Symbolic Path Tracing to Find Android Permission-Use Triggers

Kristopher K. Micinski
University of Maryland
College Park, MD, USA
micinski@cs.umd.edu

Thomas Gilray
University of Maryland
College Park, MD, USA
tgilray@cs.umd.edu

Daniel Votipka
University of Maryland
College Park, MD, USA
dvotipka@cs.umd.edu

Michelle L. Mazurek
University of Maryland
College Park, MD, USA
mmazurek@cs.umd.edu

Jeffrey S. Foster
University of Maryland
College Park, MD, USA
jfoster@cs.umd.edu

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. 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. Hogarth also showed that third-party libraries often piggyback on app permissions, but only if those permissions are already available. 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.

KEYWORDS

Symbolic execution, Android, symbolic path tracing, triggers

ESEC/FSE 2018, 4–9 November, 2018, Lake Buena Vista, Florida, United States

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Kristopher K. Micinski, Thomas Gilray, Daniel Votipka, Michelle L. Mazurek, and Jeffrey S. Foster. 2018. Symbolic Path Tracing to Find Android Permission-Use Triggers. In *Proceedings of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, Florida, United States, 4–9 November, 2018 (ESEC/FSE 2018),* 11 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deciding whether an Android app is safe to use is difficult. While apps need to list the *permissions* they request—such as contacts, calendar, camera, and location—the permission list, on its own, is insufficient for auditing an app's actual security implications [5, 19, 26, 30]. This is because permissions provide persistent, *carte blanche* access. Thus, among other important security questions, one critical concern is, what dynamically *triggers* the app to use its permissions, and under what conditions [21, 33]? For example, a third-party library might check if the app it has been linked into has location access and, only if it does, register for location updates opportunistically.

We would like to automatically infer such conditional trigger information, but doing so is challenging for two main reasons. First, Android is heavily *callback oriented*, which can make the connection between the trigger and the eventual permission use indirect. For example, a button click might queue up a task that launches a thread that eventually uses the permission some time later. Second, discovering the conditions on data corresponding to a permission use requires precise reasoning about paths through the app, which is difficult because of the way app code interleaves with the large and complex Android framework. Other researchers have explored a range of approaches to related problems [10, 14, 21, 23, 34, 36–38], but to our knowledge no prior work has demonstrated a scalable and precise way to compute conditional trigger information. (Section 6 discusses related work in more detail.)

In this paper, we introduce Hogarth,[1] a novel Android app analysis tool that performs post-hoc inference of conditional triggers for observed permission uses. 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 selects an observed permission use within those traces. Hogarth then constructs a *trigger diagram* showing the observed triggers for the selected permission in terms

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

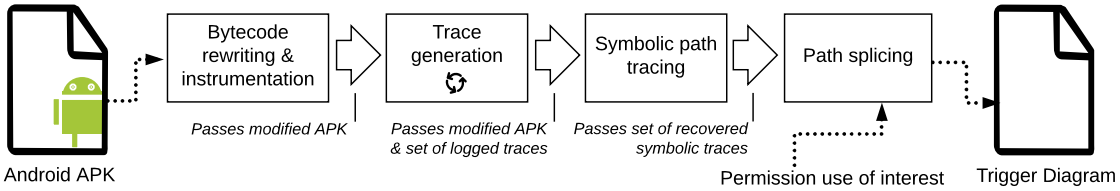

Figure 1: Hogarth's architecture.

of a sequence of callbacks and their associated conditions on data. Thus, trigger diagrams compactly represent the conditional trigger information for the observed use. (Section 2 gives an overview of tracing and trigger diagrams.)

Hogarth infers trigger diagrams using what we call *symbolic path tracing* and *path splicing*. Given the target permission use, Hogarth begins at the start of the callback method containing the call and performs symbolic execution, but *only* along the path observed in the trace. During symbolic execution, we replace 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. Hogarth then repeats this process, working backward from handler to handler until it reaches a root, which is the trigger. This novel design allows the symbolic executor, guided by actual execution data, to quickly reach the permission use of interest. At the same time, each trace—which originally just described a single execution—is generalized to a path condition. Moreover, Hogarth achieves all this without needing app source code or a detailed model of each framework method. (Section 3 describes trigger diagram inference formally.)

