Contextual Policy Enforcement in Android Applications with Permission Event Graphs

Kevin Zhijie Chen[†], Noah Johnson[†], Vijay D'Silva[†], Shuaifu Dai[†], Kyle MacNamara[†], Tom Magrino[†], Edward Wu[†], Martin Rinard[‡], and Dawn Song[†]

[†]University of California, Berkeley[*]
[‡]Massachusetts Institute of Technology

Abstract

The difference between a malicious and a benign Android application can often be characterised by context and sequence in which certain permissions and APIs are used. We present a new technique for checking temporal properties of the interaction between an application and the Android event system. Our tool can automatically detect sensitive operations being performed without the user's consent, such as recording audio after the stop button is pressed, or accessing an address book in the background. Our work centres around a new abstraction of Android applications, called a Permission Event Graph, which we construct with static analysis, and query using model checking. We evaluate application-independent properties on 152 *malicious and* 117 *benign applications, and application-specific properties on* 8 *benign and* 9 *malicious applications. In both cases, we can detect, or prove the absence of malicious behaviour beyond the reach of existing techniques.*

1 Introduction

Users of smartphones download and install software from application markets. According to the Google I/O keynote in 2012, by June 2012, the official market for Android applications, Google Play, hosted over 600,000 applications, which had been installed over 20 billion times. Despite recent advances in mobile security, there are examples of malware that cannot be detected by existing techniques.

A malicious application can compromise a user's security in several ways. Examples include leaking phone identifiers, exfiltrating the contents of an address book, or audio and video eavesdropping. Consult recent surveys for more examples of malicious behaviour [12, 14, 19].

In this paper, we focus on detecting malicious behaviour that can be characterised by the temporal order in which an application uses APIs and permissions. Consider a malicious audio application and which eavesdrops on the user by recording audio after the stop-button has been pressed, and a benign one. Both applications use the same permissions, and start and stop recording in response to button clicks. Malware detection based on syntactic or statistical patterns of control-flow or permission requests cannot distinguish between these two applications [14, 15, 18]. The difference between the two applications is semantic and requires semantics-based analysis.

The intuition behind our work is that user expectations and malicious intent can be expressed by the context in which APIs and permissions are used at runtime. A user expects that clicking a start or stop button, will respectively, start or stop recording, and further, that this is only way an audio application records. This expectation can be encoded by two API usage policies. The API to start recording audio should be called if and only if the event handler for the start button was previously called. The API to stop recording should be called if and only if the event handler for the stop button was previously called. Policies requiring that sensitive resources are not accessed by background tasks or in response to timer events can also aid in distinguishing benign from malicious behaviour.

We present Pegasus, a system for specifying and automatically enforcing policies concerning API and permission use. Pegasus combines static analysis, model checking, and

[*]The 4-7th authors are now at Peking Univerisity, UC Santa Barbara, Cornell University and University of Washington, Seattle, respectively. The work was done when the authors were in UC Berkeley.

runtime monitoring. We anticipate several applications of such technology. One is automatic, semantics-based screening for malware. Another is as a diagnostic tool that security analysts can use to dissect potentially malicious applications. A third is to provide fine-grained information about permission use to enable users to make an informed decision about whether to install an application.

Our system can be attacked by malware writers who obfuscate their applications to avoid static detection. However, such obfuscation will trigger our runtime checks and lead to convoluted code structures, which can be detected syntactically. Thus an attempt to evade our system and may only result in drawing greater scrutiny to the application.

1.1 Problem and Approach

We now describe the challenges behind policy specification and checking in greater detail, and the insights behind our solution.

Problem Definition. We consider three closely related problems. The first problem is to design a language for specifying the event-driven behaviour of an Android application. The second problem is to construct an abstraction of the interaction between an Android application and the Android event system. The third problem is to check whether this abstraction satisfies a given policy. A solution to these problems would allow us to specify security policies and detect (or prove the absence of) certain malicious behaviour.

Challenges. Property specification mechanisms typically focus on an application. Executing a task in the background, or calling an API after a button is clicked, are properties of the Android event system, not the application. Specifying policies governing event-driven API use requires a language that can describe properties of an application as well as of the operating system. For example, specifying that audio should not be recorded after a stop button is clicked, requires us to describe an application artefact, such as a button, a system artefact, such as a recording API, and the interaction between the two.

