taint transfer, i.e., the methods that user code must call before `runAll`.

4.2 Overview of the Approach

We define a *border edge* as an edge that is from library code to user code or vice versa. In other words, the caller is in library code and the callee is user code (i.e., a callback), or the caller is in user code and the callee is library code (traditional call to a library method). Edges in the first case, i.e., callbacks, are denoted *out* edges, whereas edges in the second case, i.e., normal library calls, are *in* edges. Figure 2 shows an abbreviated control flow graph for Listing 1. It further shows the *in* and *out* edges for the motivating sample. In the example, the `HttpTask` and `HttpLibary` constructor calls and the calls to `schedule` and

`runAll` are *in* edges, because these calls in the application code directly call library methods. The `runAll` method iterates over all scheduled tasks, which were registered in the `schedule` method, and calls a library-internal `run` method. The `run` method calls the `onComplete` on the callback. Thus, this method call constitutes an *out* edge.

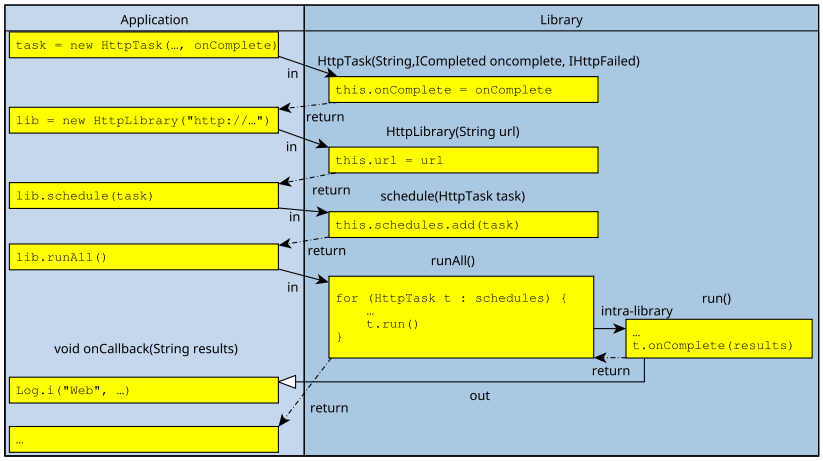

Figure 2 Shows *in* and *out* edges for Listing 1. *in* edges are regular arrows. The *out* edge is denoted by a white tip.

As a pre-analysis, CGMiner statically over-approximates the potential callback implementations, i.e., potential targets of *out* edges. We call these methods *callback candidates*. Afterwards, CGMiner statically approximates a set of all possible callback classes. In Listing 1, the constructor `HttpTask` may register a callback. All classes that (transitively) implement the interfaces `ICompleted` and `IHttpFailed` are potential callbacks. Specifically, we have the anonymous inner class implementing `ICompleted` and the (omitted) corresponding inner class for `IHttpFailed`. These classes override the interface methods, in this case only `onCallback`. These overriden methods represent the potential targets of *out* edges. These steps for identifying callback registration sites and potential callbacks are static over-approximations, which are used to bootstrap the dynamic analysis that follows later.

The dynamic analysis is used to determine which callback candidates are reached and to perform dynamic taint tracking in order to track which API calls (such as the *schedule* and the constructor calls in Listing 1) are necessary.

At the start of each callback candidate, we statically instrument a call to a method we call *reporting method*. This reporting method sends an event with information about the triggered callback to the analyzer. Because irrelevant callback candidates or spurious callback implementations will later not be reached during dynamic analysis, they will not become false positives. We provide details on how potential callbacks are identified in Section 4.3.

When a callback is invoked at runtime and the respective callback event is triggered, the analyzer uses the dynamic callgraph to find the corresponding *in* edge. The call site at the *in* edge represents the API method that triggers the callback. Note that the analyzer skips the library-internal calls between *in* and *out* edge. The *in* and *out* edges are trivial to identify based on the classes in which the respective calls and their corresponding callees are located. In the example from Listing 1, the analyzer deduces that a call to `HttpLibrary.runAll` invokes `IHttpCompleted.onCallback`. Further, CGMiner uses dynamic taint tracking to find the transfer methods. Therefore, the statements at the beginning of all potential callback

methods are marked as sinks for the dynamic taint analysis. To build the list of sources, CGMiner first collects all API sites that receive instances of potential callback classes as arguments. These API calls are then registered as sources in the dynamic taint analysis such that the respective potential callback classes, i.e., the call arguments, are tracked at runtime. Note that this approach is an over-approximation. When generating the callback summaries, CGMiner only relies on the taint paths that were actually taken at runtime. Therefore, marking too many classes as potential callbacks or registering too many APIs as sources does not reduce CGMiner's precision. CGMiner assigns an unique ID number to each taint source in order to distinguish different sources.

Transfer methods do not need to operate on the original heap object. The `schedule` method in Listing 1 never touches the callback object. It operates on the `HttpTask` that encapsulates the callback. An approach based on object identity alone would miss the `schedule` method. Dynamic taint tracking, on the other hand, can taint the `HttpTask` object when it encounters the assignment inside the constructor of `HttpTask`. Recall that this constructor receives the callback and stores it in a field. Afterwards, the dynamic taint tracker follows the `HttpTask` object as well. Similar reasoning must be applied for the `schedule` method, which stores the `HttpTask` object in a list, i.e., the list must be tainted as well.

With this taint tracking information, CGMiner can identify all border edges by looking at the taint transfers. Recall that the first statement inside the callback is a sink. CGMiner can query the taint analysis for the corresponding source and all statements in between at which taint was transferred to fields or collections (details in Section 4.5). For each statement on the taint path, it queries the dynamic callgraph to identify the corresponding *in* edge, i.e., the call from the user code that lead to the taint transfer. In the example, this allows CGMiner to identify the call to the constructor of `HttpTask` (transfer: field assignment) and to `schedule` (transfer: collection). Approaches that only inspect the call chain that ends at the callback (`runAll` in the example) would miss these intermediate calls.

4.3 Identifying Potential Callbacks

Recall that the callback analysis is started when a callback is invoked. Consequently, each possible callback method must be instrumented with an event that notifies the analyzer about which method has been called at runtime. This section shows the static analysis phase of the CGMiner approach. In this phase, CGMiner first determines possible callbacks and then instruments code in order to be notified when a callback happens.

To find the potential callbacks, CGMiner identifies all statements in the app that call library methods. The approach then checks whether a reference to a callback object is passed as an argument. We define a callback argument as follows. The declared type of the respective parameter is a reference type from the library, i.e., a class or an interface, which must be non-final and accessible to user code according to its access modifiers. Further, if the library class is a class and not an interface, it must have a non-private constructor. The type of the argument that is passed must be a class type from the user code that (potentially indirectly) inherits from the library class or implements the library interface. This class type represents a potential callback implementation. The type of the callback class can be approximated statically, either by identifying the allocation site at which the callback object was instantiated, or by looking at the declared type of the callback variable that is passed as an argument. To be complete, CGMiner applies a Variable Type Analysis (VTA) style analysis [24] to identify all possible types based on the declared type if no precise allocation site is available. Considering too many potential callbacks only leads to more instrumentation effort and does not affect the precision of CGMiner, as these spurious callbacks are not triggered at runtime.

CGMiner only identifies a set of potential library classes in this step. The concrete library method to which the callback object is passed is irrelevant for the analysis in this stage and is discarded.

