Checking Interaction-Based Declassification Policies for Android Using Symbolic Execution*

Kristopher Micinski[1], Jonathan Fetter-Degges[1], Jinseong Jeon[1], Jeffrey S. Foster[1], and Michael R. Clarkson[2]

[1] University of Maryland, College Park
{micinski,jonfd,jsjeon,jfoster}@cs.umd.edu
[2] Cornell University
clarkson@cs.cornell.edu

Abstract. Mobile apps can access a wide variety of secure information, such as contacts and location. However, current mobile platforms include only coarse access control mechanisms to protect such data. In this paper, we introduce *interaction-based declassification policies*, in which the user's interactions with the app constrain the release of sensitive information. Our policies are defined extensionally, so as to be independent of the app's implementation, based on sequences of security-relevant events that occur in app runs. Policies use LTL formulae to precisely specify which secret inputs, read at which times, may be released. We formalize a semantic security condition, *interaction-based noninterference*, to define our policies precisely. Finally, we describe a prototype tool that uses symbolic execution of Dalvik bytecode to check interaction-based declassification policies for Android, and we show that it enforces policies correctly on a set of apps.

Keywords: Information flow, program analysis, symbolic execution.

1 Introduction

The Android platform includes a *permission* system that aims to prevent apps from abusing access to sensitive information, such as contacts and location. Unfortunately, once an app is installed, it has *carte blanche* to use any of its permissions in arbitrary ways at run time. For example, an app with location and Internet access could continuously broadcast the device's location, even if such behavior is not expected by the user.

To address this limitation, this paper presents a new framework for Android app security based on *information flow control* [**?**] and user interactions. The key idea behind our framework is that users naturally express their intentions about information release as they interact with an app. For example, clicking a

* This research was supported in part by NSF grants CNS-1064997 and CNS-1421373, AFOSR grants FA9550-12-1-0334 and FA9550-14-1-0334, the partnership between UMIACS and the Laboratory for Telecommunication Sciences, and the National Security Agency.

button may permit an app to release a phone number over the Internet. Or, as another example, toggling a radio button from "coarse" to "fine" and back to "coarse" may temporarily permit an app to use fine-grained GPS location rather than a coarse-grained approximation.

To model these kinds of scenarios, we introduce *interaction-based declassification policies*, which extensionally specify what information flows may occur after which sequences of *events*. Events are GUI interactions (e.g., clicking a button), inputs (e.g., reading the phone number), or outputs (e.g., sending over the Internet). A policy is a set of *declassification conditions*, written $\phi \rhd S$, where ϕ is a linear-time temporal logic (LTL) [**?**] formula over events, and S is a sensitivity level. If ϕ holds at the time an input occurs, then that input is declassified to level S. We formalize a semantic security condition, *interaction-based noninterference* (IBNI), over sets of event *traces* generated by an app. Intuitively, IBNI holds of an app and policy if observational determinism [**?**] holds after all inputs have been declassified according to the policy. (Section **??** describes policies further, and Section **??** presents our formal definitions.)

We introduce ClickRelease, a static analysis tool to check whether an Android app and its declassification policy satisfy IBNI. ClickRelease generates event traces using SymDroid [**?**], a Dalvik bytecode symbolic executor. ClickRelease works by simulating user interactions with the app and recording the resulting execution traces. In practice, it is not feasible to enumerate all program traces, so ClickRelease generates traces up to some *input depth* of n GUI events. ClickRelease then synthesizes a set of logical formulae that hold if and only if IBNI holds, and uses Z3 [**?**] to check their satisfiability. (Section **??** describes ClickRelease in detail.)

To validate ClickRelease, we used it to analyze four Android apps, including both secure and insecure variants of those apps. We ran each app variant under a range of input depths, and confirmed that, as expected, ClickRelease scales exponentially. However, we manually examined each app and its policy, and found that an input depth of at most 5 is sufficient to guarantee detection of a security policy violation (if any) for these cases. We ran ClickRelease at these minimum input depths and found that it correctly passes and fails the secure and insecure app variants, respectively. Moreover, at these depths, ClickRelease takes just a few seconds to run. (Section **??** describes our experiments.)

In summary, we believe that ClickRelease takes an important step forward in providing powerful new security mechanisms for mobile devices. We expect that our approach can also be used in other GUI-based, security-sensitive systems.

2 Example Apps and Policies

We begin with two example apps that show interesting aspects of interaction-based declassification policies.

Bump app. The boxed portion of Fig. **??** gives (simplified) source code for an Android app that releases a device's unique ID and/or phone number. This

```
1  public class BumpApp extends Activity {
2   protected void onCreate(...) {
3    Button sendBtn = (Button) findViewById(...);
4    CheckBox idBox = (CheckBox) findViewById(...);
5    CheckBox phBox = (CheckBox) findViewById(...);
6    TelephonyManager manager = TelephonyManager.getTelephonyManager();
7     final int id = manager.getDeviceId();
8     final int ph = manager.getPhoneNumber();
9    idBox.setChecked(false); phBox.setChecked(false);
10   sendBtn.setOnClickListener(
11    new OnClickListener() {
12     public void onClick(View v) {
13      if (idBox.isChecked())
14       Internet.sendInt(id); //Internet.sendInt(ph);
15      if (phBox.isChecked())
16       Internet.sendInt(ph); //Internet.sendInt(id);
17 }})}}
```

$$\mathsf{id!} * \wedge(\mathcal{F}(\mathsf{sendBtn!unit} \wedge \mathit{last}(\mathsf{idBox}, \mathsf{true}))) \rhd \mathit{Low},$$
$$\mathsf{ph!} * \wedge(\mathcal{F}(\mathsf{sendBtn!unit} \wedge \mathit{last}(\mathsf{phBox}, \mathsf{true}))) \rhd \mathit{Low}$$

Fig. 1. "Bump" app and policy.

app is inspired by the Bump app, which let users tap phones to share selected information with each other. We have interspersed an insecure variant of the app in the red code on lines **??** and **??**, which we will discuss in Section **??**.

Each screen of an Android app is implemented using a class that extends Activity. When an app is launched, Android invokes the onCreate method for a designated main activity. (This is part of the *activity lifecycle* [**?**], which includes several methods called in a certain order. For this simple app, and the other apps used in this paper, we only need a single activity with this one lifecycle method.) That method retrieves (lines **??**–**??**) the GUI IDs of a button (marked "send") and two checkboxes (marked "ID" and "phone"). The onCreate method next gets an instance of the TelephonyManager, uses it to retrieve the device's unique ID and phone number information, and unchecks the two checkboxes as a default. Then it creates a new callback (line **??**) to be invoked when the "send" button is clicked. When called, that callback releases the user's ID and/or phone number, depending on the checkboxes.