Checking policies of the form above is a greater challenge. Software model checking is a powerful technique for checking temporal properties of programs. Software model checkers construct abstractions of a program and then check properties using flow-sensitive analysis. The execution of an Android application is the result of intricate interplay between the application code and the Android system orchestrated by callbacks and listeners. Constructing an abstraction of such behaviour is difficult because control-flow to event-handlers is invisible in the code. Moreover, static analysis of event-driven programs has received little attention, and was recently shown to be EXPSPACE-hard [25]. The first analysis challenge is to model control-flow between the event system and application.

The second analysis challenge is to design and compute an abstraction that can represent event-driven behaviour, but is small compared to the Android system. Most existing techniques abstract data values in a program, but our focus on the Android event system mandates a new abstraction. The challenge in computing an abstraction lies in modelling the Android event system, and dealing with complex heap manipulation in Android programs, and their use of reflection, and APIs from the Java and Android SDK.

Insights. We overcome the aforementioned challenges using the insights described next. Our first insight is that though the Android system is a large, complicated object, it changes the state of the application using a fixed set of event handlers. It suffices for a policy language to express event handlers, APIs, and certain arguments to APIs to specify the context in which an application uses permissions.

Even a restricted analysis of the Android event system or event handlers defined in an application is not feasible due to the size of the code and the state-space explosion problem. Our second insight is to use a graph to make the interaction between an application and the system explicit. We introduce *Permission Event Graphs* (PEGs), a new representation that abstracts the interplay between the Android event system, and permissions and APIs in an application, but excludes low-level constructs.

Our third insight is that a PEG can be viewed as a predicate abstraction of an Android application and the Android system, where predicates describe which events can fire next. Standard predicate abstraction engines use theorem provers to compute how program statements transform predicates over data. We implement a new, Android specific, *event semantics engine*, which can compute how API calls transform predicates over the Android event queue.

The final challenge, once an abstraction has been constructed is to check that it satisfies a given policy. We use standard model checking algorithms for this purpose. Detecting sequences or repeating patterns in an application can be implemented using basic graph-theoretic algorithms for reachability and loop detection.

Our experience suggests that PEGs reside in a sweet-spot in the precision-efficiency spectrum. Our analysis based on PEGs is more precise than existing syntactic analyses and is more expensive to construct. However, we gain efficiency because a single PEG can be queried to check several policies pertaining to a single application.

1.2 Content and Contributions

In this paper, we study the problem of detecting malicious behaviour that manifests via patterns of interaction between an application and the Android system. We design a new abstraction of the context in which event-handlers fire, and present a system for specifying, computing and

checking properties of this abstraction. We make the following contributions:

1. **Permission Event Graphs:** A novel abstraction of the context in which events fire, and event-driven manner in which an Android application uses permissions.
2. **Encoding user and malicious intent:** We encode user expectations and malicious behaviour as temporal properties of PEGs.
3. **PEG construction:** We devise a static analysis algorithm to construct PEGs. The algorithm computes a fixed point involving transformers, generated by the program, the event mechanism, and APIs. Our event model supports 63 different event handling methods in 21 Android SDK classes.
4. **PEG analysis:** We implement Pegasus, an automated analysis tool that takes as input a property, and checks if the application satisfies that property.
5. **Experiments:** We check 6 application-independent properties of 269 applications, and check application-specific properties of 17 applications. Pegasus can automatically identify malicious behaviour, which was previously discovered by manual analysis.

The paper is organised as follows: We summarise background on Android and introduce our running example in Section 2. PEGs are formally defined in Section 3 and can be constructed using the algorithm in Section 4. The details of our system appear in Section 5, followed by our evaluation in Section 6. We conclude in Section 8 after discussing related work in Section 7.

2 Background and Overview

In this section, we give an overview of the Android platform as relevant for this paper and illustrate PEGs with a running example.

2.1 Android

Android is a computing platform for mobile devices. It includes a multi-user operating system based on Linux, middleware, and a set of core applications. Users install third-party applications acquired from application markets. An *Android package* is an archive (`.apk` file) containing application code, data, and resource information.