Hogarth is implemented on top of Redexer [18], a Dalvik bytecode rewriting tool, and SymDroid [17], 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 4 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 [20] and the Contagio Malware dump [22]. 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 [21]. 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 [21]. Hogarth also gave us new insights into the conditions under which apps use permissions. For example, we observed that third-party libraries often check for permissions and then use them if available. Because developers often include these libraries without fully understanding them, we posit this might mislead app users and app developers. 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 5 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.

2 OVERVIEW

Figure 1 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 needs a permission). Hogarth then performs *symbolic path tracing* to infer *path conditions* (ϕ) 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*. Nodes of a trigger diagram are either callback methods or the permission use itself. An edge from A to B labeled with path condition ϕ indicates that callback A may register callback B, or directly use permission B, under condition ϕ.

Running Example. In the remainder of this section, we illustrate how Hogarth can be used to understand a use of the location permission in the *Ovia Pregnancy* app, one of the subject programs in our case study. Figure 2 shows the trigger diagram generated by Hogarth for this permission use. (Note that Hogarth currently outputs a textual description of the nodes and edges of the graph; we manually drew the visualization in the figure.) From this diagram, we can see that `BaseActivity.onStop` creates a thread that executes `flurry.Task.run`, which uses the location permission. This kind of indirect control flow is common in Android apps. We describe the path conditions ϕ_1 and ϕ_2 further below.

Bytecode Instrumentation and Trace Generation. Hogarth adds logging instrumentation to app bytecode using Redexer [18], 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

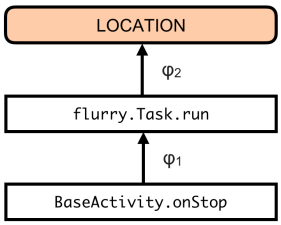

Figure 2: Trigger diagram for the *Ovia Pregnancy* app.

```
Method > 1 BaseActivity.onStop(SplashActivity@207023171)
...
BBEntry 1 2477570
API > 1 Handler.post(Handler@...,flurry.Task@222621802)
API < 1 Handler.post()
...
Method < 1 BaseActivity.onStop()
```

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

Figure 3: Two partial traces for Ovia.

thread id to a `ConcurrentLinkedQueue`. Our logging instrumentation creates a background thread to dequeue log entries and asynchronously 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 `Method >` 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 we do not record object fields to improve performance. 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 class 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.

Symbolic Path Tracing. Next, Hogarth performs *symbolic path tracing* by running symbolic execution [7, 8] 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've 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., $!\text{argument}_1.\text{isChangingConfigurations}()$ 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:

$$\begin{aligned}&\text{“passive”} \neq 0 \wedge \text{staticfd}_{113} \neq 0 \wedge \text{new}_{18}.\text{x} \geq \text{new}_{19}.\text{y}\\ &\wedge\ \text{staticfd}_{50}.\text{c.getSystemService(“connectivity”)}\\ &\qquad .\text{getActiveNetworkInfo}().\text{isConnected}()\\ &\wedge\ \text{com.flurry.sdk.fd.class} \neq 0 \wedge \text{com.flurry.sdk.fo.class} \neq 0 \wedge !\text{staticfd}_{18}.\text{isEmpty}()\\ &\text{staticfd}_{50}.\text{c.getSystemService(“connectivity”).getActiveNetworkInfo} \neq 1\\ &\wedge\ \text{staticfd}_{50}.\text{c.getSystemService(“connectivity”).getActiveNetworkInfo} \neq 2\\ &\wedge\ \text{staticfd}_{50}.\text{c.getSystemService(“connectivity”).getActiveNetworkInfo} \neq 3\\ &\ldots\\ &\wedge\ \text{staticfd}_{50}.\text{c.getSystemService(“connectivity”).getActiveNetworkInfo} \neq 8\\ &\wedge\ \text{staticfd}_{50}.\text{c.checkCallingOrSelfPermission(“ACCESS\_FINE\_LOCATION”)}\end{aligned}$$

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 to ensure that the app has permission to access location.