This app is written to work with ClickRelease, a symbolic execution tool we built to check whether apps satisfy interaction-based declassification policies. As we discuss further in Section **??**, ClickRelease uses an executable model of Android that abstracts away some details that are unimportant with respect to security. While a real app would release information by sending it to a web server, here we instead call a method Internet.sendInt. Additionally, while real apps include an XML file specifying the screen layout of buttons, checkboxes, and so on, ClickRelease creates those GUI elements on demand at calls to findViewById

(since their screen locations are unimportant). Finally, we model the ID and phone number as integers to keep the analysis simpler.

ClickRelease symbolically executes paths through subject apps, recording a *trace* of *events* that correspond to certain method calls. For example, one path through this app generates a trace

$$\mathsf{id!42}, \mathsf{ph!43}, \mathsf{idBox!true}, \mathsf{sendBtn!unit}, \mathsf{netout!42}$$

Each event has a *name* and a *value*. Here we have used names id and ph for secret inputs, idBox and sendBtn for GUI inputs, and netout for the network send. In particular, the trace above indicates 42 is read as the ID, 43 is read as the phone number, the ID checkbox is selected, the send button is clicked (carrying no value, indicated by unit), and then 42 is sent on the network. In ClickRelease, events are generated by calling certain methods that are specially recognized. For example, ClickRelease implements the manager.getDeviceId call as both returning a value and emitting an event.

Notice here that in the trace, callbacks to methods such as idBox and sendBtn correspond to user interactions. The key idea behind our framework is that these actions convey the user's intent as to which information should be released. Moreover, traces also contain actions relevant to information release—here the reads of the ID and phone number, and the network send. Thus, putting both user interactions and security-sensitive operations together in a single trace allows our policies to enforce the user's intent.

The policy for this example app is shown at the bottom of Fig. **??**. Policies are comprised of a set of *declassification conditions* of the form $\phi \triangleright S$, where ϕ is an LTL formula describing event traces and S is a security level. Such a condition is read, "At any input event, if ϕ holds at that position of the event trace, then that input is declassified at level S." For this app there are two declassification conditions. The top condition declassifies (to *Low*) an input that is a read of the ID at any value (indicated by $*$), if sometime in the future (indicated by the $\mathcal{F}$ modality) the send button is clicked and, when that button is clicked, the last value of the ID checkbox was true. (Note that *last* is not primitive, but is a macro that can be expanded into regular LTL.) The second declassification condition does the analogous thing for the phone number.

To check such a policy, ClickRelease symbolic executes the program, generating per-path traces; determines the classification level of every input; and checks that every pair of traces satisfies noninterference. Note that using LTL provides a very general and expressive way to describe the sequences of events that imply declassification. For example, here we precisely capture that only the last value of the checkbox matters for declassification. For example, if a user selects the ID checkbox but then unselects it and clicks send, the ID may not be released.

Although this example relies on a direct flow, ClickRelease can also detect implicit flows. Section **??** defines an appropriate version of noninterference, and the experiments in Section **??** include a subject program with an implicit flow.

Notice this policy depends on the app reading the ID and phone number when the app starts. If the app instead waited until after the send button were

```
18 public class ToggleRes extends Activity { ...
19  LocSharer mLocSharer = new LocSharer();
20  RadioManager mRadio = new RadioManager();
21  protected void onCreate(...) { ...}
22  private class LocSharer implements LocationListener { ...
23   public LocSharer(RadioManager rm) {
24    lm = (LocationManager) getSystemService(LOCATION_SERVICE);
25    lm.requestLocationUpdates(mCurrentProvider, SHARE_INTERVAL, distance, this);
26   }
27   public void onLocationChanged(Location l) {
28    if (mRadio.mFine) {
29     Internet.sendInt(l.mLatitude);
30     Internet.sendInt(l.mLongitude);
31    } else {
32     Internet.sendInt(l.mLatitude & 0xffffff00 );
33     Internet.sendInt(l.mLongitude & 0xffffff00 );
34  } } }
35  private class RadioManager
36   implements OnClickListener {
37    public boolean mFine = false;
38    public void onClick(View v) { mFine = !mFine; }
39 } }
```

$$\mathsf{longitude!} * \wedge last(\mathsf{mRadio}, \mathsf{true}) \rhd Low,$$
$$\mathsf{longitude!} * \wedge last(\mathsf{mRadio}, \mathsf{false}) \rhd MaskLower8$$

Fig. 2. Location sharing app and policy.

clicked, it would violate this policy. We could address this by replacing the $\mathcal{F}$ modality by $\mathcal{P}$ (past) in the policy, and we could form a disjunction of the two policies if we wanted to allow either implementation. More generally, we designed our framework to be sensitive to such choices to support reasoning about secret values that change over time. We will see an example next.

Location resolution toggle app. Fig. **??** gives code for an app that shares location information, either at full or truncated resolution depending on a radio button setting. The app's onCreate method displays a radio button (code not shown) and then creates and registers a new instance of RadioManager to be called each time the radio button is changed. That class maintains field mFine as true when the radio button is set to full resolution and false when set to truncated resolution.

Separately, onCreate registers LocSharer to be called periodically with the current location. It requests location updates by registering a callback with the LocationManager system service. When called, LocSharer releases the location, either at full resolution or with the lower 8 bits masked, depending on mFine.

The declassification policy for longitude appears below the code; the policy for latitude is analogous. This policy allows the precise longitude to be released when mRadio is set to fine, but only the lower eight bits to be released if mRadio

Primitives	$p ::= n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{unit} \mid f(p_1, \ldots, p_i)$
Events	$\eta ::= \mathit{name}!p$
Traces	$t ::= \eta\ \mathit{list}$

(a) Event and Trace Definitions.

Policies	$P ::= C_1, C_2, \ldots$
Conditions	$C ::= \phi \rhd S$
Security Levels	$S ::= \mathit{High} \mid \mathit{Low} \mid \mathit{MaskLower8} \mid \ldots$
Atoms	$A ::= \mathit{name}!s \mid s \oplus s$
Messages	$s ::= x \mid p \mid *$
Formulae	$\phi ::= A \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \exists x.\phi \mid \forall x.\phi$ $\mid \mathcal{X}\phi \mid \phi\ \mathcal{U}\ \phi \mid \mathcal{G}\phi \mid \mathcal{F}\phi \mid \phi\ \mathcal{S}\ \phi \mid \mathcal{P}\phi$

(b) Interaction-based Declassification Policy Language.

Fig. 3. Formal definitions.

is set to coarse. Here ClickRelease knows that at the *MaskLower8* level, it should consider outputs to be equivalent up to differences in the lower 8 bits.

Finally, notice that this policy does not use the future modality. This is deliberate, because location may be read multiple times during the execution, at multiple values, and the security level of those locations should depend on the state of the radio button at that time. For example, consider a trace

$$\mathsf{mRadio!false}, \mathsf{longitude!}v_1, \mathsf{mRadio!true}, \mathsf{longitude!}v_2$$

The second declassification condition ($\mathsf{longitude!}* \wedge \mathit{last}(\mathsf{mRadio}, \mathsf{false})$) will match the event with v_1, since the last value of mRadio was false, and thus v_1 may be declassified only to *MaskLower8*. Whereas the first declassification condition will match the event with v_2, hence it may be declassified to *Low*.

