Permission-use Provenance in Android Using Sparse Dynamic Analysis

Anonymous

ABSTRACT

As Android permissions grant apps unconstrained resource access, auditing an app's true security implications requires determining the contexts in which each sensitive resource may be used. Unfortunately, as the Android system is complex and callback oriented, determining the events that may trigger any important line of code is difficult. On the one hand, a static analysis would require a complex and ever-changing system model to both precisely reason across these callback boundaries and scale to large apps; on the other, a dynamic analysis would be unable to record all relevant information at runtime without becoming unresponsive.

In this paper, we introduce a novel combination of dynamic analysis, symbolic execution, and abstract interpretation that infers provenance information from a *sparse dynamic analysis*. Starting with a set of sparse execution traces, derived from instrumented app executions, our technique produces inter-callback *provenance diagrams* showing the conditions that held both at a target point of interest, and all preceeding callback registrations that lead to it. We instantiate our technique in a tool called Hogarth and apply it to a set of validation apps, auditing their sensitive resource use, to show that its results match those of a manual reverse-engineering effort. We then further apply Hogarth to a set of real-world case-study apps to demonstrate that the technique is effective at scale. For example, when building a provenance diagram from runtime logs, Hogarth took 0.05 seconds to model camera access in Slack and 1.69 seconds to model location access in Bumble.

KEYWORDS

dynamic analysis, abstraction, Android permissions, provenance, triggers, privacy, Dalvik analysis, security auditing

ACM Reference Format:
Anonymous. 2017. Permission-use Provenance in Android Using Sparse Dynamic Analysis. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 12 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Android apps can request permission to access a wide range of sensitive device resources such as contacts, calendar, and GPS location. However, if we wish to determine whether an app is secure, inspecting its permissions is insufficient, because permissions give apps unconstrained access to their corresponding resources. Understanding an app's security implications requires we audit the *context* of sensitive resource uses [28, 29, 38, 39], e.g., whether a use occurred just after a related button click [26].

Existing program analysis techniques are insufficient for discovering the context of permission uses. The key challenge is that Android is large, complex, and uses callbacks extensively, making even basic control-flow hard to infer [20, 27, 44]. Indeed, existing program analyses for Android focus primarily on finding taint flows [1, 4, 12, 24, 45] and common malware behaviors [8, 13, 14, 40, 42]. (Section 6 discusses related work in more detail.)

This paper introduces Hogarth[1], a new analysis tool for Android apps that, given a permission use in an app, infers its context using a novel combination of symbolic execution, dynamic analysis, and the principles of abstract interpretation. In theory, we could just use symbolic execution. We could start from the beginning of the app and symbolically execute forward until we reached the permission use. The callbacks along all paths, plus the path conditions, would then describe the context. However, in our experience, symbolic execution cannot scale to the full Android framework, and, although there is some work on the topic [25], it is generally hard to force symbolic execution to reach particular program points.

The key idea behind Hogarth is instead to begin with a *sparse dynamic analysis* in which we instrument app bytecode to log method calls and conditional branches taken at runtime. We then select a permission use observed in the log and perform symbolic execution, but *only* along the paths to that permission use observed in the dynamic run. During symbolic execution, we replace concrete values returned from the Android framework with symbolic variables. This novel design gives us the best of both worlds: the symbolic executor, guided by actual execution data, can quickly reach the permission use of interest. At the same time, the dynamic trace—which originally just described one run—is generalized as a path condition on framework values. Moreover, Hogarth achieves all this without needing a detailed model of each framework method.

For further scalability, Hogarth performs symbolic execution on a per-callback basis. Hogarth begins with the callback containing the permission use, and then works backward to its caller (which is identified using information in the logs), and so on until reaching the start of the app. Additionally, Hogarth uses ideas from abstract interpretation to soundly elide information from the path condition that is generally uninteresting, such as array and loop index information.

The ultimate output of Hogarth is what we call a *provenance diagram* explaining the context of a permission use. In our provenance diagrams, nodes are permission uses or app callback methods (such as button click handlers), and there is an edge from node A to node B labeled with path condition ϕ if A may call B when ϕ

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]In the movie *The Iron Giant*, the character Hogarth helps a robot come out of hiding. Our tool is designed to reveal potentially hidden Android permission uses.

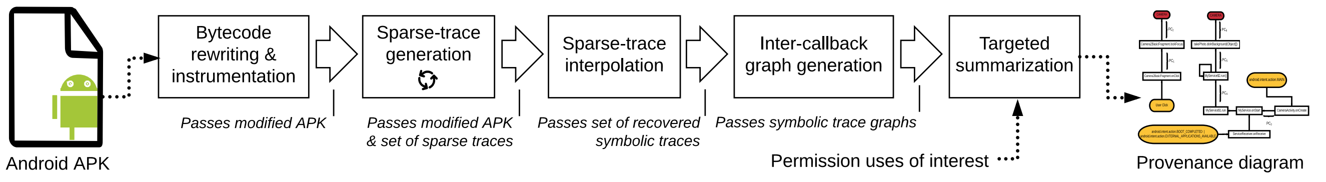

Android APK
Bytecode rewriting & instrumentation
Passes modified APK
Sparse-trace generation
Passes modified APK & set of sparse traces
Sparse-trace interpolation
Passes set of recovered symbolic traces
Inter-callback graph generation
Passes symbolic trace graphs
Targeted summarization
Permission uses of interest
Provenance diagram

Figure 1: Conceptual overview of Hogarth's process.

holds. Thus, provenance diagrams enable app auditors to discover the sequence of events and conditions under which a permission is used. We note that while Hogarth is targeted toward permission use provenance, we speculate that it could be applied to other kinds of security auditing problems or to explaining the cause of bugs or bad UI interactions. (Section 2 gives an overview of our approach, and Section 3 describes provenance inference formally.)

Hogarth's runtime instrumentation is implemented using Redexer [21], a Dalvik bytecode rewriting tool. To validate the Hogarth prototype, we applied it to a set of five apps, selected from F-Droid [23] and the Contagio Malware dump [30]. We found that Hogarth discovers provenance diagrams that match information one of the authors produced manually with a time-consuming reverse engineering effort. We then performed a case study on a selection of top apps from the Google Play store, such as Slack and Bumble, and show that our approach extends to real apps found in the wild. For example, Hogarth builds a model of camera accesses in the Slack [19] app in 0.05 seconds and a model of location accesses in the Bumble [22] app in 1.69 seconds. (Sections 4 and 5 discuss our implementation and evaluation.)

In summary, Hogarth introduces a new approach to automatically infer the provenance of permission uses in a target app's bytecode, using a novel combination of dynamic analysis, symbolic execution, and abstract interpretation. The approach demonstrated in Hogarth has potential applications for debugging, reverse engineering, and security evaluation more generally.

2 OVERVIEW

Figure 1 gives an overview of Hogarth's architecture. Its input is an Android APK file, which contains the app's Dalvik bytecode. Its first step is then to instrument the app so that, when run, it produces a *trace* of the app's execution. The trace is sparse in that it only logs key program points needed to reproduce and generalize the observed execution post hoc.

The user runs the modified app to produce a corpus of representative traces, and then Hogarth performs a process of *sparse-trace interpolation* to infer a *path condition* for every program point in logged callbacks. A path condition is a formula over inputs to the callback (both its arguments and data it receives from the Android framework) that holds whenever that program point is reached.