Path Splicing. Finally, Hogarth performs *path splicing*, which connects the symbolic path traces together and summarizes them to construct a trigger diagram. In theory, we could run symbolic path tracing on every handler observed in the trace, but this is more expensive than neccessary. Thus, Hogarth actually 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 [9] 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 4.

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 5), we characterize permission uses with a set of *codes*. First, we begin with the trigger. From Figure 2, we see the trigger was `onStop`, called when the framework stops an app. Thus we code the trigger as BG, meaning it is a background or non-UI trigger. Next, we examine the path conditions. We see that the app branched depending on app state (e.g., testing $\texttt{new}_{35}$), yielding the code AppSt; it branched depending on framework state (e.g., testing values returned from `getSystemService`), yielding the code FwSt; and it also checked whether the app has location permission, yielding the code PChk. Finally, we note that `flurry` is an analytics library, so we code the permission use as Ana, meaning it occurs in an analytics library (because the API call is in `flurry.Task.run`).

3 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.

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

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

Figure 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 $r \leftarrow \texttt{new}\ C$ 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 Trace Generation

As previously discussed, we begin by instrumenting and executing the program to gather a set of 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 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

pt	::=	$\vec{ct}$	[program trace]
ct	::=	$C.m(\vec{v})\ \vec{ti}$	[callback trace]
ti	::=	$C.m$	[app call]
	|	$C.m(\vec{v})$	[API call]
	|	`then` | `else`	[branch]
v	::=	ϵ | $C@id$	[value]

Figure 5: Traces.

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 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 2.

3.2 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 *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 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.

$state$	::=	$\langle \vec{i}, rf, \kappa, \phi, \vec{ti} \rangle$	[machine state]
rf	:	$regs \rightharpoonup a$	[register file]
κ	::=	ϵ | $(r, rf, \vec{i}) : \kappa$	[stack]
ϕ	::=	a	[path condition]
a	::=	c | $C@id$ | $\oplus \vec{a}$	[abstract value]
	|	r | $a.f$ | $C.m(\vec{a})$	

(a) Abstract machine domains.

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

(b) Abstract machine semantics.

Figure 6: Formalism for symbolic path tracing.

[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.

3.3 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 2.

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.

4 IMPLEMENTATION

We implemented Hogarth using Redexer [18] and SymDroid [17], 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 the core implementation of Hogarth's symbolic path tracing), and comprises around 10K lines of code. There were several unique challenges in implementing Hogarth.

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 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.

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 given 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 were uninteresting, and their presence made 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.

Inter-callback connections. Recall that we use the EdgeMiner [9] 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 ids. 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 path splicing. In practice, traces are quite long—in our case study, up to around 10 million lines. Thus, for scalability, 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 demand-driven approach dramatically achieved dramatic speedups compared to an earlier implementation that timed out on even modestly sized logs.

5 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. We observed that apps often access permissions conditioned upon the state of the app and framework, and found that libraries more frequently check whether the app has permission to access a resource.

5.1 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 [27], identified each API call corresponding to a permission use, and then manually read through the code line-by-line to construct a trigger diagram. In addition to its decompiler, JEB simplified the 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 the expert's manual results.