3 Program Traces and Security Definition

Next, we formally define when a set of program traces satisfies an interaction-based declassification policy.

3.1 Program Traces

Fig. **??**(a) gives the formal syntax of events and traces. *Primitives* p are terms that can be carried by events, e.g., values for GUI events, secret inputs, or network sends. In our formalism, primitives are integers, booleans, and terms constructed from primitives using uninterpreted constructors f. As programs execute, they produce a *trace* t of *events* η, where each event $\mathit{name}!p$ pairs an event name *name* with a primitive p. We assume event names are partitioned into

those corresponding to inputs and those corresponding to outputs. For all the examples in this paper, all names are inputs except netout, which is an output.

Due to space limitations, we omit details of how traces are generated. These details, along with definition of our LTL formulas, can be found in appendices **??** and **??**, respectively. Instead, we simply assume there exists some set $\mathcal{T}$ containing all possible traces a given program may generate. For example, consider the insecure variant bump app in Fig. **??**, which replaces the black code with the red code on lines lines **??** and **??**. This app sends the phone number when the email box is checked and vice-versa. Thus, its set $\mathcal{T}$ contains, among others, the following two traces:

$$\mathsf{id!0}, \mathsf{ph!0}, \mathsf{idBox!true}, \mathsf{sendBtn!unit}, \mathsf{netout!0} \ (1)$$
$$\mathsf{id!0}, \mathsf{ph!1}, \mathsf{idBox!true}, \mathsf{sendBtn!unit}, \mathsf{netout!1} \ (2)$$

In the first trace, ID and phone number are read as 0, the ID checkbox is selected, the button is clicked, and 0 is sent. The second trace is similar, except the phone number and sent value are 1. Below, we use these traces to show this program violates its security policy.

3.2 Interaction-based Declassification Policies

We now define our policy language precisely. Fig. **??**(b) gives the formal syntax of declassification policies. A policy P is a set of *declassification conditions* C_i of the form $\phi_i \triangleright S_i$, where ϕ_i is an LTL formula describing when an input is declassified, and S_i is a *security level* at which the value in that event is declassified.

As is standard, security levels S form a lattice. For our framework, we require that this lattice be finite. We include *High* and *Low* security levels, and we can generalize to arbitrary lattices in a straightforward way. Here we include the *MaskLower8* level from Fig. **??** as an example, where $\mathit{Low} \sqsubseteq \mathit{MaskLower8} \sqsubseteq \mathit{High}$. Note that although we include *High* in the language, in practice there is no reason to declassify something to level *High*, since then it remains secret.

The *atomic predicates* A of LTL formulae match events, e.g., atomic predicate $\mathit{name}!p$ matches exactly that event. We include $*$ for matches to arbitrary primitives. We allow event values to be variables that are bound in an enclosing quantifier. The atomic predicates also include atomic arithmetic statements; here $\oplus$ ranges over standard operations such as $+$, $<$, etc. The combination of these lets us describe complex events. For example, we could write $\exists x.\mathit{spinner}!x \wedge x > 2$ to indicate the *spinner* was selected with a value greater than 2.

Atomic predicates are combined with the usual boolean connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$) and existential and universal quantification. Formulae include standard LTL modalities $\mathcal{X}$ (next), $\mathcal{U}$ (until), $\mathcal{G}$ (always), $\mathcal{F}$ (future), $\phi \, \mathcal{S} \, \psi$ (since), and $\mathcal{P}\phi$ (past). We include a wide range of modalities, rather than a minimal set, to make policies easier to write. Formulae also include $\mathit{last}(\mathit{name}, p)$, which is syntactic sugar for $\neg(\mathit{name}!*) \, \mathcal{S} \, \mathit{name}!p$. We assume a standard interpretation of LTL formulae over traces [**?**]. We write $t, i \models \phi$ if trace t is a model of ϕ at position i in the trace.

Next consider a trace $t \in \mathcal{T}$ for an arbitrary program. We write $\mathit{level}(t, P, i)$ for the security level that policy P assigns to the event $t[i]$:

$$\mathit{level}(t, P, i) = \begin{cases} \bigsqcap_{\phi_j \rhd S_j \in P} \{S_j \mid t, i \models \phi_j\} & t[i] = \mathit{name}!p \\ \mathit{Low} & t[i] = \mathsf{netout}!p \end{cases}$$

In other words, for inputs, we take the greatest lower bound (the most declassified) of the levels from all declassification conditions that apply. We always consider network outputs to be declassified. Notice that if no policy applies, the level is H by definition of greatest lower bound.

For example, consider trace (1) above with respect to the policy in Fig. **??**. At position 0, the LTL formula holds because the ID box is eventually checked and then the send button is clicked, so $\mathit{level}((1), P, 0) = \mathit{Low}$. However, $\mathit{level}((1), P, 1) = \mathit{High}$ because no declassification condition applies for ph (phBox is never checked). And $\mathit{level}((1), P, 4) = \mathit{Low}$, because that position is a network send.

Next consider applying this definition to the GUI inputs. As written, we have $\mathit{level}((1), P, 2) = \mathit{level}((1), P, 3) = \mathit{High}$. However, our app is designed to leak these inputs. For example, an adversary will learn the state of idBox if they receive a message with an ID. Thus, for all the subject apps in this paper, we also declassify all GUI inputs as *Low*. For the example in Fig. **??**, this means adding the conditions $\mathsf{idBox}! * \rhd \mathit{Low}$, $\mathsf{phBox}! * \rhd \mathit{Low}$, and $\mathsf{sendBtn}! * \rhd \mathit{Low}$. In general, the security policy designer should decide the security level of GUI inputs.

Next, we can apply *level* pointwise across a trace and discard any trace elements that are below a given level S. We define

$$\mathit{level}(t, P)^S[i] = \begin{cases} t[i] & \mathit{level}(t, P, i) \sqsubseteq S \\ \tau & \text{otherwise} \end{cases}$$

We write $\mathit{level}(t, P)^{S,in}$ for the same filtering, except output events (i.e., network sends) are removed as well. Considering the traces (1) and (2) again, we have

$$\begin{aligned} \mathit{level}((1), P)^{\mathit{Low}} &= \mathsf{id}!0, \mathsf{idBox}!\mathsf{true}, \mathsf{sendBtn}!\mathsf{unit}, \mathsf{netout}!0 \\ \mathit{level}((2), P)^{\mathit{Low}} &= \mathsf{id}!0, \mathsf{idBox}!\mathsf{true}, \mathsf{sendBtn}!\mathsf{unit}, \mathsf{netout}!1 \\ \mathit{level}((1), P)^{\mathit{Low},in} &= \mathsf{id}!0, \mathsf{idBox}!\mathsf{true}, \mathsf{sendBtn}!\mathsf{unit} \\ \mathit{level}((2), P)^{\mathit{Low},in} &= \mathsf{id}!0, \mathsf{idBox}!\mathsf{true}, \mathsf{sendBtn}!\mathsf{unit} \end{aligned}$$

Finally, we can define a program to satisfy noninterference if, for every pair of traces such that the inputs at level S are the same, the outputs at level S are also the same. To account for generalized lattice levels such as *MaskLower8*, we also need to treat events that are equivalent at a certain level as the same. For example, at *MaskLower8*, outputs `0xffffffff` and `0xffffff00` are the same, since they do not differ in the upper 24 bits. Thus, we assume for each security level S there is a appropriate equivalence relation $=_S$, e.g., for *MaskLower8*, it compares elements ignoring their lower 8 bits. Note that $x =_{\mathit{Low}} y$ is simply $x = y$ and $x =_{\mathit{High}} y$ is always true.