Next, Hogarth connects distinct handlers into a inter-callback graph, adding an edge from callback A to callback B if A registered B. In this case, we say A is the *registrar*. Finally, Hogarth performs *targeted summarization* which coalesces all runs of a callback that lead to a permission use of interest. This process yields a *provenance diagram*, which is a directed graph where nodes are either callback methods or API calls that need permissions. An edge from A to B

```
class MyService$2 extends Thread { ...
  public void run() { ...
    while (true) { ...
      v14 = v12.readLine(); ...
      if (v14.contains("takephoto(")) {
        if (((String)v15.get(0)).equalsIgnoreCase("front(")) {
          ...
          new takePhoto(myService, FRONT).execute(new String[0]);
        } else
          new takePhoto(myService, BACK).execute(new String[0]);
        } } } }

public class takePhoto extends AsyncTask { ...
  private void initialiseCamera() { ...
    if (PackageManager.PERMISSION_GRANTED ==
        ContextCompat.checkSelfPermission(myService,
          android.Manifest.permission.CAMERA)) { ...
      cameraManager.openCamera(camId,mStateCallback,cameraHandler);
      cameraDevice.createCaptureSession(outputs,
      mccsStateCallback, cameraHandler);
  } ... } ...
  protected Object doInBackground(Object[] arg2) {
    return this.doInBackground(((String[])arg2));
  }
  protected String doInBackground(String[] arg5) {
    initialiseCamera();
    return "Executed";
  } }
```

Figure 2: Code excerpt of the malicious camera use.

labeled with path condition ϕ indicates that callback A may register callback B, or directly use permission B, under that condition (ϕ) on its inputs. Thus, we can use a provenance diagram to understand the circumstances under which a permission is used, both in terms of inter-callback control flow and conditions on app data. In our implementation, these final three conceptual steps have been fused so that interpolation is done in a target-directed way (see Section 4 for details).

Running Example. In the remainder of this section, we illustrate Hogarth on Camera2Evil, an app we produced by injecting the Dendroid [32] malware into the Camera2Basic Android example app [16], which allows the user to take a picture by pressing a button. (The source of Dendroid was extracted using the JEB decompiler [35].) Once Dendroid is injected into the app, a remote command-and-control server is contacted on a timer and the app may secretly take a picture, without the user pressing a button, and upload it.

Figure 2 gives an excerpt from the Camera2Evil code. Here `MyService$2.run`, running in a background thread, listens for (adversary) commands. If it receives the right command, it creates a new instance of `takePhoto`, which is an `AsyncTask`, and executes it (callback registrar shown in blue). This queues up the `takePhoto.doInBackground(Object[])` method to be called by Android in the future.

```
Method > 977 com..MyService$2.run(com..MyService$2@244505)
API > 977 java.io.BufferedReader.readLine(java.io.BufferedReader@84370)
API < 977 java.io.BufferedReader.readLine(java.lang.String@765419)
...
API > 977 java.lang.String.contains(java.lang.String@765419,
  java.lang.String@923754)
API < 977 java.lang.String.contains(java.lang.Boolean@490195)
BBEntry 977 1119442
...
API > 977 android.os.AsyncTask.execute(com..takePhoto@6720952,
  [Ljava.lang.String;@263773)
```

```
Method > 985 com..takePhoto.doInBackground(com..takePhoto
  @6720952,[Ljava.lang.String;@263773)
...
Method > 985 com..takePhoto.initialiseCamera(
  com..takePhoto@6720952)
...
API > 985 java.lang.Object.checkSelfPermission(
  com..MyService@1131473,java.lang.String@7204364)
API < 985 java.lang.Object.checkSelfPermission(
  java.lang.Integer@3382498)
BBEntry 985 1127344
...
API > 985 android..CameraManager.openCamera(
  android..CameraManager@1134037,java.lang.String@9683765,
  com..takePhoto$1@2286550,android.os.Handler@2206143)
...
API > 985 android..CameraDevice.createCaptureSession(
  android..CameraDeviceImpl@122346893,java..ArrayList@122060324,
  com..takePhoto$3@367746,android.os.Handler@2206143)
```

Figure 3: Two partial sparse traces for Camera2Evil.

Once invoked, this method calls `initialiseCamera`, which checks to see that the camera permission has been acquired and, if so, it surreptitiously takes a photo (the sensitive call is shown in red).

Bytecode Instrumentation and Trace Generation. The first step of Hogarth's process is to add logging instrumentation to the Camera2Evil bytecode using Redexer [21], a Dalvik bytecode rewriting tool. More specifically, we insert a new method `log(...)` that writes its arguments and the current thread id to a `ConcurrentLinkedQueue`. We also insert code that creates a background thread to dequeue log entries, introspect on these data, and asynchronously write them to a file. This design helps ensure logging does not slow down the main app thread.

Then, we add calls to `log(...)` to record key events. More specifically, we insert code to record the method arguments (including the receiver) at every app method's entry. For example, we add logging to the beginning of `doInBackground(String[])` that records the `takePhoto` receiver and the `String[]` argument. We also log the arguments and return values to every API call, e.g., the `execute` calls in our running example. Finally, we insert a call to `log(...)` at the entry of every basic block. Hogarth uses this information later in its process to recover exact control flow.

Figure 3 shows two portions of the trace generated by running the instrumented Camera2Evil app. The trace at the top is for `MyService$2.run`, which is an app method as indicated by the `Method >` line in the log. Hogarth infers that this is, in fact, a callback invoked by the Android framework because this trace entry is not nested inside any other method call entry. That same trace line lists the *thread id*, in this case 977, so that we can distinguish otherwise intertwined log entries from different threads. It also includes the arguments to the `run` method, in this case only the receiver object, which is recorded as the object's class and method (here abbreviated) followed by the *object id*, in this case 244505 (every object has a unique integer *id* in the Java runtime). Notice that we do not record object fields—we only use these values to match up callback registrations to the actual callbacks, as discussed below.

The next lines in the top portion of the trace record the call to (`API > 977`) and return (`API < 977`) from `readLine`. Redexer marks a call `API` whenever it cannot find the target method in the app's class definitions. During sparse trace interpolation, API-call results are represented abstractly so that path conditions may be in terms of API calls (details below).

Then after a call/return pair to `contains` (on line 5 in the code), the line `BBEntry 977 1119442` records the entry to basic block number 1119442 (a number assigned by Hogarth). In this case, reaching that basic block tells us that the true branch of the first conditional (the result of testing `contains`) was taken. We will use this information below to build up a path condition. After some additional entries, this portion of the trace ends with a call to `execute`.

The bottom portion of Figure 3 shows a portion of the log from the subsequent call to `takePhoto.doInBackground`. Note that this may occur at an arbitrary point, further on in the log. The log entries are similar to the above, with some API calls, a basic block entry that records taking the true branch of the condition in `initialiseCamera`, and calls to two methods needing the camera permission.

Sparse Trace Interpolation. Next, Hogarth performs *sparse trace interpolation* by replaying the logs using a hybrid of abstract interpretation [9, 10] and symbolic execution [5, 6] to derive a *path condition* at every point, within app methods, reached while the trace was generated. Critically, Hogarth can do so even though the logged information is *sparse*, in particular it only includes method entries and exits, API calls, and the start of each basic block.

Then, for each callback observed in the trace (in our example, `MyService$2.run` and `takePhoto.doInBackground`), Hogarth steps through the method's instructions (and the instructions of any app method called), following the path seen in the trace and building up a path condition in terms of the callback method's inputs and the return values from API calls and field accesses (the latter not shown in this example).

For example, Hogarth will start stepping through `MyService$2.run`. The receiver `this` will be bound to an *abstract value* that records the actual value seen in the log, in this case `com..MyService$2@244505`. As Hogarth continues, it builds more complex abstract values to encode primitive operations and represent returns from framework calls, as necessary. For example, Hogarth will eventually reach the call to `readLine`. For simplicity, suppose that local variable `v12` is bound to an abstract value of the same name (v12). Then after the call, `v14` will simply be bound to the abstract symbolic value BufferedReader.readLine(v12), where the receiver is listed as the first argument.

Whenever there is a branch, Hogarth follows the path taken in the trace and records which way the branch test went in the path condition. For example, recall that the trace took the true branch when testing the return value of `contains`. Thus, Hogarth will take the same path, stepping into the `then` branch of the conditional, and it will conjoin a symbolic value

String.contains(BufferedReader.readLine(v12), ..@923754)

to the path condition. As it continues execution, Hogarth will eventually reach the `execute` call on line 8 and determine that the following path condition, which we will refer to as ϕ_4, holds:

isConnected(getActiveNetworkInfo(getSystemService(
    getApplicationContext(var_{83}),"connectivity"))) $\land$
getActiveNetworkInfo(getSystemService(getApplicationContext(var_{83}),
    "connectivity")) $\land$
readLine(new_{16}) $\land$
find(matcher(compile("\\(([^)]+)\\)"),readLine(new_{16}))) $\land$
contains(readLine(new_{16}),"takephoto(") $\land$
equals(group(matcher(compile("\\(([^)]+)\\)"),readLine(new_{16})),1),"")) $\land$
equalsIgnoreCase(get(new_7,0),"front(")

Similarly, when Hogarth executes `takePhoto.doInBackground`, it will eventually reach the call to `createCaptureSession` with the path condition ϕ_5:

(getCameraIdList(getSystemService(getApplicationContext(var_{61}),
    "camera")).length > 1) $\land$
checkSelfPermission(var_{61},"android.permission.CAMERA")

(We address these path conditions in more detail below.)

