Symbolic Path Tracing to Find Android Permission-Use Triggers

Anonymous

***Abstract*—Understanding whether Android apps are safe requires, among other things, knowing what dynamically triggers an app to use its permissions, and under what conditions. For example, an app might access contacts after a button click, but only if a certain setting is enabled. Automatically inferring such conditional trigger information is difficult because Android is callback-oriented and reasoning about conditions requires analysis of program paths. To address these issues, we introduce Hogarth, an Android app analysis tool that constructs *trigger diagrams*, which show, post hoc, what sequence of callbacks, under what conditions, led to a permission use observed at run time. Hogarth works by instrumenting apps to produce a trace of relevant events. Then, given a trace, it performs *symbolic path tracing*—symbolic execution restricted to path segments from that trace—to infer path conditions at key program locations, and *path splicing* to combine the per-segment information into a trigger diagram. We validated Hogarth by showing its results match those of a manual reverse-engineering effort on five small apps. Then, in a case study, we applied Hogarth to 12 top apps from Google Play. We found that Hogarth provided more precise information about triggers than prior related work, and was able successfully generate a trigger diagram for all but one permission use in our case study. Hogarth's performance was generally good, taking at most a few minutes on most of our subject apps. In sum, Hogarth provides a new approach to discovering conditional trigger information for permission uses on Android.**

***Index Terms*—Symbolic execution, Android, symbolic path tracing, triggers**

I. Introduction

Android apps must request *permissions* before accessing sensitive resources such as the user's microphone or location. However, on their own, permissions are insufficient for auditing an app's actual security implications [1]–[4]. This is because the presence of a permission alone does not convey *why* the permission will be used.

In this paper, we propose to shed light on why apps use permissions by introducing the concept of *trigger diagrams*, which describe paths through an app that lead to a permission use. For example, Figure 1 shows trigger diagram explaining a case in which the *Ovia Pregnancy* app accesses the user's location. The nodes of the graph are either *callbacks* invoked by the Android framework or *permission uses* in the app. An edge from one node to another indicates control flow, and edges are labeled with *path conditions*: logical formulas that must hold when the control flow occurs. For example, this diagram shows a case in which, when the app goes off screen (i.e., when onStop is called by Android), it starts a new thread whose run method uses location. Examining the path conditions reveals, among other things, that the app checks that it has permission to access location before using it along this path (we provide a detailed overview in Section II). Thus, trigger diagrams can help elucidate the circumstances of a permission use, which could potentially be useful for tasks such as auditing app behavior and reverse engineering.

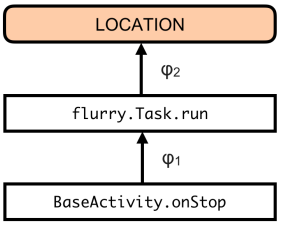

Fig. 1: Trigger diagram for the *Ovia Pregnancy* app.

In theory, trigger diagrams could be constructed using static analysis, but this is impractical. The key challenge is that apps are tightly coupled with the Android framework, i.e., significant control and data flow occur through the framework. Yet the framework is not amenable to precise static analysis, because it is large, complex, and includes native code. Thus, we cannot statically analyze an app in tandem with the framework. To alleviate this problem, some researchers have explored developing Android framework models [5]–[12], but Android has tens of thousands of methods, and, to date, there is no modeling approach that scales to all of Android. As a result, while other researchers have explored a range of problems related to trigger diagrams [6], [9], [10], [12]–[16], no prior work has demonstrated a scalable, precise technique that could be used to construct them.

To remedy this situation, in this paper, we introduce Hogarth,[1] a novel Android app analysis tool for inferring trigger diagrams. The key insight behind Hogarth is to use *dynamic analysis*, rather than static analysis. More specifically, Hogarth works by instrumenting apps to produce a *trace* of relevant events. The user runs the app to generate a set of execution traces, and then Hogarth generates a trigger diagrams for selected permission uses within those traces.

Hogarth uses two core technical ideas to construct trigger diagrams from traces. First, it observes precisely what objects are registered as callbacks, and what callbacks are invoked by Android. For example, in Figure 1, this allows Hogarth to determine that there is control flow from onStop, which registers flurry.Task, to the latter's run method. Because Hogarth observes registrations and callbacks at runtime, it does

[1]Named after one of the main characters in the movie *The Iron Giant*. Just because.

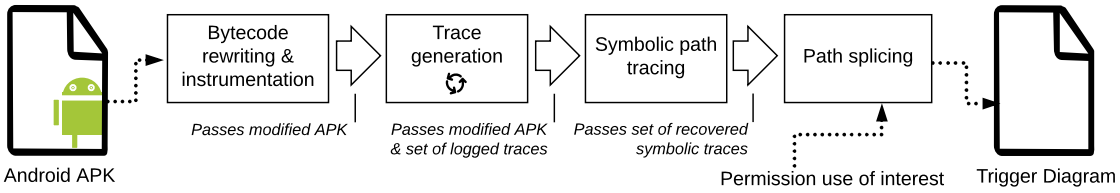

Fig. 2: Hogarth's architecture.

not need a precise model of Android—it only needs to know which methods could possibly register callbacks.

Second, to infer path conditions, Hogarth uses what we call *symbolic path tracing* and *path splicing*. Given the target permission use, Hogarth performs symbolic execution from the start of the callback containing that permission use. Critically, however, Hogarth *only* follows the path observed in the trace. Again, this obviates the need for a precise framework model, because, via the trace, Hogarth has an exact, realizable path that reaches the permission use of interest. During symbolic execution, Hogarth replaces concrete values returned from the Android framework with symbolic variables whose names describe where the value came from, either framework calls or fields. The result is a path condition that describes that path segment in terms of those symbolic variables. For example, this allows us to see the location permission check along the path in Figure 1. Hogarth repeats this process, working backward from handler to handler until it reaches a root, which is the trigger. In effect, this novel design allows Hogarth to turn a concrete execution trace into a more general description of what happened in the app along that path. Moreover, Hogarth achieves all this without needing app source code or a detailed framework model. (Section III describes trigger diagram inference formally.)