Applications are typically written in Java but may also include native code. Applications compile into a custom *Dalvik executable format* (`.dex`), which is executed by the Dalvik virtual machine.

Permissions. A *permission* allows an application to access APIs, code and data on a phone. Permissions are required to access the user's contacts, SMS messages, the SD card, camera, microphone, Bluetooth, and other parts of the phone. All permissions required by an application must be granted by a user at install time.

The Manifest. Every application has a *manifest file* (`AndroidManifest.xml`) describing the application's requirements and constituents. The manifest contains component information, the permissions required, and Android API version requirements. The component information lists the components in an application and names the classes implementing these components.

Components. The building blocks of Android applications are *components*. A component is one of four types: activity, service, content provider, and broadcast receiver, each implemented as a subclass of `Activity`, `Service`, `ContentProvider`, and `BroadcastReceiver`, respectively. An *activity* is a user-oriented task (such as a user interface), a *service* runs in the background. a *content provider* encapsulates application data, and a *broadcast receiver* responds to broadcasts from the Android system. Components (consequently, applications) interact using typed messages called *intents*.

Lifecycles. A lifecycle is a pre-defined pattern governing the order in which the Android system calls certain methods. An application can define callbacks and listeners that contribute to the lifecycle.

An activity is started using the `startActivity` or `startActivityForResult` API calls. During execution, an activity may be *running*, meaning it is visible and has focus, *paused*, if it is visible but not in focus, or *stopped* if it is not visible. Application execution usually begins in an activity. A service may be *started* or *bound*. A service is started if a component calls `startService`, following which the service runs indefinitely, even if the component invoking it dies. The `bindService` call allows components to bind to a service. A bound service is destroyed when all components bound to it terminate.

Events and APIs. Events and APIs are the two ways an Android application interacts with the system. We define an *event* as a situation in which the Android system calls application code. Examples of events are taps, swipes, SMS notifications, and lifecycle events. The code that is called when an event occurs is called an *event handler*. We define an API to be a system defined function, which applications can call. In this paper, we are concerned with event and permission APIs. An *event* API is one that changes how events are handled, such as registering a `Button.onClick` listener, or making a button invisible.

2.2 Overview and Running Example

We now demonstrate the concepts in this paper with a running example, as well as how we envision the system being used. Consider a malicious audio recording applica-

Figure 1. User interface for the running example. The application records audio in a background service after the user has clicked the stop button.

tion, which eavesdrops on the user. On startup, the application displays the interface shown in Figure 1. This interface is implemented as a `Recorder` *activity* and contains two buttons, REC and STOP.

Initially, only REC is clickable. Clicking REC initiates recording, makes STOP clickable, and disables REC from being clicked. Clicking STOP terminates recording, enables REC, and disables STOP. When the application is started, it registers a service, which creates a system timer callback, which is invoked every 15 minutes. The callback function records 3 minutes of audio and stores it on the SD card. Since services run in the background, this application will eavesdrop even after the recorder application is closed.

We now consider two problems: How can we precisely define malicious behaviour such as surreptitious recording? How can we automatically detect such behaviour?

Defining Malicious Intent. Rather than define malicious intent, we focus on defining user intent, or user expectations. In our example, the details of *how* recording happens is determined by the developer, but a user expects to be defining *when* recording happens. Moreover, the user expects that clicking REC will start recording, that clicking STOP will stop recording, and that this is the only situation in which recording occurs. This expectation contains a logical component and a temporal component, and can be formally expressed by a temporal logic formula.

$$(\neg \text{Start-Recording } \mathbf{U} \text{ REC.onClick}) \\ \wedge (\text{Stop-Recording} \iff \text{STOP.onClick})$$

This formula, in English, asserts that the *proposition* Start-Recording does not become true until the proposition REC.onClick is true, and that Stop-Recording is true if and only if STOP.onClick is true. Such a formula is interpreted over an execution trace. REC.onClick and STOP.onClick are true at the respective instants in a trace when the eponymous buttons are clicked. The propositions Start-Recording and Stop-Recording are true in the respective instants when the APIs to start and stop recording are called.