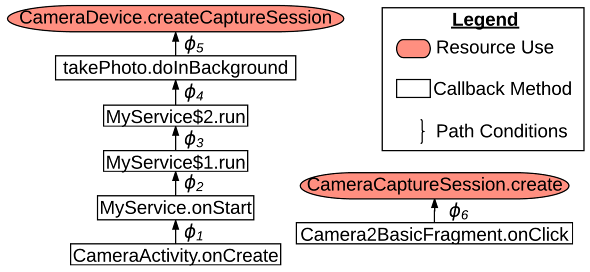
Inter-callback Graphs and Summarization. Hogarth's final steps are to connect distinct callback traces and summarize them to construct the provenance diagram from coalesced callbacks and their inferred path conditions.

Figure 4 shows a portion of the provenance diagram for Camera2Evil. First, Hogarth may assume that all permission-related API calls observed in the trace are of interest. Hogarth adds a node to the graph for each such method, in this case we have `createCaptureSession` and `capture`. We will begin by focusing on the `createCaptureSession`, which is the malicious camera use case. That method was called while `doInBackground` was executed, so Hogarth also adds `doInBackground` to the graph and adds an edge between the nodes labeled with the path condition, ϕ_5, that held when the call occurred. If there were multiple such path conditions (e.g., because `createCaptureSession` was called inside `doInBackground` multiple times) Hogarth labels the edge with the disjunction of the path conditions.

Next, Hogarth determines which callbacks were triggered by which other callbacks. For example, recall that in Camera2Evil, the `run` method scheduled `doInBackground`. Hogarth finds these dependencies using a heuristic. We use the EdgeMiner [7] data set to identify which API calls might register callbacks. Then if the trace contains a call to a registration method that passes some object which is later used as the receiver object in a callback, Hogarth assumes the registration caused the callback and adds an appropriate edge to the graph.

For example, EdgeMiner reports that `AsyncTask.execute` is such a registration method. The trace in Figure 3 for `MyService$2.run` contains a call to `execute` (in blue) with object @6720952 as an argument. Subsequently, the `doInBackground` method is invoked with @670952 as the receiver. Thus, Hogarth adds an edge to the provenance diagram from `MyService$2.run` to `doInBackground`, labeled with the path condition, ϕ_4, that held when `execute` was called.

In addition to using EdgeMiner to determine callback connections, there are some cases where this is not possible (e.g. Intent passing) and we have to manually establish these connections. We discuss this further in Section 4.



ϕ_3	((equals("goldfish",$static_{19}$) == 0) $\land$ (startsWith($static_{18}$,"generic") == 0) $\land$ (startsWith($static_{17}$,"generic") == 0) $\land$ (equals("google_sdk",$static_{12}$) == 0) $\land$ (equals("google_sdk",$static_{13}$) == 0) $\land$ (!(booleanValue(var_6) == 0)) $\land$ (getBoolean(getDefaultSharedPreferences(getApplicationContext(var_2)), "Start",0) == 0))
ϕ_6	(getId(arg_3) == #x7f0a0059)

Figure 4: Camera2Evil: Partial camera-use provenance diagram. (ϕ_1 and ϕ_2 are true; ϕ_4 and ϕ_5 are shown inline.)

Hogarth continues this process, adding callbacks to the graph until it reaches a fixpoint. We may then use the resulting graph to investigate under what circumstances permissions are used. If we trace back the diagram in Figure 4, we see that when the app launches (`onCreate`), it starts the malicious service (`MyService.onStart`) which kicks off another thread (`MyService$1.run`). Both ϕ_1 and ϕ_2 are shown as `true` because these transitions through the program will always occur after the app is started. In `MyService$1.run`, ϕ_3 tells us that the malware checks its `DefaultSharedPreferences` for two settings, whether it should start and if its Google Play bypass feature is enabled. If the bypass feature is enabled, then it checks the build information on the device to determine whether the app is running on an emulator, i.e. the product or model are "`google_sdk`", the brand or device are "`generic`" or the hardware is "`goldfish`". If ϕ_3 holds, then a second thread `MyService$2` is started to connect to the adversary's command-and-control server. ϕ_4 states that if the device has an Internet connection (either wifi or cellular), it receives some commands from the server, and the string "`take photo(front(`" is included in the commands, then it begins a new asynchronous task to take a photo and send that back to the server (`takePhoto.doInBackground`). Finally, ϕ_5 states that if there is at least one camera available on the device and the app has been granted the permission "`android.permission.CAMERA`", then the malware calls `CameraDevice.createCaptureSession` to surreptitiously take a picture.

Similarly, Hogarth generates the provenance diagram for the instance of `createCaptureSession` on the right, the benign case. In the benign case, when `onClick` is triggered by a user click, ϕ_6 states that if ID of the button that was clicked is equal to `#x7f0a0059` (the ID of the "take picture" button), then `CameraCaptureSession.capture` is called to take a picture. Using these provenance diagrams auditors can better understand the conditions of a resource use and differentiate between the malicious and benign cases.

3 PROVENANCE INFERENCE

This section gives a formal presentation of the analysis just described. To keep our presentation compact, we show how to infer

$$
\begin{array}{rcl}
prog & ::= & \overrightarrow{class} \\
class & ::= & C <: C\ \overrightarrow{method}\ \vec{f} \\
method & ::= & m(\vec{r})\ \vec{i} \\
i & ::= & \texttt{goto}\ j \mid \texttt{if}\ r\ \texttt{then}\ j \mid r \leftarrow c \mid r \leftarrow r \mid r \leftarrow r \oplus r \\
 & \mid & r \leftarrow r.f \mid r.f \leftarrow r \mid r \leftarrow \texttt{new}\ C \mid r \leftarrow r.m(r, \ldots) \\
 & \mid & \texttt{ret}\ r \\
c & ::= & n \mid str \mid true \mid false \\
\oplus & ::= & \{+, -, *, <, \neg, \wedge, \vee, \ldots\}
\end{array}
$$

$$
\begin{array}{ccccccc}
C & \in & classes & & m & \in & methods \\
f & \in & fields & & r & \in & regs \\
n & \in & integers & & str & \in & strings
\end{array}
$$

Figure 5: Simplified Dalvik bytecode.

provenance using the simplified Dalvik bytecode language in Figure 5. Here and below, we write $\vec{x}$ for a sequence of zero or more x's, and we write x_i to signify the ith element of such a sequence (starting from index 0).

In this language, a program *prog* consists of a sequence of class definitions $\overrightarrow{class}$. A single class definition consists of a class name C, its superclass, a sequence of method definitions $\overrightarrow{method}$, and a sequence of field names $\vec{f}$. Each method definition includes the method name m, a sequence of registers $\vec{r}$ for the formal parameters, and a sequence of instructions $\vec{i}$ for the method body.

Instructions are fairly standard. An unconditional jump `goto` j sets the program counter so the instruction at index j is executed next. A conditional jump `if` r `then` j branches to instruction j if the contents of register r is true. Assignments of the form $r \leftarrow c$ write a constant integer, string, or boolean into register r; assignments $r_1 \leftarrow r_2$ copy r_2 to r_1; and assignments $r_1 \leftarrow r_2 \oplus r_3$ apply some operation $\oplus$ to r_2 and r_3, storing the result in r_1. Fields are read with $r_1 \leftarrow r_2.f$ and written with $r_1.f \leftarrow r_2$. Allocation $r \leftarrow$ `new` C creates a fresh instance of class C and stores a pointer to it in r. Method invocation $r \leftarrow r_0.m(r_1, \ldots)$ performs dynamic dispatch of method m with the given receiver r_0 and arguments $r_1, \ldots$, assigning the result to r. Lastly, `ret` r exits the current method, returning r.