Potential callbacks must be instrumented. Since this is not possible for Android and Java system classes, CGMiner automatically wraps these classes. For example, when `java.lang.ArrayList` is instantiated in the application code, CGMiner replaces the call so that a wrapped version of `ArrayList` is called, which can then be instrumented. The wrapped variant inherits from `ArrayList` and forwards all protected and public methods to their corresponding super class implementations. CGMiner not only wraps constructor calls, but also values returned by system classes, e.g. `ArrayList.iterator()`.

4.4 Dynamic Callgraph Analysis

For building the dynamic call graph, CGMiner instruments the app as follows. Before each call, a `CALL` event is sent from the device to the analyzer with the unique ID of the call site. After the call site, i.e., when the call has returned, a `RETURN` event is sent for the same ID. At the beginning of each method, a `ENTER` event is sent. Before each `return` or `throw` statement, a `LEAVE` event is sent. These events allow the analyzer to reconstruct the call edges taken on the device. Due to memory and performance constraints, CGMiner does not build the dynamic callgraph on the device. Once the events are sent, they are immediately discarded on the device.

The analyzer maintains a separate call stack for each thread. For each `CALL` event, the respective call site is put on the stack. When the analyzer receives an `ENTER` event, it creates a call edge from the top call site on the stack to the method that was entered. Note that a `CALL` event may be followed by multiple pairs of `ENTER` and `LEAVE` events before the `RETURN` event occurs. The Dalvik / Java runtime calls the static initializer of a class when the class is loaded. Therefore, each call may first invoke the static initializer before the actual callee is called. CGMiner captures this semantic by leaving the call site of the static initializer on the stack until the `RETURN` event is encountered. CGMiner does not consider implicit calls to static initializers from statements that are not call sites, e.g., from assignments to static variables. In this case, the static initializer generates an `ENTER` event, but has no matching call site on the stack. Therefore, the `ENTER` event is discarded. As CGMiner reconstructs call chains to callbacks, non-call initializations are not relevant.

4.5 Dynamic Taint Analysis

As explained in Section 4.2, CGMiner uses dynamic taint tracking to track the callback object through the program, including all container objects that hold this callback object. Note that only heap objects are tracked, no primitives. Therefore, the runtime code that gets instrumented into the app can uniquely distinguish each tainted object by its identity hash code (`System.identityHashCode()`). The instrumented runtime code stores a map between the unique ID of the taint source and the identity hash code[1], and also transmits this map to the analyzer on every change, i.e., whenever a new object is tainted. These transmissions occur on the background based on a transmission queue. The events are sent as one message whenever a certain number of events have accumulated. Therefore, the transmissions do not affect the performance of the original app.

[1] The implementation takes care of checking referential equality in case of hash collisions.

All field assignments are instrumented to perform *taint transfers*. If a variable is assigned to a field, the runtime code checks whether the variable on the right side of the assignment is tainted, i.e., its identity hash code is in the taint map. If so, the base object that contains the field is tainted as well, i.e., its identity hash code is written into the map with the same source ID as the variable. These *derived taints* are field-insensitive by design. If a library stores two different callbacks in two fields of the same object, this object is associated with both sources.

Recall that the dynamic taint tracking in CGMiner is special, since the object that is tainted at the source (the callback object) is always the same object that arrives at the sink (the `this` object inside the callback). The taint tracking is only used to track the path between where the callback was registered in the library and where the control flow arrives inside the callback. This allows for some imprecision in the taint tracking, because flows where source and sink object are not identical can be discarded.

The Java Standard Library cannot be instrumented, because it is pre-installed on the device and not part not part of the app. For such cases, CGMiner relies on the static taint data flow summaries from StubDroid [2]. Based on these summaries, CGMiner adds instrumentation at the call site in the user code rather than instrumenting the library itself.

4.6 Callback Summary Modelling

In this section, we describe the model that we use for the callback summaries throughout the rest of the paper. Note that this section only contains the general principle of a summary. We will use this model in Section 4.7, where we describe the algorithm for generating the callback summaries.

In the simplest case, we model callback summaries as a single "fake" call edge $a \rightarrow \langle b, c \rangle$ from an API call a to a callback method b. The target is a pair $\langle b, c \rangle$, where c describes the object on which method b is called at the call site a. Recall that applying a callback summary corresponds to a virtual dispatch in the context of the original API call. In other words, the target method is called either on the same base object as the original API method, or an object passed to the original API call as a parameter. As an example, consider `AsyncTask`. `AsyncTask` is a class that is commonly used in Android, which is part of Android's standard library, which is automatically available to every app. It is used to perform an action asynchronously, similar to a Java thread. Developers extend the `AsyncTask` class and override the `doInBackground` method, which is the callback method called in a background thread. In order to start the task, developers invoke the `execute` method on their `AsyncTask` instance. In the case of the `AsyncTask` class, the summary would be $AsyncTask.execute \rightarrow \langle AsyncTask.doInBackground, -1 \rangle$. The special value $c = -1$ stands for the base object of the call to *execute*, i.e., the `AsyncTask` object itself. A $c \geq 0$ would denote the cth parameter object of the caller statement using zero-indexing. Note that there may be more than one edge that originates in the same API method a. In the example, the Android OS also calls methods such as `onPostExecute` and `onPreExecute`, each of which is modelled as a separate summary edge.

The case from the example in Listing 1 is more complex, since the callback object is passed through multiple intermediate API calls, i.e. transfer functions. When applying the callback summary in a callgraph algorithm, i.e., when identifying the method that shall receive calls to `HttpLibrary.runAll`, the intermediate edges are processed in reverse order. The method `schedule` is called on the base object (index -1) of the previous call to `runAll`. The constructor of `HttpTask`, is called on the object that was the first argument (index 0) on the previous call to `schedule`. The original callback method `onCallback` is invoked on the object that was the second argument (index 1) in the previous call to the constructor of `HttpTask`.

In the callback summary, these intermediate calls are modelled as intermediate edges: $HttpLibrary.runAll \rightarrow\langle HttpLibrary.schedule, -1\rangle \rightarrow\langle HttpTask.cons, 0\rangle \rightarrow \langle IHttpCompleted.onCallback, 1\rangle$.

4.7 Callback Reconstruction

Algorithm 1 shows the details of how callback summaries are created. Function `BuildCallbackSummaries` is the main entry point that builds the callback summaries. It is called when the analyzer receives an event that callback method m has been called at runtime.

`BuildCallbackSummaries` uses the method `GetLastLibraryCallSite` to get the last library call site. `GetLastLibraryCallSite` in turn uses a helper method `GetDynamicTraces`, which returns a set of call traces that end in statement s by performing a graph search on the dynamic callgraph. The call graph is flattened into a set T of sequences of call sites c. Recursions are unrolled once in `GetDynamicTraces`, since repeating the same sub-sequence of calls does not provide any additional insights for the purpose of callgraph analysis. For not bloating the description, we do not present the implementation of the helper method `GetDynamicTraces` in the pseudocode.

Given a statement inside library code, method `GetLastUserCodeCallSite` uses the traces returned by `GetDynamicTraces` and returns the last user code statement that happened before and that transitively triggered the given library code statement. Similarly, method `GetLastUserCodeCallSite` takes a statement inside a callback in user code and identifies the last library statement that happened before and (transitively) invoked the given statement from the callback method. The helper method `Predecessor` takes a statement and returns the predecessor statement on the dynamic callgraph. For simplifying the presentation, we assume that this statement is unique. Our implementation can handle multiple candidates.