Our first three apps were selected from the F-Droid repository [20]. *Call Recorder* [4] 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* [35] mutes notification 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* [31] 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 [22]. 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 [16] that allows the user to take a picture by pressing a button. We modified *Camera2Basic* by injecting the Dendroid [24] 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 the camera in the modified *Camera2Basic*. For example, Hogarth generated the following path condition for the final handler before using the camera:

```
isConnected(getActiveNetworkInfo(getSystemService(
      getApplicationContext(var83),"connectivity"))) ∧
getActiveNetworkInfo(getSystemService(getApplicationContext(var83),
      "connectivity")) ∧
readLine(new16) ∧
find(matcher(compile("\(([^)]+)\)"),readLine(new16))) ∧
contains(readLine(new16),"takephoto(") ∧
equals(group(matcher(compile("\(([^)]+)\)"),readLine(new16)),1),"")) ∧
equalsIgnoreCase(get(new7,0),"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, these clauses included loop postconditions that were necessary for the app to proceed out of a loop but not directly related to the permission use. By design, 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.

5.2 Case Study

Next, we applied Hogarth to a range of apps to gain experience in a more realistic setting. We selected 12 apps from the dataset used by Micinski et al.'s paper on AppTracer [21], which inferred information related to trigger diagrams (see Section 6 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 [3] 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.

In order to characterize each selected permission use, we examined its associated trigger diagram. We used an iterative, inductive coding process, in which the codebook, or set of qualitative categories, is refined as new data is analyzed, and previously coded cases are revisited for consistency [29]. Coding was performed by a single researcher. Our final codebook is as follows: Two codes describe the trigger, which may be a related UI event (UI) or in the background (BG), meaning either a system event or an unrelated UI event. Four codes describe the path conditions, which may depend on app state (AppSt), framework state (FwSt), the permissions an app has (PChk), or checks related to the filesystem (FChk). Finally, three codes indicate where the permission use occurred: in the app code (App), in a known third-party analytics library (Ana), or in some other third-party library (Lib).

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

5.3 Results

The results of our case study are shown in Table 1. 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 trigger and 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. We also report the set of codes for the path condition and the location where the use occurred. 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.

Triggers. Hogarth allowed us to determine the trigger of the selected permissions in each of our apps, except for the first permission use of *AVG Antivirus*, which timed out after an hour (more discussion below). We observed several broad trends. First, in the majority of the cases, finding the trigger required going back at least one callback, sometimes two, indicating that the connection between triggers and permission uses is indeed often indirect. Second, the two most sensitive permissions, camera and microphone, were triggered by a UI interaction, which is as expected [21], and both occurred in app code rather than in library code.

Third, the storage permission was used in the background in every case except *AVG Antivirus*, which opened files (to scan for viruses) on the virus-scanning screen. We observed that storage was frequently used in library code.

Finally, the use of Mic in *Ringtone Maker* had an interesting pattern: After the user interaction trigger, the app displayed a permission authorization dialog, after which Mic was used. We expect this pattern is common when a permission is first used. However, we did not see it elsewhere because the apps in our dataset use the older Android permissions API, which does not require just-in-time permission authorization.

Micinski et al.'s AppTracer infers trigger-relevant information using a dynamic tracing architecture similar to the one used by Hogarth. However, it uses a temporal locality heuristic to attribute permission uses to UI events [21]. This heuristic can create ambiguities when triggers are temporally far from their associated permission uses; Hogarth's symbolic path tracing should be able to resolve these ambiguities. We found that, in fact, the data we collected matched AppTracer's results for all of the permission uses 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 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.

App Name	Log (Lines)	Size (MB)	Perm Use 1[1]	Trigger[2] [len]	Path Cond[3]	Loc[4]	Steps	Tm (s)	Perm Use 2[1]	Trigger[2] [len]	Path Cond[3]	Loc[4]	Steps	Tm (s)