Note that this language omits many features of Dalvik bytecode, such as arrays, static methods and fields, etc. We discuss details of handling full Android apps in Section 4.

3.1 Sparse Trace Generation

As previously discussed, provenance inference begins by instrumenting and executing the program to gather a set of dynamic traces. In our implementation (Section 4), we modify the app's bytecode to add tracing instrumentation. Here we elide that step, and simply describe the trace this instrumentation yields.

Figure 6 gives a grammar for *program traces pt*, which consists of a sequence of *callback traces* $\vec{ct}$. Each callback trace records what happens from the time Android invokes an app callback to the time the callback returns to the framework. A callback trace $C.m(\vec{v})\ \vec{ti}$ records the class C and method m called by the framework, along with the argument values $\vec{v}$, where v_0 encodes the method receiver. Each value is either ignored, written ϵ, or a class C paired with an *id*. In our implementation, we log all constants as ϵ for performance reasons (specifically, writing all strings to the trace

$$
\begin{array}{rcll}
pt & ::= & \vec{ct} & \text{[program trace]} \\
ct & ::= & C.m(\vec{v})\ \vec{ti} & \text{[callback trace]} \\
ti & ::= & C.m & \text{[app call]} \\
 & \mid & C.m(\vec{v}) & \text{[API call]} \\
 & \mid & \texttt{then} \mid \texttt{else} & \text{[branch]} \\
v & ::= & \epsilon \mid C@id & \text{[value]}
\end{array}
$$

Figure 6: Dynamic traces.

is expensive). Notice that we do not record object fields—we only use these values to match up callback registrations to the actual callbacks, as discussed below.

Each callback trace also includes a sequence of *trace items* $\vec{ti}$ that occurred during the callback. There are four kinds of trace items. First, $C.m$ logs a call to an app method m of class C. We elide arguments because these will be recovered via symbolic execution. Second, $C.m(\vec{v})$ logs a call to an API method m of class C. In this case, we do record the argument values $\vec{v}$ since they may include possible callback registrations. Lastly, `then` and `else` log which way each `if` instruction branched. These model the `BBEntry` trace entires in Section 2.

3.2 Sparse Trace Interpolation

Next, Hogarth uses a hybrid abstract interpretation and symbolic execution to simulate the app bytecode, in order to infer path conditions describing the paths seen in the trace. Figure 7 formalizes this process as a series of operational rules over machine *states* $\langle \vec{i}, rf, \kappa, \phi, \vec{ti} \rangle$. Here $\vec{i}$ is the sequence of instructions remaining to be executed. The register file *rf* maps registers to *abstract values a*, which include constants, object *id*s paired with classes, and operations among abstract values, which are standard. Abstract values also include r, which stands for the value read from a register (here, always a parameter to the method); $a.f$, which stands the value read from a field of an object; and $C.m(\vec{a})$, which stands for the value returned from an API call with arguments $\vec{a}$ (where, again, the first argument is the receiver object). These last three forms allow Hogarth to track the conditions on "inputs" to the callback from parameters or from values returned by the Android framework.

Each machine state also includes a stack κ—a (possibly-empty) sequence of triples $(r, rf, \vec{i})$, where r is the register to be written upon returning, rf is the previous register file to reinstate, and $\vec{i}$ is the sequence of instructions to resume upon returning. The path condition ϕ is a (boolean) abstract value, and $\vec{ti}$ is the trace to be followed. Finally, the trace suffix $\vec{ti}$ is a sequence of remaining trace items in the log to be processed.

Figure 7b lists the machine's operational rules. [Jump] replaces the instruction sequence with those at the target address, using helper function *instr*(j) (not formalized). [Then] and [Else] handle conditional branches, using the observed trace to guide the machine. When the head of the trace is `then`, the concrete execution took the true branch, so [Then] conjoins the branch condition $rf(r)$ to the current path condition ϕ (meaning the abstract value at $rf(r)$ must correspond to true) and jumps. When the head of the trace is `else`, execution fell through, so [Else] simply steps past the branch

$$\begin{array}{rcll}
\mathit{state} & ::= & \langle \vec{i},\ rf,\ \kappa,\ \phi,\ \vec{ti} \rangle & \text{[machine state]} \\
rf & : & \mathit{regs} \rightharpoonup a & \text{[register file]} \\
\kappa & ::= & \epsilon \mid (r,\ rf,\ \vec{i}) : \kappa & \text{[stack]} \\
\phi & ::= & a & \text{[path condition]} \\
a & ::= & c \mid C@id \mid \oplus\vec{a} & \text{[abstract value]} \\
 & \mid & r \mid a.f \mid C.m(\vec{a}) &
\end{array}$$

(a) Abstract machine domains.

$\langle \texttt{goto}\ j : \vec{i},\ rf,\ \kappa,\ \phi,\ \vec{ti} \rangle \leadsto \langle \mathit{instr}(j),\ rf,\ \kappa,\ \phi,\ \vec{ti} \rangle$	[Jump]
$\langle \texttt{if}\ r\ \texttt{then}\ j : \vec{i},\ rf,\ \kappa,\ \phi,\ \texttt{then} : \vec{ti} \rangle \leadsto \langle \mathit{instr}(j),\ rf,\ \kappa,\ rf(r) \wedge \phi,\ \vec{ti} \rangle$	[Then]
$\langle \texttt{if}\ r\ \texttt{then}\ j : \vec{i},\ rf,\ \kappa,\ \phi,\ \texttt{else} : \vec{ti} \rangle \leadsto \langle \vec{i},\ rf,\ \kappa,\ \neg rf(r) \wedge \phi,\ \vec{ti} \rangle$	[Else]
$\langle r \leftarrow c : \vec{i},\ rf,\ \kappa,\ \phi,\ \vec{ti} \rangle \leadsto \langle \vec{i},\ rf[r \mapsto c],\ \kappa,\ \phi,\ \vec{ti} \rangle$	[AssnC]
$\langle r \leftarrow r' : \vec{i},\ rf,\ \kappa,\ \phi,\ \vec{ti} \rangle \leadsto \langle \vec{i},\ rf[r \mapsto rf(r')],\ \kappa,\ \phi,\ \vec{ti} \rangle$	[AssnR]
$\langle r_1 \leftarrow r_2 \oplus r_3 : \vec{i},\ rf,\ \kappa,\ \phi,\ \vec{ti} \rangle \leadsto \langle \vec{i},\ rf[r_1 \mapsto \oplus(rf(r_2), rf(r_3))],\ \kappa,\ \phi,\ \vec{ti} \rangle$	[AssnOp]
$\langle r' \leftarrow r.f : \vec{i},\ rf,\ \kappa,\ \phi,\ \vec{ti} \rangle \leadsto \langle \vec{i},\ rf',\ \kappa,\ \phi,\ \vec{ti} \rangle$ where $rf' = rf[r' \mapsto a.f] \wedge a = rf(r)$	[AssnF]
$\langle r.f \leftarrow r' : \vec{i},\ rf,\ \kappa,\ \phi,\ \vec{ti} \rangle \leadsto \langle \vec{i},\ rf,\ \kappa,\ \phi,\ \vec{ti} \rangle$	[FWrite]
$\langle r \leftarrow \texttt{new}\ C : \vec{i},\ rf,\ \kappa,\ \phi,\ \vec{ti} \rangle \leadsto \langle \vec{i},\ rf[r \mapsto C@id],\ \kappa,\ \phi,\ \vec{ti} \rangle$ where id fresh	[New]
$\langle r'' \leftarrow r.m(\vec{r}) : \vec{i},\ rf,\ \kappa,\ \phi,\ C.m : \vec{ti}' \rangle \leadsto \langle \vec{i}',\ rf',\ \kappa',\ \phi,\ \vec{ti}' \rangle$ where $rf' = [\vec{r}' \mapsto rf(\vec{r})] \wedge m(\vec{r}')\ \vec{i}' = \mathit{lookup}(C.m) \wedge \kappa' = (r'',\ rf,\ \vec{i}) : \kappa$	[Call]
$\langle \texttt{ret}\ r : \vec{i},\ rf,\ (r',\ rf',\ \vec{i}') : \kappa,\ \phi,\ \vec{ti} \rangle \leadsto \langle \vec{i}',\ rf'[r' \mapsto rf(r)],\ \kappa,\ \phi,\ \vec{ti} \rangle$	[Ret]
$\langle r' \leftarrow r.m(\vec{r}) : \vec{i},\ rf,\ \kappa,\ \phi,\ C.m(\vec{v}) : \vec{ti} \rangle \leadsto \langle \vec{i},\ rf[r' \mapsto C.m(rf(\vec{r}))],\ \kappa,\ \phi,\ \vec{ti} \rangle$	[API]

(b) Abstract machine semantics.