A second example of user expectation is that an SMS is not sent unless the user performs an action, such as clicking a button. A third example is that when an SMS arrives,

Figure 2. Permissions requested by the recording application during installation.

the user is notified. These properties can be expressed by the two formula below. The second formula expresses that a broadcast message (such as an SMS notification) is not aborted by the application.

$$\neg \text{Send-SMS } \mathbf{U} \text{ Button.onClick}$$

$$\neg \text{BroadcastAbort}$$

The three formulae above fall into two different categories. The SMS and broadcast properties are application independent. They can be checked against all applications, and are part of a cookbook of generic properties we have developed. The properties about recording are application specific and have to be written by the analyst.

The set of propositions is defined by our tool, and includes permissions, API calls, certain event handlers, and constant arguments to API calls. To aid the analyst, we have implemented a tool that extracts from an application's manifest, the names and types of user interface entities such as buttons and widgets, and their relevant event-handlers.

We express user intent with formulae. We say that an application exhibits *potentially malicious intent* if it does not satisfy a user intent formula. Our tool Pegasus automatically checks if an application satisfies a formula. If an application violates a property, Pegasus provides diagnostic information about why the property fails. The analyst has to decide if failure to respect user intent is indeed malicious. We discuss this issue in greater detail later.

Detecting Potentially Malicious Intent. How can we determine if an Android application respects a formula specifying user intent? Figure 2 depicts the permissions requested by the recorder during installation. Techniques that only examine permission requests [1, 18, 20] will only

know that the application uses audio and SD card permissions. Since control-flow between the Android system and event-handlers is not represented in a call graph, structural analysis of call graphs [12, 23], will not identify the behaviours discussed above.

The challenge in checking temporal properties is to construct an abstraction satisfying two requirements: It must be small enough for model checking to be tractable. It must be large enough to avoid generating a large number of false positives. Permissions used by an application, call graphs, and control flow graphs can be viewed as abstractions that can be efficiently analysed but do not satisfy the second requirement. We now describe an abstraction that enriches permission sets and call graphs with information about event contexts.

Permission Event Graphs. We have devised a new abstraction called a Permission Event Graph (PEG). In a PEG, every vertex represents an event context and edges represent the event-handlers that may fire in that context. Edges also capture the APIs and permissions that are used when an event-handler fires. Since permissions such as those for accessing contact lists, are determined by APIs calls and the argument values, knowledge of APIs does not subsume permissions. Example information that a PEG can represent is that clicking a specific button causes the READ_CONTACTS permission to be used, while the ACCESS_FINE_LOCATION permission is used in a background task.

A portion of the PEG for the running example is shown in Figure 3. Every vertex represents an event context. There are two types of edges. Solid edges represent synchronous behaviour, and dashed edges represent asynchronous behaviour. We refer to the firing of one or more event handlers as an *event* and the use of one or more APIs and permissions as an *action*. An edge label $\frac{E}{A}$ represents that when the event E occurs, the action A is performed.

Figure 3 shows that when the event-handler REC.onClick is called, the action denoted Start-Recording occurs. This action represents calling an API to start recording. We have omitted portions of the PEG related to the activity initialisation, destruction, and the service lifecyle. Next, the event STOP.onClick is enabled, and when it occurs, causes the Stop-Recording action. The dashed edge from onResume indicates an asynchronous call to start a service.

The PEG captures semantic information about an application that is not computed by existing techniques. For example, we see that there are two distinct contexts in which the audio is recorded. We also see that recording stops if we click STOP, but this is not the only way to stop recording.

Examining the PEG reveals that the application records audio even if REC is not clicked. Moreover, we can determine the sequence of events leading to this malicious behaviour: a new service is started, a timer is then created, and timer events start recording. PEGs generated in practice are too large to examine manually. In such cases, specifications can be treated as queries about the application, and model checking can be used to answer such queries.

Security Analysis with PEGs. The techniques we develop have several uses. All the uses follow the workflow of starting with a set of properties, automatically constructing a PEG for an application and model checking the PEG, manually examining the results of model checking, and repeating this process if required.

There are several kinds of properties that an analyst can check. We have developed a cookbook of application-independent properties, such as background audio or video recording. An analyst can write application-specific properties to check that an application functions as expected. For example, clicking REC should start recording, and STOP should stop recording. An analyst can also pose questions about the behaviour of specific event-handlers: Does clicking the STOP button stop recording? If the application is sent to the background, will recording continue or stop? If the application is killed while recording, will the data be saved to the SD card? All these questions can be encoded as temporal properties.