The main summary generator `BuildCallbackSummaries` first obtains the last statement in the library code before the callback was invoked (line 20). This is the last statement in the library before the control flow is passed back to user code, i.e., the *out* edge. The relevant interactions between user code and library API occur between this statement and the statement that originally passed the callback to the library (the *in* edge and source for the dynamic data flow analysis). We will explain the special case of $S_c = \epsilon$ (first branch in line 21) later. In line 25, CGMiner uses the method `GetPathsBetween` to query the dynamic taint graph for all taint paths between the two statements. A taint path is a sequence of statements that assigns a tainted variable or field to another variable or field, i.e., passes around a reference to a tainted object. This definition implies that method calls are part of the taint path as well, because they assign the value of the tainted argument at the call site to the corresponding parameter variable inside the callee. Note that there can be more than one path between source and sink, so $\mathbb{P}$ is a set of lists of statements. CGMiner first iterates over all paths and then over the statements in each path. It builds a new summary for each path. Hence, the initialization of the summary (line 27) is inside the loop over the paths.

The summary starts with the statement that passes the callback object to the library, i.e., the *in* edge. This statement is simply the source from which taints arrive in the callback method, as shown in line 27. Method `GetSource` returns the API at which the source was registered. The assignment statements on the taint path that copy around the callback object inside the library are not directly visible to the user code. Instead, the user code calls API methods that transitively trigger these statements through library-internal call chains. In the example, an assignment somewhere inside `schedule` or one of its transitive callees assigns the parameter with the task to a field. This assignment is on the taint path, but only the preceding call to the transfer method `schedule` is relevant as a part

Algorithm 1 Callback Reconstruction Algorithm.

```
1  Function GetLastUserCallSite(s):
       INPUT: s – the first statement in the callback
       OUTPUT: The last statement in user code

       T = GetDynamicTraces (s)
2      foreach t ∈ T do
3          foreach c ∈ t do
4              c' = Predecessor (c)
5              if not IsLibrary (GetMethod (c')) then
6                  if IsLibrary (GetMethod (c)) then
7                      return c'
8      return ϵ
9
10 Function GetLastLibraryCallSite(s):

       INPUT: s – the first statement in the callback
       OUTPUT: The last statement in library code

       T = GetDynamicTraces (s)
11     foreach t ∈ T do
12         foreach c ∈ t do
13             c' = Predecessor (c)
14             if not IsLibrary (GetMethod (c)) then
15                 if IsLibrary (GetMethod (c')) then
16                     return c'
17     return ϵ
18
19 Function BuildCallbackSummaries(m):

       INPUT: m – the callback method
       OUTPUT: A set of callback summaries

       Δ = ∅
20     S_c = GetLastLibraryCallSite (FirstStmt (m))
21     if S_c = ϵ then
22         φ =GetSource (m)
23         Δ = {φ → ⟨m, γ(m)⟩}
24     else
25         ℙ = GetPathsBetween (GetSource (m), S_c)
26         foreach p ∈ ℙ do
27             δ = ω(m)
28             foreach s ∈ p do
29                 S_u =GetLastUserCallSite (s)
30                 δ = δ ∘ ⟨ω(S_u), γ(S_u)⟩
31             Δ = Δ ∪ {δ}
32     return Δ
33
```

of the summary. In line 29, CGMiner uses the helper method `GetLastUserCallSite` to identify this corresponding API method by conducting a backward search in the dynamic callgraph. For statements that are already in user code, i.e., the first statement on the path, `GetLastUserCallSite` is an identity function.

Each identified transfer statement in user code maps to one fragment of a summary. In line 30 the current API method is concatenated to the summary built so far. For example, if the summary $\mathit{HttpLibrary.runAll} \rightarrow \langle \mathit{HttpLibrary.schedule}, -1 \rangle$ existed before, a new right arrow is appended to the next method and parameter index. As explained above, for the last statement of a taint path, $S_u = S_c$ holds, i.e., a taint path always ends with the API call that finally invokes the callback. S_c is the last library call site (line 20 in Algorithm 1), i.e., the last statement that was executed in the library before invoking the callback.

For extending the summary, CGMiner uses two helper functions: ω and γ. The ω method performs the generalization from concrete statements and methods to API interfaces. For call sites, ω retrieves the API signature. For callback methods, ω retrieves the name of the interface or abstract API class that declares the method. The γ method takes a call statement and identifies the tainted parameter, i.e., the parameter that contains the callback object, by querying the dynamic data flow graph. As explained in section 4.6, CGMiner uses the special value -1 if the base object of the call is tainted.

Note that Algorithm 1 also works for cases without transfer methods. Android's `AsyncTask.execute` method is part of the Android SDK, i.e., a pre-installed library on the device. It cannot be instrumented. Conceptually, the *in* edge points to a fake node (a method for which we have no implementation) and the *out* edge points from this node to the callback method `doInBackground`. In this case, the summary is a simple edge from a single API call site to a single callee method as explained in Section 4.6. In the algorithm, $S_c = \epsilon$ holds, and the first branch is taken in line 21. CGMiner retrieves the taint source, i.e., the *in* edge and construct an edge to the callback method `m`. The parameter index is derived from the taint graph using an overload of γ that processes the parameter variables of `m` instead of the call arguments at a call site.

4.8 Extensions and Special Cases

For simplicity, the algorithm presented in the pseudocode assumes that a single callback method is only connected to a single source, i.e., `GetSource` returns a single method. In other words, the developer does not re-use the same callback implementation for different independent API calls. Our implementation supports such re-uses.

Further, recall from Section 4.5 that CGMiner uses StubDroid summaries to model the effects of methods that cannot be instrumented in the dynamic taint analysis. In these cases, the transfer method cannot be found using a backwards search on the dynamic callgraph as shown in line 29. Instead, CGMiner marks these statements and directly uses statement s in such a case.

4.9 Applying Summaries

Many static analysis approaches require a callgraph. Computing the callgraph on the application code as well as the code of all libraries required by the application can require significant computational resources and time. Therefore, it makes sense to replace the libraries by summaries. These summaries must capture the control flow of the library with respect to its external interface, i.e., it must correctly model callbacks back to the application code. CGMiner generates such summaries. They can applied whenever a callgraph is needed on an application that uses a library for which a summary was previously computed.

As such, we want to apply the generated callback summaries during the callback construction. In the case of the motivating sample in Listing 1, we want to apply the edge summary $HttpLibrary.runAll \rightarrow \langle HttpLibrary.schedule, -1\rangle \rightarrow \langle HttpTask.cons, 0\rangle \rightarrow \langle IHttpCompleted.onCallback, 1\rangle$. In this section, we introduce Algorithm 2. In the case of the motivating example, the algorithm outputs an edge from `runAll` to the anonymous implementation referenced by `onComplete` and `onFailed`, resulting in a precise callgraph. This shows that we need the intermediate edges in order to determine the link between the implementation supplied at the constructor call (referenced by `onComplete` in the sample) and the call to `runAll`. Without these intermediate edges we have no information on the actual type of the callback object at the callback invocation site. Without such information, we would need to create edges from `runAll` to all possible implementations of `ICompleted`, even if they are not used as a callback.

Given a call site s and a set of callback summaries Δ, method `FindReceivers` in Algorithm 2 enumerates the potential callees at s. `FindReceivers` performs a traditional callgraph search via `QueryCallgraph`. It then augments these callees with the callbacks that are found by applying the callback summaries.

Algorithm 2 Summary Application Algorithm.