Hogarth is implemented on top of Redexer [17], a Dalvik bytecode rewriting tool, and SymDroid [18], a Dalvik bytecode symbolic executor we modified extensively. To achieve sufficient tracing performance, Hogarth uses a fast parallel queue to buffer runtime traces and write them to a file with a separate thread. Hogarth elides some information, specifically about arrays, from path conditions to make them more readable. Finally, Hogarth invokes symbolic path tracing in a demand-driven fashion to reduce the amount of code that must be symbolically executed. (Section IV describes Hogarth's implementation in more detail.)

To validate the Hogarth prototype, we applied it to a set of five apps selected from F-Droid [19] and the Contagio Malware dump [20]. We found that Hogarth discovers trigger diagrams that match information one of the authors produced manually with a time-consuming reverse engineering effort.

We also performed a case study in which we applied Hogarth to 12 top apps from Google Play, chosen from a dataset from a prior related paper [16]. We found that Hogarth successfully identified triggers that could not be resolved by the prior work, which used a simpler temporal heuristic to find triggers [16]. Hogarth also allowed us to effectively understand why apps were using permissions. For example, we were able to determine that one app (*Doctor on Demand*) checked to see whether the user had selected to visit a nearby doctor before polling for location data, and that another (*Samsung Cloud Print*) compared the user's country code to ensure they were in the US or South Korea before accessing their phone number. Finally, we found that Hogarth generally performs well, e.g., Hogarth can build a model of the location permission in *Grubhub* in 46 seconds, given a 4.4 million line log. (Section V discusses our evaluation.)

In summary, Hogarth introduces a new approach to construct trigger diagrams showing the cause and the conditions leading to a target permission use. While Hogarth focuses on Android and permission uses, we believe our approach can be used for debugging, reverse engineering, and other auditing purposes more generally.

II. Overview

Figure 2 shows Hogarth's architecture. Its input is an Android APK file, which contains the app's Dalvik bytecode. Hogarth's first step is to instrument the app so that, when run, it produces a *trace* of the app's execution. For performance reasons, the trace is sparse in that it only logs key program points needed to reproduce the observed execution.

The user runs the modified app to produce one or more traces and selects a permission use of interest (more precisely, an Android API call that requires a permission). Hogarth then performs *symbolic path tracing* to infer path conditions for various program points that were observed in the trace. A path condition is a formula over inputs to the callback (including values it receives from the Android framework) that holds whenever that program point is reached. Finally, Hogarth performs *path splicing* to work backward from the permission use to find its triggers, connecting all such paths together to yield a trigger diagram.

In the remainder of this section, we illustrate how Hogarth can be used to produce the diagram in Figure 1. (Hogarth currently outputs a textual description of the graph; we manually drew the visualization in the figure.)

a) Bytecode Instrumentation and Trace Generation: Hogarth adds logging instrumentation to app bytecode using Redexer [17], a Dalvik bytecode rewriting tool. First, we use Redexer to statically link our logging library into the app. Our logging library contains a new method `log(...)` that writes its arguments and the current thread id to a `ConcurrentLinkedQueue`. Our logging instrumentation creates a background thread to dequeue log entries and asynchronously

```
Mtd > 1 BaseActivity.onStop(SplashActivity@207023171)
...
BBEntry 1 2477570
API > 1 Handler.post(Handler@...,flurry.Task@2..02)
API < 1 Handler.post()
...
Mtd < 1 BaseActivity.onStop()
```

```
Mtd > 181 flurry.Task.run(flurry.Task.run@222621802)
...
BBEntry 181 2471390
API > 181 LocationManager.requestLocationUpdates(...)
API < 181 LocationManager.requestLocationUpdates()
...
Mtd < 181 flurry.Task.run()
```

Fig. 3: Two partial traces for *Ovia*.

write them to a file. This 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 class and identity of the method's arguments (including the receiver object) at every method's entry. For example, we add logging to the beginning of the BaseActivity.onStop handler within *Ovia*. We also log the arguments and return values of every API call. 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 *Ovia* app. The trace at the top is for flurry.Task.run, which is an app method as indicated by the Mtd > line in the log. Because this trace entry is not nested inside any other method call entry, Hogarth infers that this is a callback invoked by the Android framework. That same trace line lists the thread id (in this case 1) so that Hogarth can distinguish otherwise intertwined log entries from different threads. It also includes the method arguments, in this case just the receiver object. Objects are recorded as the object's class followed by the object's Java id (every object has a unique integer id in the Java runtime). Note that, to improve performance, we do not record object fields. Hogarth only uses object ids to identify callbacks (discussed more below).

The line BBEntry 1 2477570 records the entry to basic block numbered 2477570 (a number assigned by Hogarth during app instrumentation). Hogarth uses this information later during symbolic path tracing to decide which way to follow branches.

The next two lines record the call to (API > 1) and return from (API < 1) Handler.post, an Android API method. Redexer marks a call API whenever it cannot find the target method in the app's method definitions. In this case, the call to Handler.post asks Android to create a thread for the flurry.Task object passed as its second argument. Finally, the last line in the top portion of the figure indicates the return from onStop.

The bottom portion of Figure 3 shows a portion of the trace for the subsequent execution of flurry.Task.run. Note that in general, arbitrary other trace elements might occur between the top and bottom portions of the trace, depending on the Android scheduler. (For this particular trace, several other threads do execute between the post and the run.) Once the run method begins, it eventually enters a basic block that calls requestLocationUpdates, which is an Android framework method that requires the location permission.

b) Symbolic Path Tracing: Next, Hogarth performs *symbolic path tracing* by running symbolic execution [21], [22] from the start of each relevant callback, stepping through that method's instructions and those of any called app methods, until it reaches target points in the trace. (We discuss exactly where Hogarth starts and stops symbolic path tracing below.) For each target point, Hogarth records the *path condition* generated by symbolic execution. Whenever Hogarth reaches a branch, it follows the path taken in the trace, which is apparent from the BBEntry trace items.