Ovia Pregnancy	1.7M	11.4	Loc	BG [1]	AppSt, PChk, FwSt	Ana	46K	26	Sto	BG [1]	FChk	Ana	12K	24
Grubhub	4.4M	11.8	Loc	UI [1]	FwSt	App	1.8K	46	Sto	BG [2]	FChk	Ana	98K	66
Flipp	7.9M	7.6	Loc	UI [1]	AppSt	App, FwSt	96K	105	Sto	BG [1]	AppSt, FwSt	Lib	1M	175
Call Blocker	10.6K	3.1	Pho	BG [0]	AppSt	App	1.4K	3	Sto	BG [0]	AppSt	App	3.1K	200
Ringtone Maker	10.6K	4.6	Mic	UI [2]	FChk, FwSt	App	4.9K	9	Sto	BG [1]	AppSt, FwSt	App	5.4K	9
Blood Donor	10M	10.7	Loc	UI [1]	AppSt, FwSt	App	28K	74	Sto	BG [0]	FChk, AppSt, FwSt	App	3.2K	68
Burger King	1.9M	13	Loc	BG [1]	FwSt, PChk	Lib	219K	83	Sto	BG [1]	AppSt	Lib	106K	27
Doctor On Demand	7.2M	12.2	Loc	UI [0]	AppSt, FwSt	App	97K	75	Sto	BG [2]	PChk, FwSt, AppSt	Lib	536K	267
Crackle	2.6M	12.6	Loc	BG [0]	FChk, FwSt, AppSt	App	11K	31	Sto	BG [2]	PChk, FChk	Lib	363K	61
Samsung Cloud Print	70K	12.8	Pho	UI [2]	AppSt, FwSt	App	312K	113	NFC	UI [0]	AppSt	App	16K	8
AVG Antivirus	2.2M	11.5	Set	–	–	–	–	–	Sto	UI [0]	AppSt	App	20K	26
Tiny Scanner	2M	15.7	Cam	UI [0]	AppSt	App	1.6K	17	Sto	BG [2]	PChk, AppSt	Ana	40K	19

[1] Loc–Location, Pho–Phone State, Mic–Microphone, Acc–Accounts, Set–User Settings, Cam–Camera, Sto–External Storage, NFC–Near Field Communications
[2] UI–Related UI, BG–Background [3] AppSt–App State, FwSt–Framework State, PChk–Permission Check, FChk–Framework Check
[4] App–App Code, Lib–3rd Party Library, Ana–Analytics Library

Table 1: Case study results.

Path Conditions. We found that in the majority of the cases, permission uses coded Lib or Ana also were coded as PChk. Investigating further, we found that the permission checks occur in the library code—the libraries first check to see if the app has a permission and, if so, they use that permission. We speculate that this is a potential security risk. Users might be granting permissions to apps for some app-specific purpose, not realizing that a third-party library within the app will piggyback on that permission.

When permission uses were based on framework state (FwSt), or performed some file-related check (FChk), Hogarth was helpful in interpreting the path condition. For example, we observed many uses of permissions that would check the state of network connectivity, e.g., before downloading a file. Similarly, file-related checks were frequent; we hypothesize this was to avoid crashing the app. However, Hogarth was less helpful in understanding path conditions related to app state (AppSt). This is because Hogarth shows the path condition in terms of variable names, which are often obfuscated in library code and are hard to understand even in unobfuscated app code without having the source available.

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 allow Hogarth to skip over portions of the trace.

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 various aspects of the context of security-relevant actions in Android apps. Pegasus [10] 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.

Several systems use dataflow analyses to identify triggering conditions in apps. DroidSIFT [38] 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 [36] 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 [37] 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 [14] 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. Third, because we depend on a representative corpus of dynamic traces, we cannot guarantee that all permission uses will be exercised.

FuzzDroid [23] uses a genetic mutation fuzzer to drive execution of an app toward a specific target location. IntelliDroid [34] uses static analysis to identify an overapproximation of inputs that could trigger malicious activity and then dynamically executes these inputs to prune false positives. Similarly, SmartDroid [39] 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 UI interactions to determine the set of UI triggers necessary to progress from the initial 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 [21] 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 that AppTracer misses. (This comparison is detailed in Section 5.3 above.)

Taint and Flow Analysis for Android. TaintDroid [13] modifies the Android firmware to perform system-wide dynamic taint-tracking and notifies the user whenever sensitive data is leaked. Phosphor [6] provides similar taint-tracking, but instead modifies the JVM to improve portability. FlowDroid [2] uses static dataflow analysis find sensitive data leaks. User-centric dependence analysis [12] uses a dataflow dependence analysis to characterize "normal" data consumption behaviors along paths from user inputs to sensitive resource uses for malware detection. 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* [8, 15, 25], 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 its trigger diagrams. Moreover, rather than concretize symbolic variables 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 [1, 32] is a related approach that has also been used to investigate contextual security in Android apps [11]. 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 program 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 provided trace. Our approach is related in that 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.