Our tool Pegasus can be used to automatically construct the PEG for an application and model check the PEG. If a property is satisfied, the analyst will have to check if it was too general, and try a more specific property. If a property is not satisfied, the model checker will generate a counterexample trace: a sequence of events and actions violating the property. The analyst has to examine the trace and see if it is symptomatic of malicious behaviour. If the behaviour is potentially malicious, the analyst will have to reproduce it at runtime. If the behaviour is benign, the analyst will have to strengthen the property that is checked to narrow the search for malicious behaviour.

To summarise, a vocabulary based on events and actions allows for describing a new family of benign and malicious behaviour beyond the reach of existing specification mechanisms. Events are a runtime manifestation of user interaction with an application, and actions describe an application's response. Specifications involving events and actions allow us to encode user intent in mechanical terms. From a user's perspective, a PEG summarises the dialogue between a user (via events) and an application. From an algorithmic perspective, a PEG is a data-structure encoding the interaction between the Android event system (via calls to event-handlers) and application code.

3 An Abstraction of Android Applications

The contributions of this section are a formal definition of PEGs, and a symbolic encoding of PEGs.

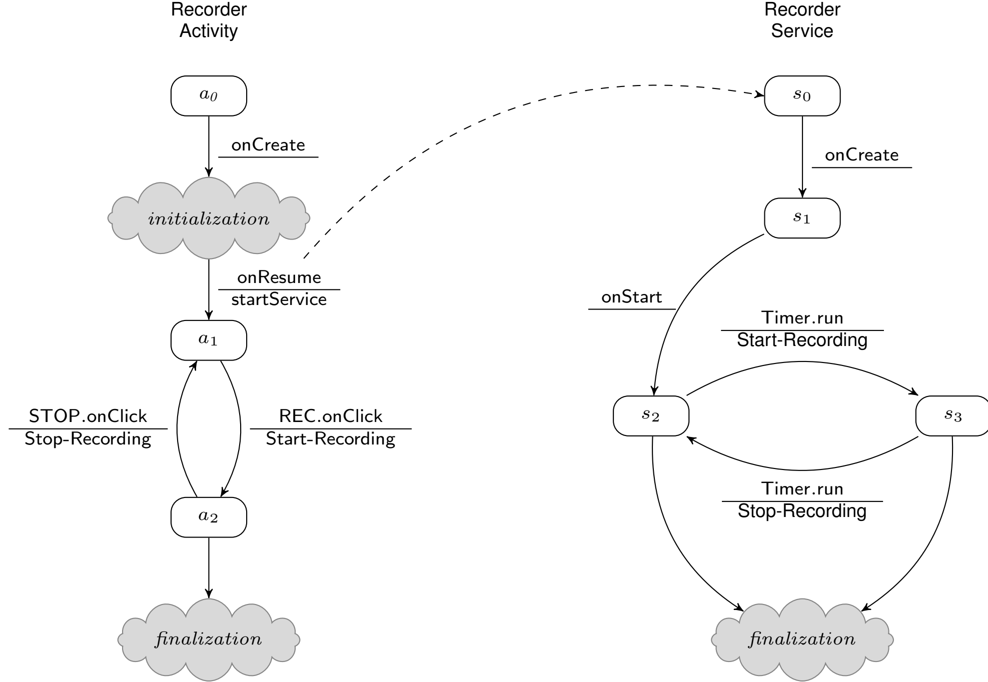

Figure 3. Permission Event Graph for the running example. Vertices represent event contexts and an edge label $\frac{E}{A}$ represents that when the event-handler E fires, an action A is performed. Dashed edge represent asynchronous tasks.

3.1 Transition Systems from Android Apps

An Android application defines an infinite-state transition system, which describes the runtime behaviour of an application. The transition system we define makes the control-flow and event-relationships in an application explicit. It is a mathematical object that we never construct, but it informs the design of our analysis.