Figure 7: Formalism for sparse trace interpolation.

instruction and conjoins the negation of the branch condition with the path condition.

[AssnC], [AssnR], and [AssnOp] update the register file so the left-hand side register r maps to the given constant, register contents, or operation, respectively. Fields are symbols derived from an underlying object; [AssnF] looks up the value stored in a register and produces a new symbol for its field. [FWrite] simply consumes the write instruction—each time we read from this field we reproduce a consistent symbolic name derived from the object's symbolic name and the field accessed. [New] handles an allocation, in which a fresh *id* (meaning one not chosen before and not in the trace) is paired with C and bound in the register file.

[Call] handles an invocation of one of the app's methods. At these control points, the log has recorded $C.m$, the class and method that was invoked at run time. Thus, the rules use *lookup* (not formalized) to retrive the corresponding method definition. [Call] then pushes the return register, the current register file, and the remaining instruction sequence onto the stack. It then begins executing the callee's instructions under a new register file mapping the formal parameters to the actual arguments. [Ret] handles a return by popping the stack frame and updating the return register. Finally, [API] binds an abstract value representing the call.

To derive path conditions for a program trace $pt = ct_0, ct_1, \ldots$, for each $ct_j = C.m(\vec{v})\ \vec{ti}$, we begin with an entry-point state:

$$\langle \mathit{lookup}(C, m), rf[r_i \mapsto a(r_i, v_i)], \epsilon, \mathit{true}, \vec{ti} \rangle$$

We begin executing the body of $C.m$ with a register file where each formal parameter r_i is bound to $a(r_i, v_i)$, where $a(r_i, C@id) = C@id$ and $a[(r_i, \epsilon) = r_i$. The initial stack is empty, the initial path condition is *true*, and the initial sequence of trace items comes from the callback trace. We write $ct_j \leadsto^* \mathit{state}$ if *state* is reachable in zero or more steps of the machine starting in the initial (entry-point) state for ct_j.

3.3 Inter-callback Graphs and Summarization

Finally, we can build the provenance diagram. We begin by first constructing an *exploded graph* as follows. Let $pt = ct_0, ct_1, \ldots$ be the program trace. Initially, we add to the graph every API method $C.m$ that requires a permission of interest. Then, until we reach a fixpoint, we pick a node $C.m$ in the graph and add an edge $C'.m' \xrightarrow{\phi} C.m$ for every ϕ, either: (1) such that we have $C'.m'(\vec{v}')\ \vec{ti} \in pt$ and

$$C'.m'(\vec{v}')\ \vec{ti}' \leadsto^* \langle \_, \_, \_, \phi, C.m(\vec{v}) : \vec{ti} \rangle$$

for some C, m, $\vec{v}$, and $\vec{ti}$, or, (2) such that $C.m(\vec{v})\ \vec{ti} \in pt$ and there exists, in any recovered trace, a state

$$\langle \_, \_, \_, \phi, C'.m'(\vec{v}') : \vec{ti}' \rangle, \text{where } v_0 = v_i', \text{for some } i$$

This first case handles adding edges from callbacks ($C'.m'$) to API calls that preceed target permissions. At the first iteration of this process, all these API calls will be the target permissions themselves. The second case handles connecting callbacks ($C.m$) back to the API calls that may register them[2]. This is done by making an edge back to registrar calls that pass the receiver object (v_0) as some argument (v_i'). After this, case (1) will connect these registrars back to their callbacks and the process continues to work backward. In total, this fixpoint computation builds a directed graph of callbacks and API calls that lead to a target sensitive resource use.

The exploded graph may have multiple edges, with different path conditions, between the same pair of nodes. To create the final *summarized graph* we combine these edges. For every connected pair of nodes $C'.m'$ and $C.m$ we consider all their path conditions

$$C'.m' \xrightarrow{\phi_0} C.m \qquad C'.m' \xrightarrow{\phi_1} C.m \qquad \ldots$$

and replace them with a single edge

$$C'.m' \xrightarrow{\phi_0 \vee \phi_1 \vee \ldots} C.m$$

Our final provenance diagrams are simply these summarized graphs where a callback and registrar form a node.

[2]calls that could be a callback registration according to the EdgeMiner database

4 IMPLEMENTATION

We implemented Hogarth using Redexer [21] and some functionality taken from SymDroid [20]. Our code was written in Java (for the logging machinery) and OCaml (for additions to Redexer and for the core interpolation and summarization phase), and required on the order of 10,000 lines of code. There were several unique challenges in implementing our approach for full Android apps.

Logging instrumentation. We observed that, in practice, it is crucial to ensure Hogarth's instrumentation does not affect app performance too much, especially in the UI thread, since Android kills applications whose UI thread becomes non-responsive. Thus, our inserted `log` calls perform no introspection themselves. Instead, they add messages to a `ConcurrentLinkedQueue`, which uses a wait-free algorithm to communicate with a separate worker thread that retrieves messages from the queue to produce the trace output. In total, the message passing interface adds between 10 and 20 Dalvik instructions for each inserted `log` call.

In our formalism, API calls do not themselves carry an internal trace. However, that is insufficient in practice. For example, consider `java.util.Collections.sort`, which we treat as an API method. When called, it in turn will call the app's `compare` method for objects passed in, which will have logging calls inside it. To handle this case, our implementation maintains a *phantom context* representing the unknown portion of the stack inside the API call. This allows us to soundly process or skip calls back into user code.

Sparse trace interpolation. Recall that the abstract values in Figure 7a have a fairly rich structure. Hogarth encodes registers, field accesses, and method calls as symbolic variables with special names. For example, an API call value $C.m(a)$ where C is a *BufferedReader*, m is a *readLine*, and a is a new C', may be encoded as a symbolic variable named BufferedReader.readLine(new$_{56}$) where 56 refers to a particular allocation site.

One issue arises given this encoding: in the presence of loops and recursion, we might wind up reusing a symbolic variable name. This could cause the same symbolic variable to stand for multiple values, which might then yield multiple, possibly contradictory branch conditions. To sidestep this issue, we observe that path condition clauses relevant for permission uses typically do not involve variables that change with loop iterations. Thus, if Hogarth is in a loop and is about to reuse a symbolic variable already in the path condition, it heuristically removes all clauses involving that variable before reusing it, strongly updating its meaning in the path condition. In practice this means path conditions will only include information about the last iteration of a loop, which in our experience is the most useful behavior.

As we developed Hogarth, we found that path conditions often contain many abstract values for arrays. In practice those values were uninteresting and their presence in path condition decreases their utility significantly. Thus, our implementation includes a special abstract value $\top$ that represents any possible abstract value, and we model all arrays as $\top$. Constraints on $\top$ are discarded and not added to the path condition, and abstract values derived from $\top$ are widened to $\top$ (e.g., $\top.f$ evaluates to $\top$).

One last issue in sparse trace interpolation involves invocations of `<clinit>` methods, which are invoked whenever the Dalvik Virtual Machine decides to load a class. Thus, there is no syntactic call site for such calls, and therefore they do not really fit well in a provenance diagram. We opted to simply elide such calls from our analysis. To do so correctly, we add extra logging to the end of `<clinit>` methods so we know when they exit (whereas we do not log the return of other app methods).

Inter-callback connections. Recall that we use the EdgeMiner [7] database to identify possible registrar methods, and then we connect up a registrar with a callback if the receiver of the latter was an argument to the former, using the object *id* for comparison.

While this approach is largely successful, there are a few cases where Android reuses the same object for different callbacks, particularly `Intents` and `Threads`; thus we cannot rely on their object *id*s. We address this issue by using a different *id* in these cases. For `Intents` (which are essentially key-value maps), we add a *magic id* field that gets a fresh value each time, and use that in place of the object *id*. For `Threads`, we use the thread *id* in place of the object *id*.