For example, Hogarth begins symbolic path tracing at onStop. The receiver this is bound to an *abstract value* representing the actual value seen in the log, in this case SplashActivity@207023171. As Hogarth continues, it builds more complex abstract values to encode primitive operations and represent returns from framework calls, as necessary. As Hogarth branches, it conjoins the branch condition on to the path condition, which is initially just *true*. For example, when Hogarth eventually reaches the call to Handler.post in *Ovia*, Hogarth obtains the following path condition ϕ_1:

$$\begin{aligned}&\text{new}_{35} \neq 0 \wedge \text{staticfd}_{113}.\text{c.a.get}(\text{argument}_0.\text{f})).\text{iterator}().\text{hasNext}()\\ &\wedge\ !\text{argument}_1.\text{isChangingConfigurations}() \wedge\ !\text{argument}_0.\text{f.isEmpty}()\\ &\wedge\ \text{staticfd}_{12}.\text{c.a.get}(\text{argument}_0.\text{f}) \neq 0 \wedge \text{new}_{34}.\text{iterator}().\text{hasNext}()\end{aligned}$$

This path condition mentions objects created by the app, e.g., new_{35} is an object created at source location 35. The actual trace includes the object id; we have abbreviated for clarity. The path condition also mentions the state of static fields, e.g., functions on staticfd_{113}, as well as functions on the state of the arguments, e.g., $!\text{argument}_1$.isChanging-Configurations() asserts that the app is not currently changing screen configurations.

Similarly, when Hogarth executes flurry.Task.run, it will eventually reach the call to requestLocationUpdates with the path condition ϕ_2:

$$\begin{aligned}&\text{“passive”} \neq 0 \wedge \text{staticfd}_{113} \neq 0 \wedge \text{new}_{18}.\text{x} \geq \text{new}_{19}.\text{y}\\ &\wedge\ \text{staticfd}_{50}.\text{c.getSvc(“connectivity”).isConnected()}\\ &\wedge\ \text{com.flurry.sdk.fd.class} \neq 0 \wedge\ !\text{staticfd}_{18}.\text{isEmpty()}\\ &\wedge\ \text{staticfd}_{50}.\text{c.getSvc(“connectivity”).getActiveNetworkInfo} \neq 1\\ &...\\ &\wedge\ \text{staticfd}_{50}.\text{c.checkCalling..Permission(“ACCESS\_FINE\_LOCATION”)}\end{aligned}$$

This path condition mentions the state of many global variables, but most relevant to our analysis are the framework-related calls to check the state of the connectivity service and the check that the app has permission to access location.

c) Path Splicing: Finally, Hogarth performs *path splicing*, which connects the symbolic path traces together and summarizes them to construct a trigger diagram. Hogarth invokes symbolic path tracing on-demand during path splicing. This final step begins with a user-specified permission use. In

this case, we choose the call to requestLocationUpdates in Figure 3. Hogarth then begins working backward. Since this call occurred during flurry.Task.run, Hogarth adds a node for that callback to the diagram, runs symbolic path tracing from the start of the callback to the permission use, and then adds an edge between the nodes, labeled with the path condition ϕ_2. If there were multiple such path conditions, Hogarth would label the edge with the disjunction of the path conditions.

Next, Hogarth determines where the callback flurry.Task.run was registered. Hogarth heuristically looks for calls to API methods that register callbacks according to the EdgeMiner [23] dataset. Hogarth assumes that any such call where the argument matches the value bound to this in the callback—here, flurry.Task.run@222621802—was responsible for registering the callback. In our example, this occurred in the call to Handler.post. Thus, Hogarth runs symbolic path tracing from the beginning of the callback containing the registration up to that registration point, adding appropriate nodes and edges to the diagram. In this case, there will be an edge from onStop to run labeled with ϕ_1.

This process continues until Hogarth can find no more matching callback registrations. In our example, Hogarth stops with onStop.

In addition to using EdgeMiner to determine callback connections, there are some cases where this is not possible (e.g., Intent passing), so Hogarth adds extra metadata to certain objects to help establish these connections. We discuss this further in Section IV.

d) Reviewing Trigger Diagrams: Finally, we use the resulting trigger diagram to investigate the circumstances under which permissions are used. An expert can use the diagrams to make a range of decisions or to aid more general reverse engineering tasks. In our experiments (Section V), we systematically considered each node in the trigger diagram and studied the path condition generated by Hogarth. For example, in *Ovia*, the first use registered the thread when the app was not restarting the screen, and used the user's location through an analytics library after ensuring it had permission to do so. In the second permission use we examined, the app accessed the filesystem to store cached data as the result of a network request performed by the app.

III. Trigger Diagram Inference

This section gives a formal presentation of our symbolic path tracing analysis to find triggers. To keep our presentation compact, we describe our approach using the simplified Dalvik bytecode language in Figure 4. 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.

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

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

Fig. 4: Simplified Dalvik bytecode.

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 content 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 IV.

A. Trace Generation

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

Figure 5 gives a grammar for *program traces* pt, which consist 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, written $C@id$. In our implementation, we log all constants as ϵ for performance reasons (specifically, writing all strings to the trace is expensive). Notice that we do not record object fields—we only record object addresses 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

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

Fig. 5: Traces.

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

(a) Abstract machine domains.

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

(b) Abstract machine semantics.

Fig. 6: Formalism for symbolic path tracing.

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 entries in Section II.

B. Symbolic Path Tracing

Figure 6 formalizes *symbolic path tracing*, which symbolically executes a program along the path in a given trace. Hogarth uses symbolic path traces as a subroutine when constructing a trigger diagram.

We describe symbolic path tracing 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 ids 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 method parameter); $a.f$, 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 last two items in the machine state are the path condition ϕ, a (boolean) abstract value, and $\vec{ti}$, the remaining trace to be followed.

Figure 6b 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 and jumps. When the head of the trace is else, execution fell through, so [Else] simply steps past the branch 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.

[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 retrieve 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.