Definition 1 (Interaction-based Noninterference (IBNI)). *A program satisfies security policy P, if for all S and for all $t_1, t_2 \in \mathcal{T}$ (the set of traces of the program) the following holds:*

$$level(t_1, P)^{S,in} =_S level(t_2, P)^{S,in} \implies level(t_1, P)^{S} =_S level(t_2, P)^{S}$$

Looking at traces for the insecure app, we see they violate non-interference, because $level((1), P)^{Low,in} = level((2), P)^{Low,in}$, but $level((1), P)^{Low} \neq level((2), P)^{Low}$ (they differ in the output). We note that our definition of noninterference makes it a 2-hypersafety property [**?**,**?**].

4 Implementation

We built a prototype tool, ClickRelease, to check whether Android apps obey the interaction-based declassification policies described in Section **??**. ClickRelease is based on SymDroid [**?**], a symbolic executor for Dalvik bytecode, which is the bytecode format to which Android apps are compiled. As is standard, SymDroid computes with *symbolic expressions* that may contain *symbolic variables* representing sets of values. At conditional branches that depend on symbolic variables, SymDroid invokes Z3 [**?**] to determine whether one or both branches are feasible. As it follows branches, SymDroid extends the current *path condition*, which tracks branches taken so far, and forks execution when multiple paths are possible. Cadar and Sen [**?**] describe symbolic execution in more detail.

SymDroid uses the features of symbolic execution to implement nondeterministic event inputs (such as button clicks or spinner selections), up to a certain bound. Since we have symbolic variables available, we also use them to represent arbitrary secret inputs, as discussed below in Sec. **??**. There are several issues that arise in applying SymDroid to checking our policies, as we discuss next.

4.1 Driving App Execution

Android apps use the Android framework's API, which includes classes for responding to events via callbacks. We could try to account for these callbacks by symbolically execution Android framework code directly, but past experience suggests this is intractable: the framework is large, complicated, and includes native code. Instead, we created an *executable model*, written in Java, that mimics key portions of Android needed by our subject apps. Our Android model includes facilities for generating clicks and other GUI events (such as the View, Button, and CheckBox classes, among others). It also includes code for LocationManager, TelephonyManager, and other basic Android classes.

In addition to code modeling Android, the model also includes simplified versions of Java library classes such as StringBuffer and StringBuilder. Our versions of these APIs implement unoptimized versions of methods in Java and escape to internal SymDroid functions to handle operations that would be unduly complex to symbolically execute. For instance, SymDroid represents Java String objects

with OCaml strings instead of Java arrays of characters. It thus models methods such as String.concat with internal calls to OCaml string manipulation functions. Likewise, reflective methods such as Class.getName are handled internally.

For each app, we created a driver that uses our Android model to simulate user input to the GUI. The driver is specific to the app since it depends on the app's GUI. The driver begins by calling the app's onCreate method. Next it invokes special methods in the Android model to inject GUI events. There is one such method for each type of GUI element, e.g., buttons, checkboxes, etc. For example, Trace.addClick(id) generates a click event for the given id and then calls the appropriate event handler. The trace entry contains the event name for that kind of element, and a value if necessary. Event handlers are those that the app registered through standard Android framework mechanisms, e.g., in onCreate.

Let m be the number of possible GUI events. To simulate one arbitrary GUI event, the driver uses a block that branches m ways on a fresh symbolic variable, with a different GUI action in each branch. Typical Android apps never exit unless the framework kills them, and thus we explore sequences of events only up to a user-specified *input depth* n. Thus, in total, the driver will execute at least m^n paths.

4.2 Symbolic Variables in Traces

In addition to GUI inputs, apps also use secret inputs. We could use SymDroid to generate concrete secret inputs, but instead we opt to use a fresh symbolic variable for each secret input. For example, the call to manager.getDeviceId in Fig. **??** returns a symbolic variable, and the same for the call to manager.getPhoneNumber. This choice makes checking policies using symbolic execution a bit more powerful, since, e.g., a symbolic integer variable represents an arbitrary 32-bit integer. Note that whenever ClickRelease generates a symbolic variable for a secret input, it also generates a trace event corresponding to the input.

Recall that secret inputs may appear in traces, and thus traces may now contain symbolic variables. For example, using α_i's as symbolic variables for the secret ID and phone number inputs, the traces (1) and (2) become

$$\mathsf{id!}\alpha_1, \mathsf{ph!}\alpha_2, \mathsf{idBox!true}, \mathsf{sendBtn!unit}, \mathsf{netout!}\alpha_2 \ (1')$$
$$\mathsf{id!}\alpha_1, \mathsf{ph!}\alpha_2, \mathsf{idBox!true}, \mathsf{sendBtn!unit}, \mathsf{netout!}\alpha_2 \ (2')$$

We must take care when symbolic variables are in traces. Recall *level* checks $t, i \models \phi$ and then assigns a security level to position i. If ϕ depends on symbolic variables in t, we may not be able to decide this. For example, if the third element in $(1')$ were $\mathsf{idBox!}\alpha_3$, then we would need to reason with conditional security levels such as $\mathit{level}(t, P, 0) = \textbf{if } \alpha_3 \textbf{ then } \mathit{Low} \textbf{ else } \mathit{High}$. We avoid the need for such reasoning by only using symbolic variables for secret inputs, and by ensuring the level assigned by a policy does not depend on the value of a secret input. We leave supporting more complex reasoning to future work.

4.3 Checking Policies with Z3

Each path explored by SymDroid yields a pair (t, Φ), where t is the trace and Φ is the path condition. ClickRelease uses Z3 to check whether a given set of such trace–path condition pairs satisfies a policy P. Recall that Definition **??** assumes for each S there is an $=_S$ relation on traces. We use the same relation below, encoding it as an SMT formula. For our example lattice, $=_{High}$ produces true, $=_{Low}$ produces a conjunction of equality tests among corresponding trace elements, and $=_{MaskLower8}$ produces the conjunction of equality tests of the bitwise-and of every element with `0xffffff00`.

Given a trace t, let t' be t with its symbolic variables primed, so that the symbolic variables of t and t' are disjoint. Given a path condition Φ, define Φ' similarly. Now we can give the algorithm for checking a security policy.