Demand-driven interpolation and summarization. Although Hogarth was originally implemented with five distinct passes as described in Section 2, we discovered in scaling the process to our case study that the vast majority of callback invocations are not actually upstream from the target permission use. This means a significant amount of work was being done to interpolate and coalesce invocations not relevant to the provenance diagram at hand. For this reason, we performed a pass-fusion optimization, turning the final three passes (interpolation, inter-callback graph generation, and targeted summarization) into a single demand-driven interpolation and summarization phase.

In this new phase, all invocations in the log containing the target permission use are first interpolated and coalesced into a single summarized callback. Then, from this initial singleton frontier, the algorithm makes proximate inter-callback connections, interpolates each, summarizes syntactically like callbacks into a new frontier of summarized callbacks, and then continues in a demand-driven manner until all entry points are reached and all upstream invocations have been exhausted. Because this pass fusion allows us to elide the majority of logged invocations at each phase, it was crucial to scaling Hogarth and supporting larger apps.

5 EVALUATION

We performed two studies using Hogarth, a validation study and a case study. We used the validation study to a) check that Hogarth worked correctly and b) evaluate the impact of its approximations on the end result. We obtained ground truth by using five moderately-sized apps that were manually reversed engineered by the third author, who has professional experience reverse engineering Android apps. While reverse engineering these apps, we generated a provenance diagram in the style of Figure 4. Then, we ran Hogarth on each app to ensure that it produced the same graph of handlers and correctly inferred the path conditions leading to each registration and permission use.

Once we were convinced that Hogarth was producing correct results, we performed a case study on five larger apps that appeared in Micinski et. al's paper on AppTracer [26]. AppTracer used temporal sequences of events to understand permission usage in apps and classify them as either foreground or background. We demonstrate how Hogarth can be used to precisely understand provenance for these permissions.

5.1 Validation Study

For our validation study, we selected five apps that use permissions in the background. The first app is our running example, the Camera2Evil app. Three others are benign apps from the open-source F-Droid [23] repository. We include an additional malicious app from the Contagio Malware dump [30].

To obtain ground truth, we had a reverse engineering expert manually identify the permission uses of interest within the app's source code. The expert then examined all of the app's code to trace backwards and determine how each method could be reached via handler registrations. Throughout this process, the expert collected path conditions he believed were relevant to the use of the permission. These results were synthesized into a provenance diagram for each app.

Next, we ran Hogarth on each app to generate a provenance diagram. We compared the results of Hogarth with our manually-generated provenance diagram. Using this procedure, we confirmed that Hogarth correctly inferred the path conditions for the logs we used. However, Hogarth both a) occasionally misses edges not identified in EdgeMiner due to an incomplete model of the Android API, and b) generates more verbose path conditions than our expert. Below, we walk through each example app and discuss how Hogarth works on each.

Camera2Evil. The first app in our validation study was our running example, Camera2Evil. As discussed in previous sections, Camera2Evil has two triggers leading to the use of the `CAMERA` permission. One of them is the benign behavior, as a result of clicking a button, and the other was the malicious trigger, which is invoked when the app receives a command from the control server.

Hogarth generated a diagram identical in structure to the diagram produced by manual reverse engineering. However, Hogarth generated path conditions which included additional clauses that our expert did not identify as relevant to the permission use. These redundant clauses are those which either a) mentioned the configuration of member variables or b) loop postconditions.

Hogarth successfully discovered the path leading to the malicious permission use, which happens when a line received from the server contains the string "`takephoto(front(`." Camera2Evil makes a call to `BufferedReader.readLine` until it finds a string matching the specific name. As we discussed in Section 4, Hogarth uses a strategy to allocate symbolic names in the presence of loops. This helped ensure that Hogarth correctly included the last iteration of the loop in the summary path condition.

Call Recorder. The Call Recorder app [3] allows users to record calls and store a copy on their device. The app registers an Android broadcast receiver[18], which will receive messages from the Android system whenever a phone call takes place and start recording accordingly. Hogarth infers the path leading to the `RECORD_AUDIO` permission, rooted at the `MyPhoneReceiver.onReceive` handler within the app. Hogarth correctly infers an edge from `onReceive` to `RecordService.onStartCommand`, the background service that performs the recording, and correctly infers all path conditions for this edge: that a user-controlled recording flag has been set, that external storage is mounted, and that the phone is off-hook. Hogarth also infers several checks for null variables our expert excluded.

Misbothering. Misbothering SMS [41] is an app that mutes notification for any message sent by a user not in the contacts list. We used Hogarth on Misbothering and were correctly able to discover the circumstances under which it accesses the user's contacts. Upon receiving an SMS, an listener within the app runs, which executes a set of checks to ensure that particular app variables are initialized correctly and checks whether the sender of the SMS was in the user's contacts. The handler also checks that a non-empty message was received. The Hogarth diagram matches the diagram generated by our expert for this handler.

Contact Merger. The Contact Merger app [37] helps identify duplicate contacts by analyzing a user's contact list and recommending entries that may refer to the same person. Hogarth assembled a correct provenance diagram for each logged scenario. For example, when the app starts up, the `READ_CONTACTS` permission is used after the app checks that the an internal contacts database has been set up (the source code indicates this is a workaround for a bug), that a specific file (used to store the contacts) exists, and that the contact analysis has not yet been performed. Several behaviors were not covered by our logs, which did not include a restart or a new package installation, or run the app for at last an hour. Incomplete logs, of course, are a limitation of dynamic analysis.

Smart Studio Proxy. Smart Studio Proxy (also known as Android Trojan Spy [36]) is a piece of malware, discovered in 2015, designed to collect a variety of user data and ship it to the attacker. The malware accesses the user's SMS messages whenever a new message arrives, any message is changed by the user, the phone wakes from sleep mode, the network connectivity changes, or 30 minutes has passed since the last upload to the attacker (because there was no available network connection previously). Hogarth correctly identifies each behavior covered in the logs, but fails to correctly assemble the root of a provenance diagram in one case. This was due to an incomplete model of the intent system in Android, not covered by EdgeMiner.

5.2 Case Study

Next, we wanted to study how Hogarth could be applied to analyze permission usage in production apps. For this portion, we chose to study five apps from the set included in [**?**]. We ran each app, exercising the behavior of a few permissions. We then looked at the logs from each app and used Hogarth to synthesize a provenance diagram for each permission use.

Slack. We studied how Slack was using the camera permission. This was marked as a foreground use in [26]. Our log of the Slack app included one use of the camera permission. Using Hogarth, we discovered that this was as the result of clicking a button within a file picker to take a button. We observed that Slack first checked to ensure that it had permission to write to the user's external storage, and allowed us to discover that after taking the picture, it would be stored on the user's device. Slack uses the user's default camera app to take the picture. Because of this, it checks that the user has a default camera app installed. Last, it builds up a timestamp for the image and embeds it a path which will be used to store the picture after it is taken. The path condition also includes several checks to ensure this timestamp and the path is of the right format.

Bumble. We studied the use of the device ID and location permissions in Bumble. Both of these were marked as background uses in [26]. We found that Bumble used location in a callback `handleMessage`, which receives and processes messages sent to it. We found that Bumble used a continuous background service which would check the user's location and send it back to the server. Before sending the user's location, the app checks a variety of conditions. For example, the app appears to use the Google awareness API to check that the user has recently been moving, and checks the configuration of the wifi and network state. The app also allows the user to use a fake location, set within the app. If this fake location is set, no location update is sent.

Bumble used device ID in the same handler. While forming the packet to send a location update to the server, the app accessed the user's subscriber ID (the IMSI associated with the user's SIM card), which it included in the update packet sent to the server.

Samsung Cloud Print. We investigated the use of the camera permission in the Samsung Cloud Print app. Hogarth allowed us to verify that the camera was being used in response to the start of a page in the app that takes a picture of a QR code. This allowed us to deduce that the permission was used throughout the time when that page was on the screen. Before opening the camera, the app checked a variety of fields to ensure they were non-null.

Ovia Pregnancy Tracker. Ovia used the user's location in the background. Using Hogarth we were able to tell that the app set up a thread to run continuously. We were able to trace this thread's provenance back to point on which the app becomes active again.