The remaining rules introduce abstract values that represent data from "outside" the current method, i.e., from fields and from the framework. [AssnF] looks up the value stored in a register and produces an abstract value representing its field f. [FWrite] is a simply no-op, since any subsequent read of the field will represent the read symbolically. [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. Finally, [API] binds an abstract value representing the call. Notice this is similar in spirit to reading a field, where rather than try to model a value coming from outside the method, we simply track where it came from.

To build the trigger diagram, described next, Hogarth runs symbolic path tracing on each relevant handler in the trace. Hogarth starts at the beginning of the handler and runs until reaching a particular instruction—either the target, permission-using API call, or an intermediate callback on the way to that API call.

Given a program trace $pt = ct_0, ct_1, \ldots$ and a particular $ct_j = C.m(\vec{v})\ \vec{ti}$, we write $C.m(\vec{v})\ \vec{ti} \rightsquigarrow^* state$ if $state$ is reachable in zero or more steps of the machine starting in the initial (entry-point) state, defined as

$$\langle lookup(C.m), rf[r_i \mapsto a(r_i, v_i)], \epsilon, true, \vec{ti}\rangle$$
$$\text{where } a(r_i, C@id) = C@id$$
$$\text{and } a(r_i, \epsilon) = r_i$$

Here we begin with the instructions of the target method and a register file where each formal parameter r_i is bound to $a(r_i, v_i)$. The initial stack is empty, the initial path condition is $true$, and the initial sequence of trace items comes from the callback trace.

C. Path Splicing

Finally, we can construct the trigger diagram by splicing together the paths that lead to the permission use. For purposes of this discussion, we assume we have run symbolic path tracing on every callback to every possible location. In our actual implementation (discussed next), we do so on demand to improve performance.

We begin by choosing an API call $C.m(\vec{v})$ of interest in the trace—in our application, this is an API call that requires a permission. Then we add an edge $C'.m'(\vec{v'}) \xrightarrow{\phi} C.m(\vec{v})$ to the diagram for the $C'.m'(\vec{v'})\ \vec{ti}' \in pt$ such that

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

for some ϕ. In other words, we add nodes for the callback that contains the target call and for the call itself, and we add an edge between them labeled with the path condition from symbolic path tracing. For example, this corresponds to the edge from flurry.Task.run to location in Figure 1.

Next, until we reach a fixed-point, we pick a node $C.m(\vec{v})$ in the graph and find all $C'.m'(\vec{v'})\ \vec{ti} \in pt$ such that there exists a state

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

and add an edge $C'.m'(\vec{v'}) \xrightarrow{\phi} C.m(\vec{v})$ to the trigger diagram. In other words, we work backward from the target permission use, adding edges from registrars to callbacks until we can add no more such edges. A call is considered a registration if one of its arguments ($\vec{v'_i}$) is the callback receiver ($\vec{v_0}$). In our implementation, we further restrict this step to methods known to be callback registrars.

Finally, we compress the diagram slightly. Notice that, as constructed so far, the diagram may have multiple edges, with different path conditions, between the same pair of nodes. To create the final trigger diagram we combine these edges. For every connected pair of nodes $C'.m'(\vec{v'})$ and $C.m(\vec{v})$ we consider all their path conditions

$$C'.m'(\vec{v'}) \xrightarrow{\phi_0} C.m(\vec{v}) \qquad C'.m'(\vec{v'}) \xrightarrow{\phi_1} C.m(\vec{v}) \qquad \ldots$$

and replace them with a single edge

$$C'.m'(\vec{v'}) \xrightarrow{\phi_0 \vee \phi_1 \vee \ldots} C.m(\vec{v})$$

In our implementation, we perform some further simplifications to make the diagrams easier to read.

IV. Implementation

We implemented Hogarth using Redexer [17] and SymDroid [18], the latter of which we had to extend significantly. Our code was written in Java (for the logging machinery) and OCaml (for additions to Redexer and for symbolic path tracing), and comprises around 10K lines of code. There were several unique challenges in implementing Hogarth.

a) Logging instrumentation: 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 do not perform logging directly. 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, depending on the number of arguments.

In our formalism, API calls always return to app code without any intervening calls to the app. But in practice, this may not hold. For example, within a call to java.util.Collections.sort, the framework will call a compare from the app, which will contain logging calls. Our implementation handles this case by extending the symbolic executor to support a nested stack for the call from the framework to the app.

b) Symbolic Path Tracing: Recall that the abstract values in Figure 6a 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 from this encoding: in the presence of loops and recursion, we might reuse 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 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 are uninteresting, and their presence makes path conditions much harder to read. 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 $\top$).

A final issue in symbolic path tracing involves <clinit> methods, which are invoked whenever the Dalvik Virtual Machine decides to load a class. Because there is no syntactic call site for such calls, they do not fit well within a trigger diagram. We opted to simply elide such calls from our analysis, which is sound because these methods can never register handlers.

c) Inter-callback connections: Recall that we use the EdgeMiner [23] 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 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 ids. We address this 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.

d) Demand-driven path splicing: In practice, traces are quite long—in our experiments, up to around 10 million lines. Thus, it is important that Hogarth not perform symbolic path tracing on the entire trace. Instead, Hogarth is demand-driven. It first divides the trace into segments, one for each top-level callback. To begin path tracing, all paths in the trace containing the target permission use are symbolically traced, and the results are combined and put into the trigger diagram. Then, Hogarth works backward one callback at a time, running symbolic path tracing only on callbacks that registered nodes in the diagram so far. We found this approach achieved dramatic speedups compared to an earlier implementation that timed out on even modestly sized logs.

V. Evaluation

We evaluated Hogarth using two studies. First, we performed a validation study to confirm that Hogarth's output is correct. We ran Hogarth on five moderately sized apps for which the third author had earlier created trigger diagrams manually. We compared the manual results to Hogarth's results and found they were consistent, except Hogarth's path conditions were sometimes more verbose and Hogarth missed some edges because it relies on dynamic traces.

Second, we conducted a case study in which we applied Hogarth to 12 popular free apps from Google Play and constructed trigger diagrams for two permission uses per app. We found that Hogarth allowed us to identify triggers for each permission use we studied, analyzing most apps within minutes. Additionally, we found that the path conditions produced by Hogarth were helpful in identifying why apps were accessing permissions.