Algorithm 1 *To check a set $\mathcal{T}$ of trace–path condition pairs, do the following. Let P be the app's security policy. Apply* level *across each trace to obtain the level of each event. For each (t_1, Φ_1) and (t_2, Φ_2) in $\mathcal{T} \times \mathcal{T}$, and for each S, ask Z3 whether the following formula (the negation of Definition* **??***) is unsatisfiable:*

$$\mathit{level}(t_1, P)^{S,in} =_S \mathit{level}(t_2', P)^{S,in} \wedge \mathit{level}(t_1, P)^{S} \neq_S \mathit{level}(t_2', P)^{S} \wedge \Phi_1 \wedge \Phi_2'$$

If no such formula is unsatisfiable, then the program satisfies noninterference.

We include Φ_1 and Φ_2' to constrain the symbolic variables in the trace. More precisely, t_1 represents a *set* of concrete traces in which its symbolic variables are instantiated in all ways that satisfy Φ_1, and analogously for t_2'.

If the above algorithm finds an unsatisfiable formula, then Z3 returns a counterexample, which SymDroid uses in turn to generate a pair of concrete traces as a counterexample. For example, consider traces (1') and (2') above, and prime symbolic variables in (2'). Those traces have the trivial path condition true, since neither branches on a symbolic input. Thus, the formula passed to Z3 will be:

$$\alpha_1 = \alpha_1' \wedge \mathsf{true} = \mathsf{true} \wedge \mathsf{unit} = \mathsf{unit} \wedge \big(\alpha_1 \neq \alpha_1' \vee \mathsf{true} \neq \mathsf{true} \vee \mathsf{unit} \neq \mathsf{unit} \vee \alpha_2 \neq \alpha_2'\big)$$

Thus we can see a satisfying assignment with $\alpha_1 = \alpha_1'$ and $\alpha_2 \neq \alpha_2'$, hence noninterference is violated.

4.4 Minimizing Calls to Z3

A naive implementation of the noninterference check generates n^2 equations, where n is the number of traces produced by ClickRelease to be checked by Z3. However, we observed that many of these equations correspond to pairs of traces with different sequences of GUI events. Since GUI events are low inputs in all our policies, these pairs trivially satisfy noninterference (the left-hand side of the implication in Definition **??** is false). Thus, we need not send those equations to Z3 for an (expensive) noninterference check.

We exploit this observation by organizing SymDroid's output traces into a tree, where each node represents an event, with the initial state at the root.

Traces with common prefixes share the same ancestor traces in the tree. We systematically traverse this tree using a cursor t_1, starting from the root. When t_1 reaches a new input event, we then traverse the tree using another cursor t_2, also starting from the root. As t_2 visits the tree, we do not invoke Z3 on any traces with fewer input events than t_1 (since they are not low-equivalent to t_1). We also skip any subtrees where input events differ.

5 Experiments

To evaluate ClickRelease, we ran it on four apps, including the two described in Section **??**. We also ran ClickRelease on several insecure variants of each app, to ensure it can detect the policy violations. The apps and their variants are:

- *Bump.* The bump app and its policy appear in Fig. **??**. The first insecure variant counts clicks to the send button sends the value of the ID after three clicks, regardless of the state of the ID checkbox. The second (indicated in the comments in the program text) swaps the released information—if the ID box is checked, it releases the phone number, and vice-versa.
- *Location toggle.* The location toggle app and its policy appear in Fig. **??**. The first insecure variant always shares fine-grained location information, regardless of the radio button setting. The second checks if coarse-grain information is selected. If so, it stores the fine-grained location (but does not send it yet). If later the fine-grained radio button is selected, it sends the stored location. Recall this is forbidden by the app's security policy, which allows the release only of locations received while the fine-grained option is set.
- *Contact picker.* We developed a contact picker app that asks the user to select a contact from a spinner and then click a send button to release the selected contact information over the network. The security policy for this app requires that no contact information leaks unless it is the last contact selected before the button click. (For example, if the user selects contact 1, selects contact 2, and then clicks the button, only contact 2 may be released.) Note that since an arbitrarily sized list of contacts would be difficult for symbolic execution (since then there would be an unbounded number of ways to select a contact), we limit the app to a fixed set of three contacts. The first insecure variant of this app scans the set of contacts for a specific one. If found, it sends a message revealing that contact exists before sending the actual selected contact. The second insecure variant sends a different contact than was selected.
- *WhereRU.* Lastly, we developed an app that takes push requests for the user's location and shares it depending on user-controlled settings. The app contains a radio group with three buttons, "Share Always," "Share Never," and "Share On Click." There is also a "Share Now" button that is enabled when the "Share On Click" radio button is selected. When a push request arrives, the security policy allows sharing if (1) the "Always" button is selected, or (2) the "On Click" button is selected and the user presses "Share Now." Note that, in the second case, the location may change between the time the request arrives

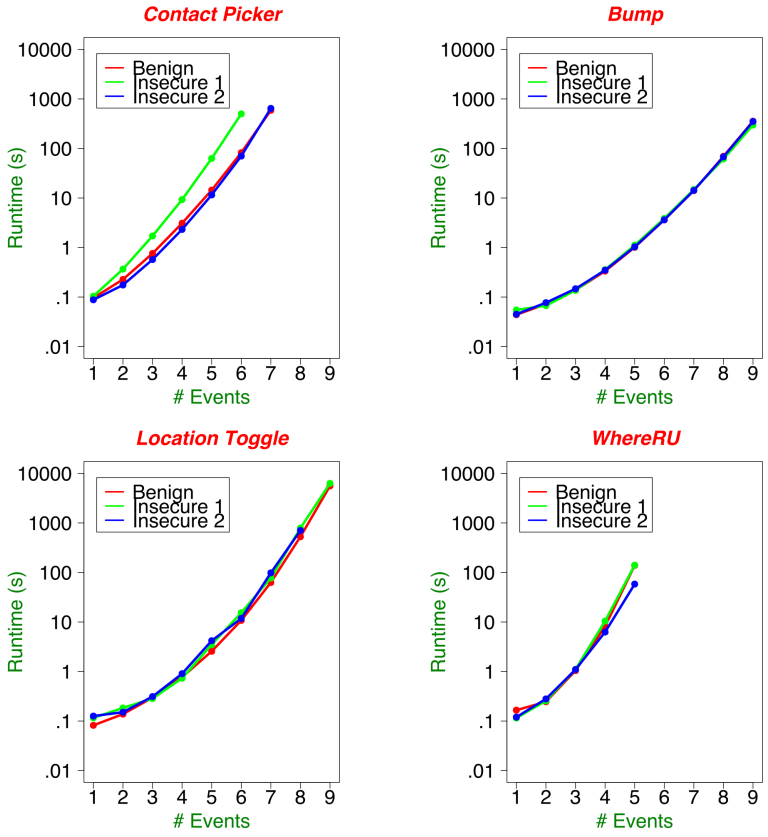

Fig. 4. Runtime vs. number of events.

and the time the user authorizes sharing; the location to be shared is the one in effect when the user authorized sharing, i.e., the one from the most recent location update before the button click. Also, rather than include the full Android push request API in our model, we simulated it using a basic callback. This app has two insecure variants. In the first one, when the user presses the "Share Now" button, the app begins continuously sharing (instead of simply sharing the single location captured on the button press). In the second, the app shares the location immediately in response to all requests.