7 CONCLUSION AND FUTURE WORK

In this paper we introduced Hogarth, a new tool that uses symbolic path tracing and path splicing to create trigger diagrams that explain the causes and conditions of permission uses in Android apps. We found that Hogarth improves on prior work, provides new insight into the conditions under which apps use permissions, and is scalable, generally taking only minutes to explore apps that generated logs comprising millions of lines.

In future work, we plan to modify Hogarth to perform symbolic path tracing on demand, harnessing human intuition to reduce the amount of necessary work. We also plan to explore how to most usefully visualize trigger diagrams. We envision productive interactions between a live visualization and an app auditor, allowing fine-grained probing into the source code as needed.

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] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. In *Proceedings of the 12th Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 246–256. https://doi.org/10.1145/93542.93576

[2] 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

[3] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 217–228. https://doi.org/10.1145/2382196.2382222

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

[5] Rebecca Balebako, Jaeyeon Jung, Wei Lu, Lorrie Faith Cranor, and Carolyn Nguyen. 2013. "Little Brothers Watching You": Raising Awareness of Data Leaks on Smartphones. In *Proceedings of the 9th Symposium on Usable Privacy and Security (SOUPS '13)*. ACM, New York, NY, USA, Article 12, 11 pages. https://doi.org/10.1145/2501604.2501616

[6] 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

[7] 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

[8] 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

[9] 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)*. Internet Society, San Diego, CA. http://www.internetsociety.org/doc/edgeminer-automatically-detecting\-implicit-control-flow-transitions\-through\-android-framework

[10] 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.

[11] Yi Chen, Wei You, Yeonjoon Lee, Kai Chen, XiaoFeng Wang, and Wei Zou. 2017. Mass Discovery of Android Traffic Imprints through Instantiated Partial Execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, USA, 815–828.

[12] 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.

[13] 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

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

[18] 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, Raleigh, NC, USA, 3–14.

[19] Ilaria Liccardi, Joseph Pato, Daniel J. Weitzner, Hal Abelson, and David De Roure. 2014. 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 (MOBIQUITOUS '14)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, 140–150. https://doi.org/10.4108/icst.mobiquitous.2014.258066

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

[21] 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

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

[23] 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, 300–311.

[24] 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).

[25] 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

[26] Irina Shklovski, Scott D. Mainwaring, Halla Hrund Skúladóttir, and Höskuldur Borgthorsson. 2014. Leakiness and Creepiness in App Space: Perceptions of Privacy and Mobile App Use. In *Proceedings of the 32nd Annual ACM Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2347–2356. https://doi.org/10.1145/2556288.2557421

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

[28] 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).

[29] David R Thomas. 2006. A general inductive approach for analyzing qualitative evaluation data. *American journal of evaluation* 27, 2 (2006), 237–246.

[30] Christopher Thompson, Maritza Johnson, Serge Egelman, David Wagner, and Jennifer King. 2013. 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 (SOUPS '13)*. ACM, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/2501604.2501605

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

[32] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449. http://dl.acm.org/citation.cfm?id=800078.802557

[33] 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

[34] 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.

[35] 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).

[36] 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

[37] 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 & communications security (CCS '13)*. ACM, New York, NY, USA, 1043–1054. https://doi.org/10.1145/2508859.2516676

[38] 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

[39] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. 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 (SPSM '12)*. ACM, New York, NY, USA, 93–104. https://doi.org/10.1145/2381934.2381950