A. Validation Study

To test the correctness of Hogarth, we wanted to compare Hogarth's output to ground truth. Thus, we selected five apps that were small enough that they could be manually analyzed by the third author, a reverse engineering expert with professional experience. Our expert decompiled each app with JEB [24], identified each API call corresponding to a permission use, and then manually read through the code line-by-line to construct a trigger diagram. JEB simplified the task of manual reverse engineering by allowing variable renaming and the on-demand display of all potential static calls for a given function (using control-flow analysis). We then ran Hogarth on the app and compared its output to our manual results.

Our first three apps were selected from the F-Droid repository [19]. *Call Recorder* [25] allows users to record calls and store a copy on their device. Hogarth correctly identifies that the microphone is used after recording is enabled. For example, Hogarth finds a path condition that mentions the app's state being in recording mode, and also that the directory in which recordings will be stored exists. *Misbothering SMS* [26] mutes notifications for any message sent by a user not in the contacts list. Using Hogarth, we correctly observed that contacts are accessed by a callback after receiving a text message. Third, *Contact Merger* [27] identifies duplicate contacts by analyzing a user's contact list and recommending entries that may refer to the same person. Hogarth identified the use of the contacts in this app, but failed to observe the use of contacts whenever a new app was installed, because we did not see this behavior in the generated trace. Incomplete logs, of course, are a limitation of dynamic analysis.

The next app was *SmartStudioProxy*, a piece of malware from the Contagio Malware dump [20]. This app collects a variety of user data and ships it to an attacker [28]. Hogarth correctly found the triggers for each permission use in the trace, but failed to find a path that was triggered every 30 minutes, as we did not run the app that long.

The last app was *Camera2Basic*, an Android example app [29] that allows the user to take a picture by pressing a button. We modified *Camera2Basic* by injecting the Dendroid [30] malware, also from the Contagio dump, into it. Dendroid contacts a remote command-and-control server on a timer. The app may also secretly take a picture, without the user pressing a button, and upload it. Hogarth successfully finds both the legitimate and malicious use of camera in the

App Name	Log (Lines)	Size (MB)	1st P.[1] Use	Len	Steps	Tm (s)	2nd P. Use	Len	Steps	Tm (s)
Ovia Pregnancy	1.7M	11.4	Loc	1	46K	26	Sto	1	12K	24
Grubhub	4.4M	11.8	Loc	1	1.8K	46	Sto	2	98K	66
Flipp	7.9M	7.6	Loc	1	96K	105	Sto	1	1M	175
Call Blocker	10.6K	3.1	Pho	0	1.4K	3	Sto	0	3.1K	200
Ringtone Maker	10.6K	4.6	Mic	2	4.9K	9	Sto	1	5.4K	9
Blood Donor	10M	10.7	Loc	1	28K	74	Sto	0	3.2K	68
Burger King	1.9M	13	Loc	1	219K	83	Sto	1	106K	27
Doctor On Demand	7.2M	12.2	Loc	0	97K	75	Sto	2	536K	267
Crackle	2.6M	12.6	Loc	0	11K	31	Sto	2	363K	61
Samsung Cloud Print	70K	12.8	Pho	2	312K	113	NFC	0	16K	8
AVG Antivirus	2.2M	11.5	Set	–	–	–	Sto	0	20K	26
Tiny Scanner	2M	15.7	Cam	0	1.6K	17	Sto	2	40K	19

[1] Loc–Location, Pho–Phone State, Mic–Microphone, Acc–Accounts, Set–User Settings, Cam–Camera, Sto–External Storage, NFC–Near Field Communications

TABLE I: Case study results.

modified app. For example, Hogarth generated the following path condition for the final handler before using the camera:

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

This path condition checks that the phone is connected to the internet and examines the structure of a command received from the malware command-and-control server, looking for a takePhoto command for the front-facing camera. Other path conditions in the app included several clauses our expert did not report in his analysis because he found them irrelevant For example, Hogarth will include all loop postconditions in the path condition; however, not all postconditions will necessarily be relevant to an auditor's interpretation of the eventual permission use.

In sum, Hogarth's results generally matched the results from our expert reverse engineer. The key differences were due to missing possible executions because the analysis is dynamic, and due to generating more verbose path conditions.

B. Case Study

Next, we applied Hogarth to a range of larger apps to evaluate its scalability and usefulness in a more realistic settig. Specifically, we sought to evaluate whether Hogarth could correctly identify triggers, what information we could learn from the path conditions it generated, and how it scaled to large apps. We selected 12 apps from the dataset used by Micinski et al.'s paper on AppTracer [16], which inferred information related to trigger diagrams (see Section VI for more discussion). To choose apps, we ordered Micinski et al.'s list of apps by the number of permission uses reported in that paper, most to least. From this list, we chose the first 12 that did not use multidex[2], which Hogarth does not currently support. We considered only apps that included uses of two different permissions (excluding internet and wifi state, which AppTracer's authors considered uninteresting).

We instrumented each app and generated a trace for it by manually running it. We attempted to cover as much app behavior as we could find, e.g., by clicking on all buttons and screens we could (excluding behavior which costs money). We then used the PScout [31] database to identify API calls in the trace that required permissions. Next, we chose two distinct, "interesting" permissions and selected one use of each at random. Our interesting permissions, in order of priority, were camera (Cam), microphone (Mic), location (Loc), write user settings (Set), user accounts (Acc), external storage (Sto), phone state (Pho), and Near-Field Communication (NFC). For example, if an app used Mic, Set, and Acc, we chose one random use of Mic and one random use of Set.