Scalability. We ran our experiments on a 4-core i7 CPU @3.5GHz with 16GB RAM running Ubuntu 14. For each experiment we report the median of 10 runs.

In our first set of experiments, we measured how ClickRelease's performance varies with input depth. Figure **??** shows running time (log scale) versus input depth for all programs and variants. For each app, we ran to the highest input depth that completed in one hour.

App	Input Depth	Time (ms) Exploration	Analysis	Total
Bump	3	114	15	142
Bump (insecure 1)	5	2,100	1,577	3,690
Bump (insecure 2)	4	266	70	344
Location toggle	2	113	12	128
Location toggle (insecure 1)	2	143	12	163
Location toggle (insecure 2)	3	117	12	143
Contact picker	2	79	2	94
Contact picker (insecure 1)	2	325	27	361
Contact picker (insecure 2)	2	149	9	170
WhereRU	3	849	183	1,045
WhereRU (insecure 1)	3	860	234	1,108
WhereRU (insecure 2)	2	257	10	280

Fig. 5. Results at minimum input depth.

For each app, we see that running time grows exponentially, as expected. The maximum input depth before timeout (i.e., where each curve ends) ranges from five to nine. The differences have to do with the number of possible events at each input point. For example, WhereRU has seven possible input events, so it has the largest possible "fan out" and times out with an input depth of five. In contrast, Bump and Location Toggle have just three input events and time out with an input depth of nine. Notice also the first insecure variant of Contact Picker times out after fewer events than the other variants. Investigating further, this occurs due to that app's implicit flow (recall the app branches on the value of a secret input). Implicit flows cause symbolic execution to take additional branches depending on the (symbolic) secret value.

Minimum Input Depth. Next, for each variant, we manually calculated a *minimum* input depth guaranteed to find a policy violation. To do so, first we determined possible app GUI states. For example, in Bump (Fig. **??**), there is a state with idBox and phBox both checked, a state with just idBox checked, etc. Then we examined the policy and recognized that certain input sequences lead to equivalent states modulo the policy. For example, input sequences that click idBox an even number of times and then click send are all equivalent. Full analysis reveals that an input depth of three (which allows the checkboxes to be set any possible way followed by a button click) is sufficient to reach all possible states for this policy. We performed similar analysis on other apps and variants.

Fig. **??** summarizes the results of running with the minimum input depth for each variant, with the depths listed in the second column. We confirmed that, when run with this input depth, ClickRelease correctly reports the benign app variants as secure and the other app variants as insecure. The remaining columns of Fig. **??** report ClickRelease's running time (in milliseconds), broken down by the exploration phase (where SymDroid generates the set of symbolic traces) and the analysis phase (where SymDroid forms equations about this set

and checks them using Z3). Looking at the breakdown between exploration and analysis, we see that the former dominates the running time, i.e., most of the time is spent simply exploring program executions. We see the total running time is typically around a second or less, while for the first insecure variant of Bump it is closer to 4 seconds, since it uses the highest input depth.

Our results show that while ClickRelease indeed scales exponentially, to actually find security policy violations we need only run it with a low input depth, which takes only a small amount of time.

6 Limitations and Future Work

There are several limitations of ClickRelease we plan to address in future work.

Thus far we have applied ClickRelease to a set of small apps that we developed. There are two main engineering challenges in applying ClickRelease to other apps. First, our model of Android (Section **??**) only includes part of the framework. To run on other apps, it will need to be expanded with more Android APIs. Second, we speculate that larger apps may require longer input depths to go from app launch to interfering outputs. In these cases, we may be able to start symbolic execution "in the middle" of an app (e.g., as in the work of Ma et al. [**?**]) to skip uninteresting prefixes of input events.

ClickRelease also has several limitations related to its policy language. First, ClickRelease policies are fairly low level. Complex policies—e.g., in which clicking a certain button releases multiple pieces of information—can be expressed, but are not very concise. We expect as we gain more experience writing ClickRelease policies, we will discover useful idioms that should be incorporated into the policy language. Similarly, situations where several methods in sequence operate on and send information should be supported. Second, currently ClickRelease assumes there is a single adversary who watches `netout`. It should be straightforward to generalize to multiple output channels and multiple observers, e.g., to model inter-app communication. Third, we do not consider deception by apps, e.g., we assume the policy writer knows whether the `sendBtn` is labeled appropriately as "send" rather than as "exit." We leave looking for such deceptive practices to future work.

Finally, since ClickRelease explores a limited number of program paths it is not sound, i.e., it cannot guarantee the absence of policy violations in general. However, in our experiments we were able to manually analyze apps to show that exploration up to a certain input depth was sufficient for particular apps, and we plan to investigate generalizing this technique in future work.

7 Related Work

ClickRelease is the first system to enforce extensional declassification policies in Android apps. It builds on a rich history of research in usable security, information flow, and declassification.

One of the key ideas in ClickRelease is that GUI interactions indicate the security desires of users. Roesner et al. [**?**] similarly propose *access control gadgets* (ACGs), which are GUI elements that, when users interact with them, grant permissions. Thus, ACGs and ClickRelease both aim to better align security with usability [**?**]. ClickRelease addresses secure information flow, especially propagation of information after its release, whereas ACGs address only access control.

Android-based systems. TaintDroid [**?**] is a run-time information-flow tracking system for Android. It monitors the usage of sensitive information and detects when that information is sent over insecure channels. Unlike ClickRelease, TaintDroid does not detect implicit flows.

AppIntent [**?**] uses symbolic execution to derive the *context*, meaning inputs and GUI interactions, that causes sensitive information to be released in an Android app. A human analyst examines that context and makes an expert judgment as to whether the release is a security violation. ClickRelease instead uses human-written LTL formulae to specify whether declassifications are permitted. It is unclear from [**?**] whether AppIntent detects implicit flows.

Pegasus [**?**] combines static analysis, model checking, and run-time monitoring to check whether an app uses API calls and privileges consistently with users' expectations. Those expectations are expressed using LTL formulae, similarly to ClickRelease. Pegasus synthesizes a kind of automaton called a *permission event graph* from the app's bytecode then checks whether that automaton is a model for the formulae. Unlike ClickRelease, Pegasus does not address information flow.

Jia et al. [**?**] present a system, inspired by Flume [**?**], for run-time enforcement of information flow policies at the granularity of Android components and apps. Their system allows components and apps to perform trust declassification according to capabilities granted to them in security labels. In contrast, ClickRelease reasons about declassification in terms of user interactions.

Security type systems Security type systems [**?**] statically disallow programs that would leak information. O'Neill et al. [**?**] and Clark and Hunt [**?**] define interactive variants of noninterference and present security type systems that are sound with respect to these definitions.