Doctor on Demand. We used Hogarth to understand how Doctor on Demand used the location permission. We were able to see that location was used as the result of clicking on a button to schedule an appointment. The app polls the user's last known location so that it can begin to start a new page to schedule an appointment at a clinic nearby the user. Before doing so, it checks to ensure the user is not exiting the app.

5.3 Performance and Scalability

Table 1 shows Hogarth's performance on our set of validation and case study apps. The app name is shown in the left column, followed by the permission. Several apps we evalauted had multiple places within the app where the permission was used. We include a line in the table for each distinct target (lexical point in the program in which the permission was used). The second set of columns measures both the number of joins performed between handler abstractions, and the total number of steps taken by the semantics. The *Mem* column lists the maximum heap size used by Hogarth. Finally, the last set of columns describes's Hogarth's running time, broken up in phases: *Parse* for log parsing, *Summ* for targeted summarization, and *Mini* for minimization, followed by the total running time for that target. The total running time also includes various other operations (such as loading the EdgeMiner database).

We observed that the running time of the summarization phase was roughly linear in the number of steps taken. Memory usage of Hogarth was closely related to the size of the parsed log, and dominates the memory consumption compared to the subsequent summarization phase. Minimization time depends on the size of formulas sent to Z3.

6 RELATED WORK

There are several threads of related work.

Contextual Security Analysis for Android. Several researchers have proposed program analyses that aim to infer the context, or provenance, of security-relevant actions in Android apps. Pegasus [8] analyzes apps to infer Permission Event Graphs (PEGs), which describe the relationship between Android events and permission uses. In contrast to Hogarth's provenance diagrams, PEGs do not include predicates about the app state.

DroidSIFT [48] builds data-dependency graphs using a context-sensitive, flow-sensitive, interprocedural dataflow analysis to identify triggering conditions (i.e., user interaction or system event) for sensitive resource use. AppIntent [45] also uses data-flow analysis to identify program paths that may leak private information, and then employs directed symbolic execution on those paths to find inputs that could trigger a leak. As the full Android system is too complicated to effectively apply symbolic execution in a scalable manner, the authors use a system model to assist the analysis. AppContext [43] identifies inter-procedure control-flow paths from program entry points to potentially malicious behaviors and then performs a data-flow analysis to identify the conditions that may trigger malicious behaviors. AppContext is similar to our approach in that it identifies the crucial branch conditions and inter-procedural contexts that lead to sensitive behaviors. TriggerScope [14] uses a combination of static analysis and symbolic execution to identify particularly complex trigger conditions associated with potentially malicious code. They attempt to detect "logic bombs" by identifying path conditions that are abnormally complex when simplified. Hogarth has three key differences with this work. First, our approach depends on gathering a representative corpus of dynamic traces, relying on the conceit that all permission uses will be exercised.Second, because we rely on dynamic traces to drive further analysis of the application, we achieve a more precise result because all events observed in our dynamic traces are possible. Third, because we rely on a minimal model of the system, our approach is more resilient to the changes in Android from version to version.

FuzzDroid [31] uses a genetic mutation fuzzer to drive execution of an app toward a specific target location. IntelliDroid [40] uses static analysis to identify an overapproximation of inputs that could trigger malicious activity and then dynamically executes these inputs to prune false positives. These approaches identify a single path that reaches a target, whereas Hogarth identifies the set of conditions that could lead to sensitive resource use.

App	Permission	# Lines	Joins	Steps	Mem (GB)	Parse (s)	Summ (s)	Mini (s)	Total (s)
Misbothering	SMS / Con	689	90	20,099	1.215	3.57	0.054	.053	3.80
Call Recorder	Mic	15,429	52	16,079	1.215	3.76	0.083	0.14	5.31
Camera2Evil	Cam (1)	30,403	96	110,533	1.606	4.21	0.49	4.29	12.07
Camera2Evil	Cam (1)	30,403	56	36,289	1.397	4.22	0.14	3.19	9.67
Smart Studio Proxy	SMS (1)	67,780	234	903,521	1.397	6.42	5.24	6.14	23.08
Smart Studio Proxy	SMS (2)	67,780	308	174,019	1.205	4.92	0.56	.041	6.20
Contact Merger	Con	4.20M	144	955,455	3.717	81.74	7.87	0.44	152.29
Bumble	Loc / State	8.18 M	328	434,113	4.274	41.63	8.36	26.34	116.64
Slack	Cam	7.67 M	2	13,597	4.915	45.33	0.18	120.38	182.39
Ovia Pregnancy	Loc	3.82 M	8	22,069	2.810	37.62	0.12	23.80	66.46
Samsung Cloud Print	Cam	4.21 M	4	12,419	3.232	67.63	0.058	35.634	108.130
Doctor on Demand	Loc	4.89 M	54	31,277	2.810	27.39	0.21	5.90	38.48

Table 1: Performance measurements for our validation and case study.

Lastly, AppTracer [26] uses dynamic analysis to discover what user interactions temporally precede sensitive resource uses. Hogarth builds on AppTracer's tracing infrastructure, which also uses Redexer, but adds inter-callback graph generation, to discover dependencies among callbacks, and symbolic execution/abstract interpretation, to recover path conditions. As a result, Hogarth can infer much richer contextual information about sensitive resource uses than AppTracer.

Taint and Flow Analysis for Android. TaintDroid [12] modifies the Android firmware to perform system-wide dynamic taint-tracking and notifies the user whenever sensitive data is leaked. Phosphor [4] provides similar taint-tracking, but instead modifies the JVM to improve portability. FlowDroid [1] uses static data-flow analysis find sensitive data leaks. These tools all focus on data flow, which is orthogonal to the control-flow (e.g., path conditions) dependencies that Hogarth discovers.

Other Analyses for Android. Yang et al. [42] present a model for tracking callback sequences in Android called *Callback Control-Flow Graphs* (CCFG). By connecting callbacks to their sources through the framework, CCFGs allow a static analysis to traverse context-sensitive control flow paths and identify callbacks that could be triggered. Because it is possible for different callbacks to be triggered based on the invocation context of the handler, e.g. `onClick` may trigger different callbacks depending on the widget it is associated with. This context-sensitivity improves the precision of static analysis in Android.

Building on the concept of CCFGs, EdgeMiner [7] performs a static analysis that automatically creates API summaries describing the relationship between callbacks and registrations through the framework via static data-flow analysis. As mentioned earlier, we use EdgeMiner's list of API methods that could register callbacks, but refine it by actual observations of data flows.

User-centric dependence analysis [11] uses a dependence analysis of Android apps to characterize the data consumption behaviors along paths from user inputs to sensitive resource uses. This project is a preliminary effort at identifying which user inputs a sensitive API call depends upon.

Concolic Execution. One style of symbolic execution is *concolic execution* [6, 15, 33], in which programs are instrumented to track symbolic expressions at run-time along with their concrete counterparts in the actual run. A benefit of concolic execution is that when a system call is made, any symbolic expressions passed in can be *concretized*, i.e., made equal to, their underlying values. This lets concolic executors avoid needing to model system calls, though at the expense of less power (e.g., not all possible paths that system call could take will be modeled). In a sense, Hogarth's approach is dual to concolic execution: We start with a concrete trace, and we then turn system call returns into *abstract* values so we can track their effect on program execution.

Text-based Contextual Security. Most closely related to our work, Zhang et al. [47] utilize DroidSIFT [48], to generate dependency graphs and use common Natural Language Generation techniques to create human-readable descriptions of apps' sensitive resource uses. Because our analysis produces a more precise result, we are able to provide more detailed conditional information and target our approach to app auditors who are likely to better understand the system, as opposed to end users.