C. Results

The results of our case study are shown in Table I. Each row in the table corresponds to an app in our dataset. The columns are split into three parts: app metadata (app name, total number of logged lines, and size of app bytecode), results for the first permission use, and results for the second use. For each permission we report the length of the longest path from the permission use to the trigger. For example, if an app accessed location within the onCreate method, this size would be zero. If the app instead registered a callback via setOnClickHandler and the use occurred in that callback, the length of the path would be 1. Last, we report two performance statistics: the number of instructions Hogarth symbolically stepped through and the wall-clock runtime (average of three runs). We ran Hogarth's analysis step on an 2.5GHz Intel Core i7, 16GB RAM, and an SSD, running Mac OS 10.13.1.

a) Triggers: To evaluate the extent to which Hogarth was useful for determing permission-use triggers, we compared its output to Micinski et al.'s AppTracer. AppTracer infers triggers using a dynamic tracing architecture similar to the one used by Hogarth. However, AppTracer uses a temporal locality heuristic to attribute permission uses to UI events [16]. This heuristic can create ambiguities when triggers are temporally far from their associated permission uses; Hogarth's symbolic path tracing was designed to resolve these ambiguities.

We found that, in fact, the data we collected matched AppTracer's results for every permission use where the AppTracer authors marked themselves as certain. In addition, Hogarth allowed us to understand triggers for apps about which AppTracer provided uncertain results (*Ovia*, *Flip*, *Cloud Print*). In *Grubhub* and *Ringtone Maker*, Hogarth was able to identify

[2] a feature that allows writing apps with more than one Dalvik file to circumvent bytecode size restrictions

Reason	Num. Apps
Check if File Available	5 / 12
Permission Check to Avoid Crashes	5 / 12
Check if Resource Available	4 / 12
User Configuration	3 / 12
Parsing Network Data	2 / 12

TABLE II: Themes Observed in Path Conditions.

UI-related triggers that AppTracer missed because they were too far away temporally. We therefore conclude that Hogarth can successfully improve on AppTracer's ability to audit permission-use triggers and identify which are interactive.

b) Path Conditions: Next, we sought to understand how the path conditions produced by Hogarth could be used to help understand why permissions were used by apps. To do this, we examined the path conditions in each of the trigger diagrams.

Most path conditions produced by Hogarth include 20-100 clauses, and we were able to examine all of them within five minutes. Broadly, we found that path conditions alone allowed us to understand the app's interaction with the framework, but were less helpful at understanding app state. This is because the Android framework's semantics is consistent across apps (for example, several apps call checkSelfPermission to ensure they have a permission), but understanding parts of the path conditions related to app state often requires reading the app's code. Often this was complicated by obfuscation.

Table II details several themes we observed from looking at the path conditions from our case study apps. The most frequent checks we observed in path conditions related to the app being careful about accessing resources to avoid the app crashing. For example, *Grubhub* called File.exists from within a thread to check that a path exists before writing an app log file for analytics purposes. Next, several apps checked that a particular resource (such as fine-grained location or telephony data) was available before accessing the resource. For example, *Samsung Cloud Print* checked to ensure IMEI was available before accessing it to send to analytics data to the app's server. Last, we saw several apps that examined the app's configuration or network requests.

We also inspected each of the produced path conditions to consider how much irrelevant information they contained. We observed two main cases in which Hogarth produced irrelevant information. The first was when apps use large third-party libraries. These libraries are often complicated and generate many branches, which end up in the app's path conditions. For example, Flipp called a networking library to parse data coming from a server, which generated a large number of calls to equalsIgnoreCase as part of parsing a packet. Second, Hogarth often produced large path conditions when it examined long-running threads, which often performed irrelevant operations before using a permission. In each of these cases, we were able to inspect the path condition and (within several minutes) ascertain the relevant aspects of the path conditions.

c) Performance: Finally, we observe that, even given traces with millions of entries, Hogarth takes at most a few minutes. Across our experiments, runtime is usually dominated by the amount of time it takes to parse the log. This is because symbolic path tracing is performed in a demand-driven way, only looking at parts of the trace relevant to the permission use. We observed that, occasionally, Hogarth explores large and irrelevant portions of the log. This is because Hogarth performs symbolic tracing on a per-callback basis. For example, in *DoctorOnDemand*, Hogarth made 536K calls to the step function because it explored an execution of a particular thread that lasted the length of the execution. In one case, Set in *AVG Antivirus*, Hogarth timed out because it explored hundreds of similar threads, each of which had very long traces. We believe this problem could be addressed by introducing search heuristics to skip over portions of the trace.

We believe that Hogarth greatly reduces the task of inferring trigger diagrams. All of the apps in our case study consist of at least several thousand methods, and Hogarth allowed us to understand the conditions under which permissions were used within a few minutes. By contrast, reverse engineering the apps from our validation study took several hours per trigger. Our case study apps were much larger and often included obfuscated code, so we believe that performing this task for those apps would have been significantly more challenging.

D. Limitations and Threats to Validity

Hogarth has several limitations. First, as Hogarth is a dynamic analysis, it is necessarily incomplete, i.e., it will find app behavior covered in at least one trace, but will miss app behaviors not included in traces. Based on our experience, however, it still produces useful results. Second, Hogarth relies on the EdgeMiner dataset to identify possibly callback registrations. We found several instances in which this dataset was missing certain methods. We thus used an amended dataset in our experiments, but it is possible we might have missed some cases. We also needed to add special handling of Intent objects, which are not callbacks but do effectively trigger other callbacks to run. Last, Hogarth does not support multidex (apps with multiple Dalvik bytecode files) or instrumenting native code, since SymDroid does not include these features. Both could be added with further engineering effort.

There are several threats to the validity of our results. First, we only study 12 apps in our case study, which may not be indicative of the Android app landscape as a whole. We suspect we might have seen different uses of permissions if we studied lesser-known or explicitly malicious apps. Last, we do not have ground truth for our case study apps, meaning our path conditions could fail to include relevant information. However, we did not observe this in our validation study.

VI. 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 various aspects of security-relevant actions in Android apps. Pegasus [12] analyzes apps to infer Permission Event Graphs (PEGs), which describe the relationship between Android events and permission uses. In contrast to Hogarth's trigger

diagrams, PEGs do not include predicates about the app state. Unlike Hogarth, Pegasus requires a system model to operate, and its implementation defines models for 64 methods in the Android API. Unfortunately, this is too small to scale to any to any of the apps we studied in either our validation or case study (which utilize thousands of methods). Hogarth operates without the need for a complete system model by inferring inter-callback connections based on observed registrations.