Function `FindReceivers`(s, Δ)**:**
  INPUT: s – the call site for which to find the receivers,
  $\delta := \langle\alpha_1, \beta_1\rangle \rightarrow ... \rightarrow \langle\alpha_n, \beta_n\rangle$
  – the callback summaries
  OUTPUT: The potential callees for the given call site

  $\mathbb{R} = \{$ `QueryCallgraph` (s) $\}$
  foreach $(\delta := (\omega(s) \rightarrow ... \rightarrow \langle\alpha_n, \beta_n\rangle)) \in \Delta$ **do**
    $\hat{\delta} := \langle\alpha_1, \beta_1\rangle \rightarrow ... \rightarrow \langle\alpha_1, \beta_1\rangle$
    $\mathbb{R} = \mathbb{R} \cup$ `ApplySummary` $(s, \hat{\delta}, -1)$
  return $\mathbb{R}$

Function `ApplySummary`(s, δ, i)**:**
  INPUT: s – the call site for which to find the receivers, δ – the summary, i – the argument index
  OUTPUT: The potential callees for the given call site

  $\hat{\delta} := \langle\alpha_2, \beta_2\rangle \rightarrow ... \rightarrow \langle\alpha_n, \beta_n\rangle$
  $v =$ `VariableOf`(s, i)
  $\mathbb{S} =$ `GetCallsOn`(v)
  $\mathbb{R} = \emptyset$
  foreach $\hat{s} \in \mathbb{S}$ **do**
    if $\hat{\delta} = \epsilon$ **then**
      $\mathbb{R} = \mathbb{R} \cup \{\kappa(\hat{s}, v)\}$
    else
      `FindReceivers` $(\hat{s}, \hat{\delta}, \gamma(\hat{s}, v)))$
  return $\mathbb{R}$

For applying the callback summaries, line 3 iterates over all summaries δ in the database Δ. It looks for those summaries that reference the API call from the given statement. Recall from Section 4.7 that $\omega(s)$ extracts the generic API method signature from a concrete statement. Each summary is applied using method `ApplySummary`. Note that the source statement $\omega(s)$ is removed from the sequence of calls inside the summary and only the remaining calls are passed. Method `ApplySummary` processes these intermediate calls recursively and removes one call per iteration until the final call, i.e., the one that invokes the callback method, is found. We included the structure of δ in line 1 for clarity. Line 9 shows the derived $\hat{\delta}$ with the first element removed from the summary.

For the structure of the individual calls on the summary, recall from Section 4.6 that the first element a encodes the API method, and b encodes the base object on which the API method is called. $b = -1$ refers to a call on the base object, $b \geq 0$ references the parameter with the respective index.

Method `VariableOf` in line 10 obtains the variable v that corresponds to index i in the context of statement s. CGMiner then obtains all virtual call sites $\hat{s} \in \mathbb{S}$ where variable v is the base object using method `GetCallsOn`. For each of these call sites, CGMiner continues the search for the element of the call summary using a recursive call to `ApplySummary` in line 16. Method γ takes a statement, which must be a call site, and a variable, and returns the index of that variable in the argument list of the call (or -1 if the variable is the variable is the base object of the call).

The recursion ends if the summary has no further calls to analyze (line 14). Method κ takes a statement and a variable, e.g., `s.onCallback()` and s. It returns the method that is called (`onCallback` in this case), which is the final callee that is added to the callgraph. Note that `ApplySummary` calls `FindReceivers` again once the statement and variable of the callback are known. This is necessary, because callbacks usually rely on virtual dispatch, i.e., the actual receiver depends on the possible types of the base object. In the example from Listing 1, multiple classes could implement `IHttpCompleted` and depending on some conditional, variable `completedCallback` could be any one of them at line 11. CGMiner detects that `completedCallback.onCallback` is line 11. Finding the final receivers of this virtual call is an orthogonal problem and CGMiner relies on the existing callgraph algorithm.

CGMiner only summarizes callgraph data and must be extended with summaries that capture the semantics of the client analysis. CGMiner integrates well with StubDroid [2], which summarizes data flow, but does not address control flow.

5 Implementation

We run the sample apps on real devices using DFarm [19]. For instrumenting the code and interacting with the devices, we rely on the VUSC commercial code analyzer. VUSC provides an API for instrumenting value requests into Jimple [25] code and for associating the events received at runtime with the Jimple statements at which they were generated. The device communication is derived from FuzzDroid [23] and uses Soot for instrumentation [3].

The runtime overhead of the additional code injected by CGMiner is not relevant as long as the app does not crash with an Application Not Responsive (ANR) exception. The Android system automatically sends ANRs when foreground threads (such as the UI thread) are blocked for an extended amount of time. In order to avoid ANRs, we queue events in the corresponding thread in which they occur and sent them asynchronously. The communication with the control computer happens in a separate thread controlled by an Android Service.

For the list of library classes that serves as an input to CGMiner, we crawled the Maven central repository as well as the Google Gradle repository. To limit the size of the database, we only include libraries that are referenced by at least five other Maven artifacts. For these relevant libraries, we extract the package names of all classes contained in the respective JAR file. When running CGMiner, we consider a class to be a library class when its is contained in one of these known library packages. Note that library identification is orthogonal to the callback analysis, and CGMiner is agnostic to how the list of library classes is built.

6 Limitations

CGMiner instruments the Dalvik code inside an app. If parts of the control flow between API call and callback are implemented in native code, no runtime data can be obtained from these parts. If the native code contains border edges, the callback summary will be incomplete. If a taint transfer occurs in native code, CGMiner relies on StubDroid summaries, which exist for methods from the Java Standard Library, such as `System.arrayCopy`.

Note that Android requires each APK file to be signed. Therefore, when instrumenting an app, the app needs to be resigned. Since CGMiner modifies the app for the dynamic analysis, it must be signed anew before it can be installed on the device. If the app performs integrity checking, these checks will fail. While the individual app cannot be analyzed in this case, the CGMiner approach still works if, for each library, enough apps that use the respective library can be processed.

Not every callback may be invoked in each run of each app. In our example in Listing 1, the error callback is only invoked if the HTTP connection fails. Since the callback summaries are merged over many apps in CGMiner, we consider edges that are never triggered even with dozens of apps to be irrelevant in practice.

Our evaluation is partly based on Monkey [10] for exploring the apps' user interface. Monkey is part of the official Android SDK and randomly clicks on the screen for a given amount of time. Note that CGMiner is agnostic to the input generation tool. It can be replaced with a more capable approach in future work. We also used manual exploration in order to augment the automatic analysis.

We currently do not consider Android lifecycle methods such as `onCreate`, as they are few, well-known, change rarely, and are already precisely modeled, e.g., in FlowDroid [5].

7 Evaluation

In this section, we evaluate CGMiner with regard to the following research questions:

RQ1 How many callback edges does CGMiner identify?
RQ2 Are the callback summaries correct and complete?
RQ3 How long does the instrumentation take?
RQ4 How often do transfer functions occur?
RQ5 How does CGMiner compare to EdgeMiner?
RQ6 Which summaries have been found (case study)?
RQ7 How do summaries affect data flow analysis?

7.1 Experiment Setup

We used a machine with 144 Intel Xeon Gold 6154 CPU cores and 3 TB of physical memory using OpenJDK 16. A maximum of 50 GB was assigned as Java heap space. The machine was chosen due to the performance requirements of FlowDroid for RQ6. Our DFarm installation

is equipped with around 90 devices in total, comprising Samsung Galaxy XCover Pro phones distributed over 9 device controller boards and a single master controller. Note that each run of CGMiner only uses a single device. We use a combination of manual and automatic exploration. For automatic exploration, we used the Monkey tool from the Android SDK to explore the user interface of the app at runtime. Despite its simplistic approach, Monkey achieves code coverage results comparable to more complex approaches [9]. We run each app for five minutes with automatic and the same time using manual exploration. In apps where a login was needed in order to proceed the exploration of the app, we manually created user accounts.