Integrating declassification with security type systems has been the focus of much research. Chong and Myers [**?**] introduce *declassification policies* that conditionally downgrade security labels. Their policies use classical propositional logic for the conditions. ClickRelease can be seen as providing a more expressive language for conditions by using LTL to express formulae over events. SIF (Servlet Information Flow) [**?**] is a framework for building Java servlets with information-flow control. Information managed by the servlet is annotated in the source code with security labels, and the compiler ensures that information propagates in ways that are consistent with those labels. The SIF compiler is based on Jif [**?**], an information-flow variant of Java.

All of these systems require adding type annotations to terms in the program code, e.g., method parameters, etc. In contrast, ClickRelease policies are described in terms of app inputs and outputs.

Event Based Models and Declassification Vaughan and Chong [**?**] define expressive declassification policies that allow functions of secret information to be released after events occur, and extend the Jif compiler to infer events. ClickRelease instead ties events to user interactions.

Rafnsson et al. [**?**] investigate models, definitions, and enforcement techniques for secure information flow in interactive programs in a purely theoretical setting. Sabelfeld and Sands [**?**] survey approaches to secure declassification in a language-based setting. ClickRelease can be seen as addressing their "what" and "when" axes of declassification goals: users of Android apps interact with the GUI to control when information may be released, and the GUI is responsible for conveying to the user what information will be released.

8 Conclusion

We introduced interaction-based declassification policies, which describe *what* and *when* information can flow. Policies are defined using LTL formulae describing event traces, where events include GUI actions, secret inputs, and network sends. We formalized our policies using a trace-based model of apps based on security relevant events. Finally, we described ClickRelease, which uses symbolic execution to check interaction-based declassification policies on Android, and showed that ClickRelease correctly enforces policies on four apps, with one secure and two insecure variants each.

A Operational Semantics / Trace Generation

To illustrate how traces are generated, we introduce an operational semantics for a model of Android programs. Our semantics uses an abstract machine which transitions between states and reads messages on a message queue. A *state* is a tuple (M, σ, H) that includes a *message queue* M, which is a list of pairs of a channel and a primitive; a *heap* σ, which maps locations to values; and a *handler map* H, which maps channel names to functions installed to handle events on those channels. Each channel may have at most one handler. We write $\Sigma.X$ (where X could be M, σ, or H), to mean the X component of Σ. Similarly, we write $\Sigma[X \mapsto X']$ to mean Σ with the X component replaced by X'.

As programs execute, they produce a *trace* t of *events* η. Events are writes, written $name!p$, of primitive p from channel $name$. We also include an empty event τ, which is the identity of trace concatenation.

Our semantics is stratified into two levels: a big-step semantics, shown at the top of Fig. **??**, which models evaluation of code in a handler, and a small-step semantics, shown at the bottom of the figure, which models the message queue. The semantics generates sets of traces containing input and output messages.

$\boxed{e, \Sigma_1 \Downarrow v, \Sigma_2}$

$$\textsc{RVal}\ \frac{}{v, \Sigma \Downarrow v, \Sigma} \qquad \textsc{RApp}\ \frac{e_1, \Sigma_1 \Downarrow (\lambda x.e_3), \Sigma_2 \qquad e_2, \Sigma_2 \Downarrow v_1, \Sigma_3 \qquad e_3\{x \mapsto v_1\}, \Sigma_3 \Downarrow v_2, \Sigma_4}{e_1\ e_2, \Sigma_1 \Downarrow v_2, \Sigma_4}$$

$$\textsc{RRef}\ \frac{e, \Sigma_1 \Downarrow v, \Sigma_2 \qquad \ell \notin dom(\Sigma_2.\sigma) \qquad \sigma' = (\Sigma_2.\sigma)[\ell \mapsto v]}{\mathsf{ref}\ e, \Sigma \Downarrow \ell, \Sigma_2[\sigma \mapsto \sigma']}$$

$$\textsc{RAssign}\ \frac{e_1, \Sigma_1 \Downarrow \ell, \Sigma_2 \qquad \ell \in dom(\Sigma_2.\sigma) \qquad e_2, \Sigma_2 \Downarrow v, \Sigma_3 \qquad \sigma' = (\Sigma_3.\sigma)[\ell \mapsto v]}{e_1 := e_2, \Sigma_1 \Downarrow v, \Sigma_3[\sigma \mapsto \sigma']}$$

$$\textsc{RDeref}\ \frac{e, \Sigma_1 \Downarrow \ell, \Sigma_2 \qquad (\Sigma_2.\sigma)(\ell) = v}{!e, \Sigma_1 \Downarrow v, \Sigma_2}$$

$$\textsc{RIfTrue}\ \frac{e_1, \Sigma_1 \Downarrow \mathsf{true}, \Sigma_2 \qquad e_2, \Sigma_2 \Downarrow v, \Sigma_3}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3, \Sigma_1 \Downarrow v, \Sigma_3} \qquad \textsc{RIfFalse}\ \frac{e_1, \Sigma_1 \Downarrow \mathsf{false}, \Sigma_2 \qquad e_3, \Sigma_2 \Downarrow v, \Sigma_3}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3, \Sigma_1 \Downarrow v, \Sigma_3}$$

$$\textsc{ROp}\ \frac{e_1, \Sigma_1 \Downarrow v_1, \Sigma_2 \qquad e_2, \Sigma_2 \Downarrow v_2, \Sigma_3}{e_1 \oplus e_2, \Sigma_1 \Downarrow v_1 \oplus v_2, \Sigma_3} \qquad \textsc{RCstr}\ \frac{e_i, \Sigma_i \Downarrow v_i, \Sigma_{i+1} \qquad i \in 1..n}{f(e_1, \ldots, e_n), \Sigma_1 \Downarrow f(v_1, \ldots, v_n), \Sigma_{i+1}}$$

$$\textsc{RInst}\ \frac{e_1, \Sigma_1 \Downarrow (\lambda x.e_2), \Sigma_2 \qquad H' = (\Sigma_2.H)[name \mapsto \lambda x.e_2]}{\mathsf{install}\ name\ e_1, \Sigma_1 \Downarrow \mathsf{unit}, \Sigma_2[H \mapsto H']}$$

$$\textsc{RProj}\ \frac{e, \Sigma_1 \Downarrow f(v_1, \ldots, v_n), \Sigma_2}{f^{-i}(e), \Sigma_1 \Downarrow v_i, \Sigma_2} \qquad \textsc{RSend}\ \frac{e, \Sigma_1 \Downarrow p, \Sigma_2 \qquad M' = (\Sigma_2.M)@(name, p)}{\mathsf{send}\ name\ e, \Sigma_1 \Downarrow \mathsf{unit}, \Sigma_2[M \mapsto M']}$$