Additionally, several researchers have explored the use natural-language analysis of app artifacts such as UI elements and privacy policies to identify contextual security issues for Android. For example, BACKSTAGE [2] mines Android apps for pairs of UI elements and the API calls they trigger and performs a clustering analysis to find outliers that invoke APIs atypical for their UI elements and textual descriptions. AsDroid [17] performs static analysis of top-most app methods and textual analysis of UI components they are associated with, to detect semantic mismatches. Slavin, et al. [34] produce a map of API methods to privacy policy phrases in order to check that natural-language privacy policies match the API uses in Android apps. TAPVerifier [46] builds a data-flow model of the target app and a natural-language model of its privacy policy to detect a mismatch. Wijesekera, et al. [39] and Olejnik, et al. [29] consider context outside of the app such as whether the app was in the foreground or background, whether the user was home or in public.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[2] Vitalii Avdiienko, Konstantin Kuznetsov, Isabelle Rommelfanger, Andreas Rau, Alessandra Gorla, and Andreas Zeller. 2017. Detecting Behavior Anomalies in Graphical User Interfaces. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C ’17)*. IEEE Press, Piscataway, NJ, USA, 201–203. https://doi.org/10.1109/ICSE-C.2017.130

[3] Axet. 2015. Misbothering SMS Receiver. (2015). https://f-droid.org/packages/com.github.axet.callrecorder/ (Accessed 4-11-2017).

[4] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA ’14)*. ACM, New York, NY, USA, 83–101. https://doi.org/10.1145/2660193.2660212

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI’08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://portal.acm.org/citation.cfm?id=1855756

[6] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS ’06)*. ACM, New York, NY, USA, 322–335. https://doi.org/10.1145/1180405.1180445

[7] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS ’15)*. San Diego, California, USA.

[8] Kevin Zhijie Chen, Noah M Johnson, Vijay D’Silva, Shuaifu Dai, Kyle MacNamara, Thomas R Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. 2013. Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS ’13)*. Internet Society, San Diego, CA, 234.

[9] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM Press, New York, Los Angeles, CA, 238–252.

[10] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM Press, New York, San Antonio, TX, 269–282.

[11] Karim O Elish, Danfeng Yao, and Barbara G Ryder. 2012. User-centric dependence analysis for identifying malicious mobile apps. In *Proceedings of the 1st Workshop on Mobile Security Technologies (MoST ’12)*. IEEE Press, San Jose, CA.

[12] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI’10)*. USENIX Association, Berkeley, CA, USA, 393–407. http://dl.acm.org/citation.cfm?id=1924943.1924971

[13] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based Detection of Android Malware Through Static Analysis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 576–587. https://doi.org/10.1145/2635868.2635869

[14] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. 2016. TriggerScope: Towards Detecting Logic Bombs in Android Applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P ’16)*. IEEE Press, San Jose, CA, 377–396. https://doi.org/10.1109/SP.2016.30

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[16] Google, inc. 2017. Camera2Basic Android Sample App. (2017). https://github.com/googlesamples/android-Camera2Basic

[17] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1036–1046. https://doi.org/10.1145/2568225.2568301

[18] Google Inc. and the Open Handset Alliance. 2017. Android documentation: Broadcasts. (2017). https://developer.android.com/guide/components/broadcasts.html (Accessed 5-17-2017).

[19] Slack Technologies Inc. 2017. Slack App. https://play.google.com/store/apps/details?id=com.Slack. (2017).

[20] Jinseong Jeon, Kristopher K Micinski, and Jeffrey S Foster. 2012. Symbolic Execution for Dalvik Bytecode. (2012). (Tech Report, CS-TR-5022).

[21] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. 2012. Dr. Android and Mr. Hide: fine-grained permissions in android applications. In *Proceedings of the 2nd ACM workshop on Security and privacy in smartphones and mobile devices (SPSM ’12)*. ACM, ACM, Raleigh, NC, 3–14.

[22] Bumble Holdings Limited. 2017. Bumble App. https://play.google.com/store/apps/details?id=com.bumble.app. (2017).

[23] F-Droid Limited. 2017. F-Droid - Free and Open Source Android Repository. (2017). https://f-droid.org/ (Accessed 4-11-2017).

[24] Kangjie Lu, Zhichun Li, Vasileios P. Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. 2015. Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Data-flow Analysis and Peer Voting. In *Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS ’15)*. Internet Society, San Diego, CA. http://www.internetsociety.org/doc/checking-more-and-alerting-less-detecting-privacy-leakages-enhanced-data\discretionary{-}{}{}flow-analysis-and-peer

[25] Kin-Keung Ma, Khoo Yit Phang, Jeffrey Foster, and Michael Hicks. 2011. Directed symbolic execution. *Static Analysis* (2011), 95–111.

[26] Kristopher Micinski, Daniel Votipka, Rock Stevens, Nikolaos Kofinas, Michelle L. Mazurek, and Jeffrey S. Foster. 2017. User Interactions and Permission Use on Android. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI ’17)*. ACM, Denver, Colorado, USA, 362–373. https://doi.org/10.1145/3025453.3025706

[27] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. 2012. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[28] Helen Nissenbaum. 2004. Privacy as Contextual Integrity. *Washington Law Review* 79 (2004), 119–157.

[29] Katarzyna Olejnik, Italo Dacosta, Joana Soares Machado, Kévin Huguenin, Mohammad Emtiyaz Khan, and Jean-Pierre Hubaux. 2017. SmarPer: Context-Aware and Automatic Runtime-Permissions for Mobile Devices. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (IEEE S&P ’17)*. IEEE, San Jose, CA, United States. https://hal.archives-ouvertes.fr/hal-01489684

[30] Mila Parkour. 2017. Contagio Mobile. (2017). http://contagiominidump.blogspot.com/ (Accessed 4-11-2017).

[31] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments. In *Proceedings of the 39th International Conference on Software Engineering (ICSE ’17)*. ACM, Buenos Aires, Argentina.

[32] Marc Rogers. 2014. Dendroid malware can take over your camera, record audio, and sneak into Google Play. (2014). https://blog.lookout.com/blog/2014/03/06/dendroid/ (Accessed 4-11-2017).

[33] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272. https://doi.org/10.1145/1081706.1081750

[34] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. 2016. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE ’16)*. ACM, ACM, Austin, TX, 25–36.

[35] PNF Software. 2017. JEB Decompiler. (2017). www.pnfsoftware.com (Accessed 5-19-2017).

[36] Lukas Stefanko. 2015. Android Trojan Spy Goes 2 Year Undetected. (2015). http://b0n1.blogspot.com/2015/04/android-trojan-spy-goes-2-years.html?spref=tw (Accessed 4-11-2017).

[37] Rene Treffer. 2014. Contact Merger. (2014). https://f-droid.org/repository/browse/?fdfilter=contacts&fdid=de.measite.contactmerger (Accessed 4-11-2017).

[38] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. 2015. Android Permissions Remystified: A Field Study on Contextual Integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security ’15)*. USENIX Association, Washington, D.C., 499–514. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wijesekera

[39] Primal Wijesekera, Arjun Baokar, Lynn Tsai, Joel Reardon, Serge Egelman, David Wagner, and Konstantin Beznosov. 2017. The Feasibility of Dynamically Granted Permissions: Aligning Mobile Privacy with User Preferences. *CoRR* abs/1703.02090 (2017). http://arxiv.org/abs/1703.02090

[40] Michelle Y Wong and David Lie. 2016. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS ’16)*. Internet Society, San Diego, CA.

[41] Amir Yalon. 2015. Misbothering SMS Receiver. (2015). https://f-droid.org/repository/browse/?fdfilter=Misbothering+SMS+Receiver&fdid=net.yxejamir.misbotheringsms (Accessed 8-25-2017).

[42] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static Control-flow Analysis of User-driven Callbacks in Android Applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 89–99. http://dl.acm.org/citation.cfm?id=2818754.2818768

[43] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. 2015. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In *Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE '15)*, Vol. 1. ACM, Florence, Italy, 303–313. https://doi.org/10.1109/ICSE.2015.50

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

[45] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X. Sean Wang. 2013. AppIntent: analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security (CCS '13)*. ACM, New York, NY, USA, 1043–1054. https://doi.org/10.1145/2508859.2516676

[46] Le Yu, Xiapu Luo, Chenxiong Qian, and Shuai Wang. 2016. Revisiting the description-to-behavior fidelity in android applications. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, IEEE Press, Osaka, Japan, 415–426.

[47] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. 2015. Towards Automatic Generation of Security-Centric Descriptions for Android Apps. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 518–529. https://doi.org/10.1145/2810103.2813669

[48] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 1105–1116. https://doi.org/10.1145/2660267.2660359