For our callback generation, we randomly collected 700 apps from the Google Play Store between 2008 and 2021, augmented with apps from AndroZoo [1]. We include older apps to merge the callback summaries over multiple versions of a library, and to also include error cases, e.g., failing HTTP connections due to the server no longer being operational. In our experience, newer versions of libraries return the old methods (including their callbacks) for backwards compatibility. On the other hand, new versions may introduce new additional methods with callbacks, requiring us to run CGMiner again on the new version.

For research questions 2 and 5, we inspected callback summaries manually. Two researchers conducted the manual inspection. Upon disagreeing, a third researcher has been involved and these cases were discussed until a consensus was reached.

7.2 Baseline over the Dataset

To better understand the performance of CGMiner, we measure the sizes of the original apps in our dataset, i.e., before the instrumentation. The number of classes ranges from 6 to 37,175 with an average of 14,371 and a median of 13,136. The apps contain between 108 and 226,966 methods, with an average of 85,270 and a median of 69,360 methods. In the Jimple intermediate representation, the apps contain between 693 and 2,793,272 units (i.e., Jimple instructions), with an average of 1,107,276 and a median of 1,008,848 units.

7.3 RQ1: Number of Generated Callbacks

From the 700 apps in our dataset, CGMiner created callback summaries for 338 apps. Not all apps contain callback-driven libraries according to our definition. Hybrid apps, for example, implement their logic in JavaScript and only present HTML content to the user via Android's `WebView` component. Other apps use libraries that our library detection does not recognize, or simply do not use callbacks. Some apps contain native code, which is not supported in CGMiner. Recall that CGMiner creates summaries for libraries rather than apps. Furthermore, some apps are merely add-ons such as themes for other applications and do not have any launchable main activity. Therefore, as long as a single app uses the library's callback-driven API, a summary can be generated.

In total, CGMiner constructed 1,476 summaries, which is around 8 summaries per app on average. Figure 3 shows the cumulative distribution of the number of callbacks found per app. The x axis is the number of edges, and the y axis shows how many apps lead to the given number of edges. The maximum number of edges obtained from one app is 67, the minimum is zero, with a median of 5. For each summary, we recorded the number of API methods that must be called to invoke the callback. On average, one call is required, with a maximum of 2 calls and a minimum of one call. The median is one call.

To augment our callback summaries, we automatically generated artificial apps in an attempt to trigger the callback candidates for which CGMiner did not yield an edge on our original app set. This is a best effort approach. We accept that some of these apps will

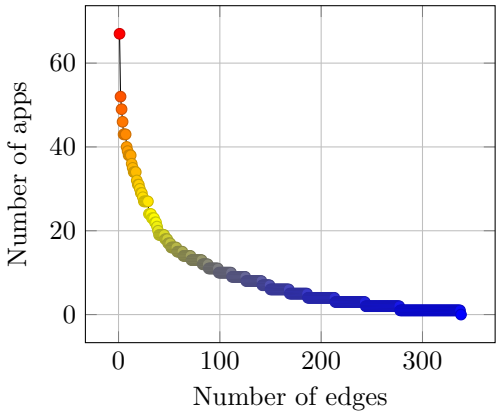

Figure 3 Distribution of callback edge counts over the apps.

crash or fail to invoke the callback. Recall that callback candidates are over-approximated, i.e., it may be impossible to generate a working app for some candidates. However, since CGMiner is a dynamic approach, broken apps do not lead to false positives in the callback summaries generated by observing these apps. The generated artificial apps yielded 1,871 edges.

7.4 RQ2: Correctness of Generated Callbacks

We manually verified the callback summaries generated by CGMiner. We merged the summaries from apps in the dataset with the generated apps mentioned in RQ1. We found 94.62% of all callback summaries to be correct. 38 edges out of 2046 were false positives. Transfer statements were missing in 72 cases.

Since CGMiner is a dynamic approach, it is inherently an underapproximation. To better understand the degree of unsoundness, we manually inspected a random subset of 100 callback candidates which for which CGMiner did not find an edge. We found that 95% of these callback candidates are indeed not callback methods. 5% were callback candidates that were missed due to not triggering the respective method in an app at runtime,i.e., actual false negatives.

Another approach to check for missing callbacks is to use benchmark suites. To our best knowledge, there is no ground truth benchmark specifically for callback edges. Therefore, we used the artificially generated apps introduced in Section 7.3 as a base. On these apps, CGMiner retrieved edges for 82 % of the callback candidates. For 5%, the generated apps missed at least one method call to actually trigger the callback. For the rest, these are not valid callbacks, i.e. these are true negatives.

7.5 RQ3: Instrumentation Performance

Our implementation of CGMiner integrates into an analysis framework that schedules jobs for processing and performs them when free capacity is available on a system consisting of analysis server, DFarm device farm server, DFarm controllers, etc. We therefore measure the performance of the relevant parts of the analysis individually, because the overall time is dominated by the infrastructure.

First, an APK file is imported into the analysis framework and its code is transformed to Jimple. This step takes 59 seconds on average (minimum: 8s, maximum: 130s, median: 54s). Note that this time also includes decoding the app's resource files and manifest, because

the instrumentation framework assumes that they can be modified as well. In fact, the framework injects an application class (if not yet present), services, and permissions as part of the communication infrastructure between device and analysis server.

After the app has been imported, the instrumentation is performed, which takes 9 seconds on average (minimum: 4s, maximum: 14s, median: 4s). This time includes the part specific to CGMiner, i.e., defining the callback events. The CGMiner part alone never takes more than one second with an average of 0.3 seconds and a mean of 0.2 seconds. Translating the callback event definitions into statements, along with the other required modifications to establish the connection between device and analysis host, is part of the VUSC analysis framework. It counts into the 9 seconds and not the one second. On average, the overall analysis performs 432,000 instrumentation steps (maximum: 556,000 steps, minimum: 302,000 steps, median: 396,000 steps). Each step can be a single statement added or removed, a change to a value in a statement, etc.

Next, the transformed Jimple code and resource files including the manifest are written back into an APK file. This step takes 44 seconds on average (maximum: 52s, minimum: 27s, mean: 43s). The total time spent before running the app is 112 seconds on average (minimum: 40s, maximum: 208s, median: 106s). After building, we run the apps for a fixed period of time and send inputs manually and afterwards by using Monkey. Therefore, measuring the runtime performance is not informative. We observe that the apps still meet the responsiveness requirements of the Android operating system.

We conclude that CGMiner's runtime is dominated by the dynamic exploration (5 minutes in our configuration), and not by the analysis and instrumentation beforehand (roughly 2-3 minutes). Note that CGMiner is intended to be used as a tool to generate callback summaries as a one-time effort. The performance numbers shown correspond to the time needed to generate the summaries. In contrast, applying the callback summaries generated by CGMiner does not require any dynamic analysis.

7.6 RQ4: Prevalence of Transfer Functions

In contrast to previous approaches from the literature [8], CGMiner supports complex callback registration that require transfer functions. In this research question, we evaluate how important transfer edges are in real-world apps. Conceptually, disregarding transfer functions via approximations may to lead to a loss of flow-sensitivity as well as false positive callgraph edges as shown in Section 3.