$\boxed{\Sigma_1 \rightarrow^{\eta} \Sigma_2 \ \text{ and } \ \vdash e \rightsquigarrow t}$

$$\textsc{THandle}\ \frac{\Sigma_1.M = (name, p)@M' \qquad \Sigma_1.H(name) = \lambda x.e \qquad \Sigma' = \Sigma_1[M \mapsto M'] \qquad e\{x \mapsto p\}, \Sigma' \Downarrow v, \Sigma_2}{\Sigma_1 \rightarrow^{\tau} \Sigma_2} \qquad \textsc{TInput}\ \frac{\Sigma' = \Sigma[M \mapsto (\Sigma.M)@(name, p)]}{\Sigma \rightarrow^{name!p} \Sigma'}$$

$$\textsc{TOutput}\ \frac{\Sigma.M = (\mathsf{netout}, p), M' \qquad \Sigma' = \Sigma[M \mapsto M']}{\Sigma \rightarrow^{\mathsf{netout}!p} \Sigma'} \qquad \textsc{TProg}\ \frac{\Sigma_0 = ([(\mathsf{onCreate}, \mathsf{unit})], \emptyset, \{\mathsf{onCreate} \mapsto \lambda x.e\}) \quad x \notin FV(e) \qquad \Sigma_i \rightarrow^{\eta_i} \Sigma_{i+1} \quad i \in [0..n]}{\vdash e \rightsquigarrow \eta_0 \cdot \eta_1 \cdots \eta_n}$$

Fig. 6. Semantics for our Android subset.

The big-step semantics proves judgments of the form $e, \Sigma_1 \Downarrow v, \Sigma_2$, meaning evaluation of expression e in state Σ_1 produces a value v and new state Σ_2.

The first several rules are standard. RVal evaluates a value to itself, without changing the state. RApp evaluates e_1 to a lambda, evaluates e_2 to a value, and then evaluates the body of the lambda with the actual argument substituted for the formal variable: we use the notation $e\{x \mapsto v\}$ for e when unbound occurrences of x in e have been syntactically replaced with v. RRef evaluates e to a value v, finds a fresh location ℓ in the heap, and then evaluates to ℓ, returning a state where the heap maps ℓ to v. RAssign evaluates e_1 to a location ℓ and then updates the contents of ℓ. RDeref evaluates e to a location and returns the contents of that location.

RIfTrue evaluates e_1 to a value and if it evaluates to true, evaluates e_2. RIfFalse is analogous. ROp evaluates a binary operation by applying the designated operation to the values of the two subexpressions.

RCstr and RProj construct terms and project from constructed terms. Finally, RInst evaluates e_1 to a lambda and adds it as a handler for channel *name*; the result value is the unit value unit (a nullary constructor). RSend evaluates e to a primitive p, and then adds the message $(name, p)$ to the end of the message queue. Here we use @ for concatenation. Note that RSend only allows primitives to be sent, and not locations or lambda expressions.

The first three rules in small-step semantics at the bottom of Fig. **??** prove judgments of the form $\Sigma_1 \rightarrow^{\eta} \Sigma_2$, meaning the machine can take a step from state Σ_1 to a new state Σ_2, producing an event η. THandle consumes a message $(name, p)$ from the front of the message queue (recall RSend adds a message to the *end* of the message queue), looks up the handler for *name*, and then invokes the handler, passing p as its argument. Running a handler is not an externally visible operation, which is indicated by an empty event τ on the reduction arrow.

TInput models an input message, which may be due to user input (e.g., GUI clicks) or secret input from the system (e.g., a callback with updated location information). This rule non-deterministically picks some channel *name* and an arbitrary primitive p, and then sends p on that channel. The input is recorded as an event on the reduction arrow. Notice that here we do not distinguish the security level of an input—we choose to leave that up to the security policy designer, who can opt to either always designate GUI inputs as low-security (as we do in our experiments) or make them high-security.

TOutput models writes to the network, consuming a message from a distinguished channel netout. Writing to this channel corresponds to the InfoSender.sendInt calls in Section **??**. Since these messages may be seen by the observer—i.e., they are "low visible" outputs—we record the write event on the reduction arrow. Note that by convention there is no user handler for this channel.

Finally, TProg proves a judgment of the form $\vdash e \rightsquigarrow t$, meaning running the program e produces a trace t of events. This rule creates an initial state Σ_0 in which e is bound as a handler on a special onCreate channel, and the message queue contains an initial message on that channel. The rule then repeatedly steps to the next state $n + 1$ times. It produces the event trace $\eta_0 \cdots \eta_n$. Notice that the length of the trace n is nondeterministic; in general, since these are reactive

$$
\begin{array}{rcl}
t, i \models \mathit{name}!p_1 & \iff & t[i] = \mathit{name}!p_1 \\
t, i \models \mathit{name}!* & \iff & \text{there exists } p, t[i] = \mathit{name}!p_1 \\
t, i \models p_1 \oplus p_2 & \iff & \models p_1 \oplus p_2 \\
t, i \models \neg\phi & \iff & t, i \not\models \phi \\
t, i \models \phi \wedge \psi & \iff & t, i \models \phi \text{ and } t, i \models \psi \\
t, i \models \phi \vee \psi & \iff & t, i \models \phi \text{ or } t, i \models \psi \\
t, i \models \phi \rightarrow \psi & \iff & t, i \models \psi \text{ or } t, i \models \neg\phi \\
t, i \models \mathcal{G}\phi & \iff & \text{for all } j \;.\; j \geq i \Rightarrow (t, j \models \phi) \\
t, i \models \mathcal{F}\phi & \iff & \text{there exists } j, j \geq i \Rightarrow (t, j \models \phi) \\
t, i \models \mathcal{P}\phi & \iff & \text{there exists } j, j \leq i \Rightarrow (t, j \models \phi) \\
t, i \models \phi \,\mathcal{U}\, \psi & \iff & \text{there exists } j, j \geq i \Rightarrow ((t, j \models \psi) \text{ and} \\
 & & \text{for all } k, i \leq k < j \Rightarrow (t, k \models \phi)) \\
t, i \models \phi \,\mathcal{S}\, \psi & \iff & \text{there exists } j, j \leq i \Rightarrow ((t, j \models \psi) \text{ and} \\
 & & \text{for all } k, j < k \leq i \Rightarrow (t, k \models \phi)) \\
t, i \models \forall x.\phi & \iff & \text{for all } p, (t, i \models \{x \mapsto p\}\phi) \\
t, i \models \exists x.\phi & \iff & \text{there exists } p, (t, i \models \{x \mapsto p\}\phi)
\end{array}
$$

Fig. 7. Models relation for LTL used in the paper

programs, they can usually run for an any number of steps as long as additional input arrives.

From the set of all executions of a program we can extract a set of traces, which we later use to define noninterference.

Definition 2 (Program Traces). *We define the set of traces of a program e as*

$$\mathit{traces}(e) = \{t \mid \vdash e \rightsquigarrow t\}$$

B Linear Temporal Logic

Figure **??** gives the definition of the models relaion for LTL as used in **??**. Note that we assume an interpretation $\models p_1 \oplus p_2$ for the atomic propositions of the form $p_1 \oplus p_2$.