States. A *runtime state*, or just state, is composed of an application state and a system state. The *application state* consists of the application program counter, a valuation of program variables, and the contents of the stack and the heap. The *system state* consists of the contents of the event queue, event handlers and listeners that are enabled, and other global system states. The set of states

$$State = App\text{-}States \times Sys\text{-}States$$

contains all combinations of application and system states. These sets include unreachable states. A state $\sigma = (p, s)$, consists of an application state p and a system state s. We denote the set of initial states of execution as $Init$.

Transitions. An application changes its internal state by executing statements, and changes the system state by making API calls. Conversely, the system may change its own state or change application state by calling event handlers. A *transition* is the smallest change in the state of an application or system, and

$$Trans \subseteq State \times State$$

is the *transition relation* of an application. A transition t is caused by executing a statement, denoted $stmt(t)$, in the application or system code. We call t is an *application transition* if $stmt(t)$ is in the application code and is a *system transition* otherwise.

Transition Systems. The evolution of an application and the Android system over time is mathematically described by a *transition system*

$$T = (State, Trans, Init, stmt)$$

consisting of a set of states $State$, a transition relation $Trans$, a set of initial states $Init$, and a function $stmt$ that labels transitions with statements.

Traces. We formalise the execution of the application in the system. A *trace* of T is a sequence of states $\pi = \pi_0, \pi_1, \ldots$ in which π_0 is an initial state and every pair (π_i, π_{i+1}) is a transition. We write $traces(\sigma)$ for the set of traces originating from σ, and write $traces(T)$ for the set of traces of T. A trace contains complete information about application and system transitions.

3.2 Permission Event Graphs

The transition system defined by an application is infinite and checking its properties is undecidable in general. The standard approach to addressing such undecidability is to construct an *abstraction* of this transition system. We now introduce *Permission Event Graphs* , a variation of transition systems, which can represent finite-state abstractions of Android applications.

Example 1. Revisit the PEG for the running example in Figure 3. A vertex in the figure is called an abstract state. The abstract state a_1 represents all possible runtime states in which the REC.onClick event-handler may be called in the recorder activity. An edge in the figure is an abstract transition. The abstract transition from a_1 to a_2 has a label representing that if the event-handler REC.onClick is called, the application will disable the REC, enable the STOP, start recording, and transition to the state a_2. The dashed edge is an *asynchronous transition*, representing that the action Start-Recording launches an asynchronous task. In this case, a service is started. $\triangleleft$

A formal definition of PEGs follows. We use the prefix "a" to indicate sets used as abstractions. We write $\mathcal{P}(S)$ for the set of all subsets of a set S.

We define an event to be a set of event handlers. For example, the event STOP.onClick represents a single event handler. We can also define an event onClick that corresponds to all event handlers which may be called when a button is clicked. Formally, let $Handler$ be a set of event handlers and $Event$ be a set of symbols, each representing one or more event-handlers.

Definition 1. A *Permission Event Graph* (PEG) over a set of event symbols $Event$ and APIs API is a tuple

$$PEG = (aState, aTrans, bTrans, aInit)$$

consisting of the following.

- A set of abstract states $aState$. Every abstract state represents a set of runtime states, which form the context of an event.
- A labelled transition relation $aTrans \subseteq aState \times Event \times \mathcal{P}(\mathsf{API}) \times aState$, where each transition (s_1, E, A, s_2), represents that in state s_1, the event E may fire, and causes the APIs in A to be called, leading to abstract state s_2.
- A relation $bTrans \subseteq \mathcal{P}(\mathsf{API}) \times aState$, where each tuple (A, s), represents that the action A causes an asynchronous transition to the abstract state s.
- A set $aInit$ of abstract initial states.

PEGs are different from control flow graphs, call graphs, and other standard graph-based abstractions of programs. A PEG is different from to a control flow graph because it does not represent the syntactic structure of source code. A PEG only contains calls to system APIs, rather than all calls, as in a call graph, but also includes the values of arguments, hence is related, but incomparable (mathematically) to a call graph. We use the word "Permission" in the name because permissions are determined by calls to APIs and their arguments. We use the word "Event" to emphasise that state transitions represent the effect of firing event handlers.

We use graph algorithms to analyse PEGs. To derive the PEGs of an application efficiently, our abstraction engine uses the symbolic encoding introduced next.