During callback identification (see Section 7.3), CGMiner discovered CGMiner 2046 edges, 146 of which require transfer functions (6.00%). Note that these numbers are on API level. Even a single transfer edge can be highly important if the respective API is used frequently in apps.

To measure the impact of these 146 edges, we therefore check how often these APIs that require transfer functions are called in real-world apps. To avoid any bias from the apps on which the transfer edges were originally identified, we chose a separate evaluation dataset. We randomly picked 1988 apps from the same Play Store and AndroZoo data source explained in Section 7.1. On this app set, 1928 apps (96.98 %) use transfer functions in their code. On average, each app uses 103.65 different transfer functions. For comparison, apps in the dataset use 1089.24 callbacks on average. These results show that transfer functions are highly relevant in practice.

Table 2 shows the ten most frequently-used transfer functions and their edges together with the number of times the respective transfer function was encountered in our evaluation app set. Six of the most found functions are related to wrapped IO calls. For example, a `read` method call on an `BufferedReader` instance triggers the `read` callback on the reader which was specified during the construction of the `BufferedReader` object.

Listing 3 Transfer Function Code.

```
1 StringReader sr = new StringReader(str) {
2   public void close() {
3     // additional callback code
4     super.close();
5   }
6 };
7 BufferedReader br = new BufferedReader(sr);
8 br.read();
```

In other words, the constructor of the `BufferedReader` is a transfer function. Listing 3 shows a code example for such a case. Without modeling the transfer function, approaches such as EdgeMiner must model an edge from the call to the `BufferedReader` constructor in Line 7 to all methods of the `Reader` that is passed as the first argument. In the example, this would even be a call to `close`, even though the `StringReader` is never closed[2].

In total, 19.55 % of the callbacks that EdgeMiner has identified require transfer functions. All of them are missed. In contrast, CGMiner only misses 3.52% of the transfer functions that are required for the callbacks identified by CGMiner.

7.7 RQ5: Comparison with EdgeMiner

For a comparison on the Android system we used Android 4.2, since the since the Edgeminer paper used Android 4.2 for evaluation. EdgeMiner yields 5,125,472 edges in total for Android 4.2, whereas CGMiner yields 2046 edges. We found that the EdgeMiner output contains reference to non-existing parameters or callbacks with incompatible types. First, we removed these edges automatically. Furthermore, we noticed that EdgeMiner's output may contain multiple callback edges overloads referencing all implementations albeit an edge for the abstract superclass or interface was enough. We therefore removed the edges of these overloads automatically and made sure that the removal process does not change the semantics. After this cleanup, 17298 callback edges remain (0,36% of the original edge set). This constitutes as our new base set for EdgeMiner, which we verified manually.

On this base set we compute a false positive rate of 47.42% for EdgeMiner. Manually checking the CGMiner edges only yields a false positive rate of 1.86%. CGMiner's dynamic analysis avoids the false positives that arise from EdgeMiner's VTA callgraph and the resulting imprecise points-to set for that is used to derive the types of registers / variables that store callback objects. We make available the annotated outputs of EdgeMiner and CGMiner as part of our data package. We removed Android APIs not present in Android 4.2 from CGMiner results, since EdgeMiner cannot possibly have results involving these APIs and apps in RQ1 may use newer API methods than those present in Android 4.2.

Note that EdgeMiner's data is based on the Android system's implementation JAR alone without third-party libraries. For a fair comparison, we used the library detector integrated in VUSC to obtain maven coordinates of libraries used in apps in RQ1. We downloaded the JAR files of the library and executed EdgeMiner on these JARs. Table 1 shows the results for the Android system (comprising the Android SDK and the Java standard library) as well as third party libraries. While EdgeMiner has more edges on

[2] The `StringReader` has no finalizer either that would call `close`.

the Android system jar, it has significantly more false positives and significantly more incomplete edges than CGMiner. Incomplete edges are edges that lack one or more necessary transfer functions. For example, in the motivating example of Section 3, an edge $\langle HttpTask.cons, 0\rangle \rightarrow \langle IHttpCompleted.onCallback, 1\rangle$ would be incomplete, because it is missing the necessary transfer edges to `schedule` and `runAll`.

To get a better understanding of the sources of imprecision, we analyzed the false positives and the incomplete edges produced by EdgeMiner in detail. Setters and constructors are particularly relevant sources of imprecision. In total, EdgeMiner reports 2826 constructor edges and 1516 setter edges. EdgeMiner places edges from these methods to the callbacks. In reality, however, these methods do not invoke any callback function, neither directly or transitively. Instead, the references to callback objects are saved into fields. Only later, when other methods are called, these references are read back from the field and the respective callback is invoked.

Table 1 Results on different libraries for CGMiner and EdgeMiner. TP: true positive edges, FP: false positive edges, IE: incomplete edges (missing transfers). Regarding "Other": We have included several other libraries, which we made sure to supply to EdgeMiner as well.

	CGMiner			EdgeMiner		
Library	**TP**	**FP**	**IE**	**TP**	**FP**	**IE**
Android	1051	27	21	4957	7704	2586
Java	574	8	46	702	494	709
Apache HttpClient	59	0	0	14	1	0
kotlin	46	0	0	0	0	0
Xml Pull Parser	36	0	0	41	4	86
Apache HttpCore	12	0	0	0	0	0
Rxjava	9	0	0	0	0	0
play-services-ads-lite	8	0	0	0	0	0
Gson	7	1	0	0	0	0
Firebase	5	0	0	0	0	0
Google common	4	2	5	0	0	0
play-services-basement	2	0	0	0	0	0
play-services-maps	2	0	0	0	0	0
C3DEngine	1	0	0	0	0	0
AndEngine	1	0	0	0	0	0
Cocos2dx	1	0	0	0	0	0
Other	118	0	0	0	0	0
Total	1936	38	72	5714	8203	3381
Rate	94.62%	1.86%	3.52%	33.03%	47.42%	19.55%

We next describe some examples of such false edges. One constructor of the `ConcurrentSkipListSet` class, for example, takes a `Comparator` as a parameter. A `ConcurrentSkipListSet` is a sorted set, which orders elements according to this comparator. The constructor only saves the comparator instance to a field, and the callback is triggered when a new element is inserted into the set using the `add` or `addAll` methods. EdgeMiner places an edge from the constructor to the Comparator's `compare` method, although these methods are only called upon adding an element. In total, EdgeMiner reports incomplete edges in 1734 out of the 2826 constructor edges, and 779 are false positives (11.08 % correctness rate). In contrast, CGMiner

yields only 13 incomplete and 16 false positive edges on 323 constructor edges (91.02 %). Similarly, most setters set a field to a specific value and do trigger callbacks. For example, EdgeMiner assumes an edge from `LayoutInflater.setFactory(LayoutInflater$Factory)` to `LayoutInflater$Factory.onCreateView`, although this is only the registration site of the call. Android calls the callback only upon inflating a layout using the `inflate` method. Since EdgeMiner does not support transfer edges, it misses the corresponding transfers on these edges. For EdgeMiner, out of 1516 setter edges, 663 are incomplete and 195 are false positives. This constitutes a correctness rate of 43.4 % on these edges for EdgeMiner. On 467 edges on setter methods reported by CGMiner, 21 are incomplete and 0 false positive, resulting in a correctness rate of 95.5 %.