Several systems use dataflow analyses to identify triggering conditions in apps. DroidSIFT [13] builds data-dependency graphs using a context-sensitive, flow-sensitive, interprocedural dataflow analysis to identify either user interactions or system events for sensitive resource use. Similarly, AppContext [9] identifies interprocedural control-flow paths from program entry points to potentially malicious behaviors. Then AppContext performs a dataflow analysis to identify the conditions that may trigger malicious behaviors.

Other systems apply symbolic execution to identify triggers in apps. AppIntent [6] first uses dataflow 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. TriggerScope [10] uses a combination of static analysis and symbolic execution to identify particularly complex trigger conditions associated with potentially malicious code. The tool attempts to detect "logic bombs" by identifying path conditions that are abnormally complex when simplified.

Hogarth has three key differences with these four systems. First, because we rely on a minimal model of the system, our approach is more resilient to the changes in Android from version to version. Second, because we use 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. However, because Hogarth depends on a representative corpus of dynamic traces, we cannot guarantee that all permission uses will be exercised. We see Hogarth as complementary to these systems, in that it achieves greater precision and scalability at the cost of completeness.

FuzzDroid [14] uses a mutation fuzzer to drive execution of an app toward a specific target location. IntelliDroid [15] uses static analysis to identify an overapproximation of inputs that could trigger malicious activity and then dynamically executes them to prune false positives. Similarly, SmartDroid [7] determines interprocedural control-flow paths from app entry points to possibly malicious activity. SmartDroid then executes the app on a modified version of Android, attempting all possible interactions to determine the set of triggers necessary to progress from the entry point to the potentially malicious behavior. These approaches identify a single path that reaches a target, whereas Hogarth identifies the set of conditions that could lead to sensitive resource use.

Lastly, AppTracer [16] uses dynamic analysis to discover what user interactions temporally precede sensitive resource uses. Instead of temporal heuristics, Hogarth uses symbolic path tracing to infer much richer contextual information about sensitive resource uses and identify interactive triggers AppTracer misses. (This comparison is detailed in Section V-C.)

Taint and Flow Analysis for Android: TaintDroid [32] modifies the Android firmware to perform system-wide dynamic taint-tracking and notifies the user whenever sensitive data is leaked. Phosphor [33] provides similar taint-tracking, but instead modifies the JVM to improve portability. FlowDroid [5] uses static dataflow analysis find sensitive data leaks. User-centric dependence analysis [34] uses a dataflow dependence analysis to characterize "normal" data consumption behaviors along paths from user inputs to sensitive resource uses. These tools all focus on data flow, which is orthogonal to the control-flow dependencies that Hogarth discovers.

Concolic Execution: One style of symbolic execution is *concolic execution* [21], [35], [36], in which programs are instrumented to track symbolic expressions at run-time along with their concrete counterparts in the actual run. Hogarth is similar in spirit in that it performs symbolic execution on a path corresponding to a program execution. However, rather than using the result to branch and explore further executions, Hogarth presents path conditions as output in trigger diagrams. Moreover, rather than concretize at system calls, Hogarth introduces symbolic variables to represent those calls and then finds path conditions in terms of those variables.

Dynamic Slicing: Dynamic slicing [37], [38] is a related approach that has also been used to investigate contextual security in Android apps [39]. A static program slice is the minimal set of program expressions that *may* affect a given program value. A dynamic slice is a minimal subset of the program that actually *does* affect the target value for a provided input. Dynamic slicing considers a specific execution trace and effectively eliminates all code in the static slice that is unrelated to the target for the trace. Our approach is related as it reasons about dependencies that were observed to cause a permission use. However, Hogarth infers symbolic path conditions and coarser control-flow information—a callback sequence—rather than a more precise dynamic slice.

VII. Conclusion and Future Work

In this paper we introduced Hogarth, a new tool to create trigger diagrams that explain the cause of permission uses in Android apps. Hogarth works by using two novel techniques. The first is symbolic path tracing, which performs symbolic execution along a dynamically-generated program trace. The second is path splicing, which splices together callbacks with the callbacks in which they are registered. This allows Hogarth to work without a detailed model of the Android framework. Our implementation performs demand-driven path splicing, which enabled it to scale to 12 top applications from Google Play, generally taking only minutes to explore apps that generated logs comprising millions of lines.

We believe that Hogarth provides a promising proof of concept, and we think the approach has the potential to be of significant aid in debugging, reverse engineering, and other auditing purposes.

REFERENCES

[1] R. Balebako, J. Jung, W. Lu, L. F. Cranor, and C. Nguyen, ""little brothers watching you": Raising awareness of data leaks on smartphones," in *Proceedings of the 9th Symposium on Usable Privacy and Security*, ser. SOUPS '13. New York, NY, USA: ACM, 2013, pp. 12:1–12:11. [Online]. Available: http://doi.acm.org/10.1145/2501604.2501616

[2] I. Liccardi, J. Pato, D. J. Weitzner, H. Abelson, and D. De Roure, "No technical understanding required: Helping users make informed choices about access to their personal data," in *Proceedings of the 11th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, ser. MOBIQUITOUS '14. Brussels, Belgium, Belgium: ICST, 2014, pp. 140–150. [Online]. Available: http://dx.doi.org/10.4108/icst.mobiquitous.2014.258066

[3] I. Shklovski, S. D. Mainwaring, H. H. Skúladóttir, and H. Borgthorsson, "Leakiness and creepiness in app space: Perceptions of privacy and mobile app use," in *Proceedings of the 32nd ACM Conference on Human Factors in Computing Systems*, ser. CHI '14. New York, NY, USA: ACM, 2014, pp. 2347–2356. [Online]. Available: http://doi.acm.org/10.1145/2556288.2557421

[4] C. Thompson, M. Johnson, S. Egelman, D. Wagner, and J. King, "When it's better to ask forgiveness than get permission: Attribution mechanisms for smartphone resources," in *Proceedings of the 9th Symposium on Usable Privacy and Security*, ser. SOUPS '13. New York, NY, USA: ACM, 2013, pp. 1:1–1:14. [Online]. Available: http://doi.acm.org/10.1145/2501604.2501605

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[6] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "Appintent: analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 20th ACM conference on Computer & communications security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 1043–1054. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516676

[7] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications," in *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12. New York, NY, USA: ACM, 2012, pp. 93–104. [Online]. Available: http://doi.acm.org/10.1145/2381934.2381950

[8] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 89–99. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818768

[9] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *Proceedings of the 37th IEEE International Conference on Software Engineering*, ser. ICSE '15, vol. 1. Florence, Italy: ACM, May 2015, pp. 303–313.