7.8 RQ6: Case Study on Individual Callbacks

Using CGMiner, we have identified non-obvious multi-step callbacks. The *ActionBarSherlock*[3] library allows a developer to integrate a tab view into his app. New tabs are added using `addTab` on an `ActionBar` object which takes the tab as a parameter. With `Tab.setTabListener`, the developer can register a callback that is notified when the user selects the tab. Therefore, `addTab`, which automatically opens the new tab, triggers the the `onTabSelected` callback previously registered on the tab. This callback involves two interactive objects, `ActionBar` and `Tab`. Other tools such as EdgeMiner [8] cannot precisely identify and model such a callback. In case of the `AsyncTask`, CGMiner detects that a call to `AsyncTask.execute` results in several callbacks being called: `onPreExecute`, `doInBackground`, `onPostExecute`.

CGMiner identifies similar API calls that trigger multiple callbacks in the API for the SQLite database engine. A call to `getWritableDatabase` or `getReadableDatabase` triggers the callbacks `onOpen`, `onConfigure` and `onCreate`. Some callbacks are triggered by the operating system upon external events, such as new sensor data. In this case, the last user code call site for this callback is the statement the registered the callback. Even though this statement does not immediately invoke the callback, modeling an edge from the registration site to the callback is still a common and useful approximation. CGMiner, for example, finds a connection between Android's `registerListener` method and the `onSensorChanged` of the `SensorEventListener` interface.

7.9 RQ7: Effect on Client Analysis

We next evaluate the effect of callback summaries on data flow analysis. We ran FlowDroid on 200 randomly selected apps, chosen from the same data source already explained in Section 7.1. Note that this data flow analysis is distinct from the data flow analysis we perform in our approach in Section 4.5. The purpose of the data flow analysis in Section 4.5 is to track all container objects that hold callback objects. This is only relevant when generating new summaries. In contrast, this section performs data flow analysis to determine sensitive flows. For this, we use already computed summaries from Section 7.3 to extend the call graph. We configured a timeout of 3 minutes for callgraph construction and 15 minutes for the main data flow analysis. The analysis was assigned 250 GB of heap space and a maximum of 7 cores. This configuration allowed us to parallelize multiple runs on the same machine. We evaluated three different configurations. As our baseline, we perform the FlowDroid data flow

[3] `http://actionbarsherlock.com/`

Table 2 The ten most found transfer functions in apps.

Count	Transfer function & Edge
1161	$BufferedReader.readLine \rightarrow \langle BufferedReader.cons, -1\rangle \rightarrow \langle InputStreamReader.read, 0\rangle$
1071	$Runnable.run \rightarrow \langle FutureTask.cons, -1\rangle \rightarrow \langle Callable.call, 0\rangle$
1031	$BufferedInputStream.cons \rightarrow \langle GZIPInputStream.cons, 0\rangle \rightarrow \langle AutoCloseable.close, 0\rangle$
995	$InputStream.read \rightarrow \langle BufferedInputStream.cons, -1\rangle \rightarrow \langle FileInputStream.read, 0\rangle$
983	$PrintWriter.print \rightarrow \langle PrintWriter.cons, -1\rangle \rightarrow \langle OutputStreamWriter.write, 0\rangle$
980	$View.layout \rightarrow \langle View.addOnLayoutChangeListener, -1\rangle \rightarrow \langle View\$OnLayoutChangeListener.onLayoutChange, 0\rangle$
977	$Executor.execute \rightarrow \langle ScheduledThreadPoolExecutor.cons, -1\rangle \rightarrow \langle ThreadFactory.newThread, 1\rangle$
963	$OutputStream.write \rightarrow \langle CipherOutputStream.cons, -1\rangle \rightarrow \langle ByteArrayOutputStream.write, 0\rangle$
945	$PrintStream.println \rightarrow \langle PrintWriter.cons, -1\rangle \rightarrow \langle FileWriter.write, 0\rangle$
920	$Parcel.writeBundle \rightarrow \langle Parcel.writeStrongBinder, -1\rangle \rightarrow \langle ffm.dispatchTransaction, 0\rangle$

analysis without any callback edges. We then ran FlowDroid again with callback summaries generated by EdgeMiner and with summaries generated by CGMiner. For each run, we recorded the discovered flows.

Listing 4 Callback Parameter Analysis.

```
1  class MyTaskRunnable implements Runnable {
2    public String data;
3    public void run() {
4      sink(data);
5    }
6  }
7  ThreadPoolExecutor executor = new ThreadPoolExecutor(...);
8  Runnable r = new MyTaskRunnable();
9  r.data = source();
10 executor.execute(r);
```

Recall that CGMiner and EdgeMiner only model control flow, but not data flow. In the example in Listing 4, the first parameter of Line 10 becomes the base object inside the callee `run`. This relationship is important, because the field `data` inside the callback object, i.e., the access path `r.data`, is tainted in Line 9. When the data flow analysis processes the sink call in Line 4, the taint must be available as `this.data`. In other words, FlowDroid's IFDS call edge must re-write the access path from `r.data` to `this.data`.

Neither CGMiner nor EdgeMiner create data flow summaries. Therefore, our initial runs had the required callgraph edges, but could not track data flows across the callback edges. With this configuration, our baseline yielded 2021 flows. With EdgeMiner summaries, FlowDroid found 3575 flows (77 % more than baseline). With CGMiner summaries, 2554 flows were detected, which is 26 % more than the baseline. As expected, FlowDroid discovers

more flows when provided with callback summaries. Further, since EdgeMiner has more (true positive) callback edges than CGMiner as shown in Table 1, it is unsurprising that EdgeMiner leads to more flows as well. FlowDroid tracks flows across the interprocedural data flow graph. Every additional callgraph edge has the potential to lead to more flows.

We next augment the callgraph summaries with data flow information using heuristics. Firstly, we map the base object on which the callback is invoked (which is known from the callback summary) to the `this` object of the callee. In the example in Listing 4, this leads to a data flow edge from variable `r` to the `this` object of the callback. This allows FlowDroid to map `r.data` to `this.data`. Secondly, if there is an edge from a call site to a callback method and the call site accepts a parameter that is cast-compatible to a parameter of the callback method, we assume a data flow edge. We stress that these heuristics are not meant to be complete. We use them as part of our evaluation to better estimate the effect of callback edges for data flow analyses.

With these data flow mappings, FlowDroid finds 2717 flows with EdgeMiner and 2830 flows (40 % more than the baseline) with CGMiner. In the baseline without callback summaries, no data flow mapping is possible. We observe that when we use parameter mappings, FlowDroid with CGMiner finds more data flows than FlowDroid with EdgeMiner, although CGMiner has vastly fewer callgraph summary edges than EdgeMiner. The number of flows found using EdgeMiner callbacks drops from 3575 flows when using no parameter mapping to 2717 when using parameter mappings. The explanation lies in FlowDroid's sanity checking. For example, when FlowDroid propagates taints along edges in the interprocedural control flow graph, it propagates types along as well. In each propagation step, these propagated types are checked for cast-compatibility with the target variables. EdgeMiner's spurious callback edges lead to many cast-incompatible propagations. This leads to taints being discarded. For EdgeMiner with many false positive edges or incomplete edges, i.e., edges placed at the wrong statement, this leads to a significant amount of flows being discarded. On the other hand, the increase in data flows with CGMiner summaries represents actual taint propagation along the callback edges. This is expected for examples such as the one shown in Listing 4. Since CGMiner only has few false positives, it is almost unaffected by FlowDroid's type checks, but benefits from parameter mappings being available. We then analyzed the correctness of the data flows. Due to the large amount of data flow results, we only looked at a subset of 50 flows of each evaluation run. EdgeMiner shows a precision of 94.34% on these flows. Recall that FlowDroid already discards flows with cast-incompatible assignments along the taint propagation path. Therefore, it can eliminate some false-positive flows during propagation. CGMiner delivers a true positive rate of 100%.

As stated in the beginning of this section, we evaluated on FlowDroid using 200 randomly selected apps. From these apps, we found that 94 % invoke at least one callback method. Filtering apps with no callback methods yields the following results: Without data flow mappings, the baseline has 1,987 flows (compared to 2,021 flows w/o filtering). With EdgeMiner summaries, FlowDroid finds 3,505 flows (compared to 3,575). With CGMiner summaries, FlowDroid finds 2,511 flows (compared to 2,554). With data flow mappings and EdgeMiner summaries, FlowDroid then finds 2,681 flows (compared to 2,717). With data flow mappings and CGMiner summaries, FlowDroid finds 2,787 flows (compared to 2,830).

8 Related Work

EdgeMiner [8] statically analyzes the Android framework to build models for callbacks in API methods. Due to the large code size of the Android framework, EdgeMiner over-approximates virtual dispatch using a CHA callgraph. It further suffers from the inherent challenges

of static analysis, such as dealing with reflective method calls. CGMiner avoids such imprecision and only generates edges that are possible at runtime. EdgeMiner tries to find registration and callback pairs using def-use chains. The search starts at a potential callback and follows the definitions of the base object through the library code until it reaches the start of a method that has no more potential callers within the library. In case the callback object is read from a field on the path, all writes to the field are considered as potential definitions and are thus followed, regardless of their context. EdgeMiner does not provide support for collections or arrays and would not be able to generate a correct summary for our example from Listing 3. Perez and Le [20] present Predicate Callback Summaries (PCS) that model under which conditions a callback or Android lifecycle method is invoked. Their static tool Lithium works on the Android source code and suffers from the same challenges of large-scale static callgraph analysis as EdgeMiner. It does not support our complex example either. Callback Control Flow Automata (CCFA) [21] integrates PCS and Window Transition Graphs (WTGs) [27], and focuses mainly on UI callbacks and lifecycle methods. We consider integrating a predicate analysis into CGMiner as future work. Zhang and Ryder [31] propose a static library analysis based on data reachability. Similarly, Guo et al. present an approach based on backward data dependency analysis [12]. These analyses must be conducted for each call site, which is costly in practice [15].

Some work has focused on Android UI callbacks [28], e.g., for context-sensitive linking of parameterized callbacks to their respective UI elements. The same callback may be used for multiple buttons. The clicked button is passed to the callback as a parameter, and the shared implementation may follow different control flow paths depending on that parameter value. The information which API methods may trigger callbacks is usually an external input to these algorithms, which CGMiner can provide. Other work has increased the coverage of dynamic analyses by reasoning about UI callbacks using a combination of static and dynamic analysis [6, 29]. CGMiner is more generic and therefore cannot exploit specific properties of Activities or Intents. TamiFlex [7] uses dynamic analysis to record the runtime values at reflective method calls and build models of known callees for such call sites. Harvester [22] uses static slicing and dynamic execution to extract runtime values at reflective call sites. It rewrites these call sites into explicit calls to deobfuscate apps. The outputs of these approaches are specific to a concrete target program and do not generalize over re-usable libraries. HeapDL [11] uses heap dumps to reconstruct callgraph edges. It can be used do discover callback registration methods, which directly or transitively call callback methods when such a call is present on the stack of some thread in the heap dump. However, when a callback is saved as a field during callback registration and used later on, this approach would require multiple heap dumps taken at precisely the correct timings. Otherwise, either the callback registration, the callback invocation or both are missed. StubDroid [2] statically generates data flow summaries for libraries. It requires a complete and precise callgraph of the library to work properly and can therefore benefit from the callgraph models generated by CGMiner. Our callback summaries are relevant for various analyses (power analysis, privacy analysis, injection analysis, etc.) that currently rely on manual callback models [5, 30, 18, 13].

9 Conclusion

We have presented CGMiner, an approach for dynamically monitoring apps to derive callback summaries for commonly-used libraries. These summaries can then be applied to static analyses that require a callgraph. We have shown that CGMiner yields a precision of more than 94%. With the CGMiner summaries, FlowDroid detects 40 % more flows in comparison to our baseline. In the future, we will run CGMiner on more apps to generate and provide to the community summaries of lesser-used libraries.

Data Availability. The data and implementation have been published to `https://github.com/Fraunhofer-SIT/DynamicCallbackSummaries/`. Since CGMiner is built upon the VUSC commercial scanner, you need to apply for a free academic license for VUSC to build and run CGMiner.

References

1 Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.

2 Steven Arzt and Eric Bodden. Stubdroid: automatic inference of precise data-flow summaries for the android framework. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 725–735. IEEE, 2016.

3 Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Instrumenting android and java applications as easy as abc. In *International Conference on Runtime Verification*, pages 364–381. Springer, 2013.

4 Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013.

5 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

6 Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 641–660, 2013.

7 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 241–250. IEEE, 2011.

8 Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.

9 Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440, 2015. `doi:10.1109/ASE.2015.89`.

10 Google, Inc. Ui/application exerciser monkey, 2023. URL: `https://developer.android.com/studio/test/other-testing-tools/monkey`.

11 Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: countering unsoundness with heap snapshots. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27, 2017.

12 Chenkai Guo, Quanqi Ye, Naipeng Dong, Guangdong Bai, Jin Song Dong, and Jing Xu. Automatic construction of callback model for android application. In *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 231–234. IEEE, 2016.

13 Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L Pereira, Gilles A Pokam, Peter M Chen, and Jason Flinn. Race detection for event-driven mobile applications. *ACM SIGPLAN Notices*, 49(6):326–336, 2014.

14 Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, oktober 2011.

15 Ondrej Lhoták. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 37–42, 2007.

16 Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 153–169. Springer Berlin Heidelberg, 2003. doi:10.1007/3-540-36579-6_12.

17 Li Li, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 403–414, 2016. doi:10.1109/SANER.2016.52.

18 Yepang Liu, Chang Xu, and Shing-Chi Cheung. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *2013 IEEE international conference on pervasive Computing and Communications (PerCom)*, pages 2–10. IEEE, 2013.

19 Marc Miltenberger, Julien Gerding, Jens Guthmann, and Steven Arzt. Dfarm: massive-scaling dynamic android app analysis on real hardware. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pages 12–15, 2020.

20 Danilo Dominguez Perez and Wei Le. Generating predicate callback summaries for the android framework. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 68–78. IEEE, 2017.

21 Danilo Dominguez Perez and Wei Le. Specifying callback control flow of mobile apps using finite automata. *IEEE Transactions on Software Engineering*, 47(2):379–392, 2021. doi:10.1109/TSE.2019.2893207.

22 Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS*, 2016.

23 Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 300–311. IEEE, 2017.

24 Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 264–280, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/353171.353189.

25 Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.

26 Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *The 2014 ACM international conference on Measurement and modeling of computer systems*, pages 221–233, 2014.

27 Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. Static window transition graphs for android. *Automated Software Engineering*, 25(4):833–873, 2018.

28 Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 89–99. IEEE, 2015.

29 Wei Yang, Mukul R Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 250–265. Springer, 2013.

30 Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054, 2013.

31 Weilei Zhang and Barbara G Ryder. Automatic construction of accurate application call graph with library call abstraction for java. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(4):231–252, 2007.