[10] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *Proceedings of the 37th IEEE Symposium on Security and Privacy*, ser. IEEE S&P '16. San Jose, CA: IEEE Press, May 2016, pp. 377–396.

[11] S. Blackshear, B.-Y. E. Chang, and M. Sridharan, "Selective control-flow abstraction via jumping," in *Proceedings of the 30th ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '15. New York, NY, USA: ACM, 2015, pp. 163–182. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814293

[12] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song, "Contextual policy enforcement in android applications with permission event graphs," in *Proceedings of the 20th Network and Distributed System Security Symposium*, ser. NDSS '13. San Diego, CA: Internet Society, 2013, p. 234.

[13] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the 21st ACM Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1105–1116. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660359

[14] S. Rasthofer, S. Arzt, S. Triller, and M. Pradel, "Making malory behave maliciously: Targeted fuzzing of android execution environments," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Buenos Aires, Argentina: ACM, 2017, pp. 300–311.

[15] M. Y. Wong and D. Lie, "Intellidroid: A targeted input generator for the dynamic analysis of android malware," in *Proceedings of the 23rd Network and Distributed System Security Symposium*, ser. NDSS '16. San Diego, CA: Internet Society, 2016.

[16] K. Micinski, D. Votipka, R. Stevens, N. Kofinas, M. L. Mazurek, and J. S. Foster, "User interactions and permission use on android," in *Proceedings of the 35th ACM Conference on Human Factors in Computing Systems*, ser. CHI '17. Denver, Colorado, USA: ACM, 2017, pp. 362–373. [Online]. Available: http://doi.acm.org/10.1145/3025453.3025706

[17] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. android and mr. hide: fine-grained permissions in android applications," in *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12. Raleigh, NC, USA: ACM, 2012, pp. 3–14.

[18] J. Jeon, K. K. Micinski, and J. S. Foster, "Symbolic execution for dalvik bytecode," University of Maryland, 2012, (Tech Report, CS-TR-5022).

[19] F.-D. Limited, "F-droid - free and open source android repository," F-Droid Limited, 2017, (Accessed 4-11-2017). [Online]. Available: https://f-droid.org/

[20] M. Parkour, "Contagio mobile," Mila Parkour, 2017, (Accessed 4-11-2017). [Online]. Available: http://contagiominidump.blogspot.com/

[21] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 322–335. [Online]. Available: http://dx.doi.org/10.1145/1180405.1180445

[22] C. Cadar, D. Dunbar, and D. Engler, "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*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855756

[23] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework," in *Proceedings of the 22nd Network and Distributed System Security Symposium*, ser. NDSS '15. San Diego, CA: Internet Society, 2015. [Online]. Available: http://www.internetsociety.org/doc/edgeminer-automatically-detecting\\-implicit-control-flow-transitions\\-through\\-android-framework

[24] P. Software, "Jeb decompiler," PNF Software, 2017, (Accessed 5-19-2017). [Online]. Available: www.pnfsoftware.com

[25] Axet, "Misbothering sms receiver," 2015, (Accessed 4-11-2017). [Online]. Available: https://f-droid.org/packages/com.github.axet.callrecorder/

[26] A. Yalon, "Misbothering sms receiver," 2015, (Accessed 8-25-2017). [Online]. Available: https://f-droid.org/repository/browse/?fdfilter=Misbothering+SMS+Receiver&fdid=net.yxejamir.misbotheringsms

[27] R. Treffer, "Contact merger," 2014, (Accessed 4-11-2017). [Online]. Available: https://f-droid.org/repository/browse/?fdfilter=contacts&fdid=de.measite.contactmerger

[28] L. Stefanko, "Android trojan spy goes 2 year undetected," 2015, (Accessed 4-11-2017). [Online]. Available: http://b0n1.blogspot.com/2015/04/android-trojan-spy-goes-2-years.html?spref=tw

[29] Google, inc. (2017) Camera2basic android sample app. [Online]. Available: https://github.com/googlesamples/android-Camera2Basic

[30] M. Rogers, "Dendroid malware can take over your camera, record audio, and sneak into google play," Lookout Inc, 2014, (Accessed 4-11-2017). [Online]. Available: https://blog.lookout.com/blog/2014/03/06/dendroid/

[31] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proceedings of the 19th ACM Conference on Computer and Communications Security*, ser. CCS '12.

New York, NY, USA: ACM, 2012, pp. 217–228. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382222

[32] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 393–407. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924971

[33] J. Bell and G. Kaiser, "Phosphor: Illuminating dynamic data flow in commodity jvms," in *Proceedings of the 29th ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 83–101. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660212

[34] K. O. Elish, D. Yao, and B. G. Ryder, "User-centric dependence analysis for identifying malicious mobile apps," in *Proceedings of the 1st Workshop on Mobile Security Technologies*, ser. MoST '12. San Jose, CA: IEEE Press, 2012.

[35] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 28th ACM Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 213–223. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065036

[36] K. Sen, D. Marinov, and G. Agha, "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*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

[37] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=800078.802557

[38] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proceedings of the 12th ACM Conference on Programming Language Design and Implementation*, ser. PLDI '90. New York, NY, USA: ACM, 1990, pp. 246–256. [Online]. Available: http://doi.acm.org/10.1145/93542.93576

[39] Y. Chen, W. You, Y. Lee, K. Chen, X. Wang, and W. Zou, "Mass discovery of android traffic imprints through instantiated partial execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2017, pp. 815–828.