3.3 A Symbolic Encoding of PEGs

We now devise a compact encoding of PEGs. Our encoding uses Boolean variables to represent PEG states and labels to represent actions, and can be exponentially more succinct than representing a PEG as a labelled graph.

Mode Variables and Event-Modalities. We first encode PEG states using Boolean variables. Define a set $ModeVars$ of Boolean-valued *mode variables*. The *Boolean encoding* of an abstract state is a function

$$s : ModeVars \to \{\mathsf{true}, \mathsf{false}\}$$

that assigns truth values to mode variables. The number of mode variables we need is logarithmic in the number of states of a PEG.

A Boolean formula φ over $ModeVars$ represents the set of Boolean encodings that make φ true. Recall that a *literal* is a Boolean variable or its negation, and a *cube* is a conjunction of literals. Let $Cube$ be the set of cubes over mode variables in a subset of $ModeVars$. We only use cubes and not arbitrary Boolean formula over $ModeVars$ to represent sets of encodings because cubes can be efficiently manipulated, while arbitrary formula cannot. The same encoding choice is used in the SLAM project [2].

We encode abstract transitions using tuples called *event-modalities*. An event-modality is a tuple

$$(Pre, A, Post) \in Cube \times \mathcal{P}(\mathsf{API}) \times Cube$$

consisting of a *precondition* Pre, a set of API labels A, and a *postcondition* $Post$. An event-modality $(Pre, A, Post)$ represents the set of abstract transitions that begin in some abstract state represented by Pre, and transition to some abstract state represented by $Post$, while causing the action A. Notice that events are not part of an event-modality.

Event-modalities encode abstract states and actions. We also have to encode events. An *event-map* is a function

$$event\text{-}map : Handler \to Cube$$

that maps each event-handler to a cube representing the set of abstract states in which that event-handler may fire. A *symbolic encoding* of a PEG is a tuple

$$sPEG = (aInit, EventModality, event\text{-}map)$$

consisting of a cube $aInit$ representing initial states, a set $EventModality$ of event-modalities, and an event-map.

Example 2. Consider a Boolean variable $TimerEnabled$ and a label Start-Recording. The timer-related behaviour in Figure 3 can be encoded using the value true for $TimerEnabled$ to represent s_2 and the value false to represent s_3. The abstract transition from s_1 to s_2 is represented by the event-modality below.

$$(TimerEnabled, \{\text{Start-Recording}\}, \neg TimerEnabled)$$

If the precondition $TimerEnabled$ is true, the timer event TIMER.run is enabled. If the event fires, the event handler causes the application to transition to a state satisfying the postcondition $\neg TimerEnabled$. ◁

4 The Abstraction Engine

The contribution of this section is a procedure and architecture for constructing PEGs from Android applications. Our implementation of this procedure combines a model of the Android event system and APIs with fixed point iteration in a lattice to derive PEGs.

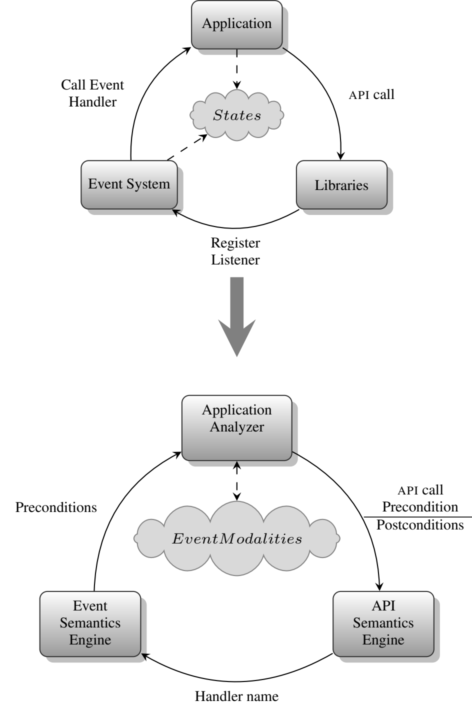

Figure 4. Intuition behind the abstraction engine. The system computes event modalities by combining a static analyser, which abstracts application semantics, an API semantics engine, which abstracts API calls, and an event semantics engine, which abstracts the event system.

4.1 The Core Algorithm

The interaction between an application, the event mechanism and libraries in an Android application is summarised in the upper part of Figure 4. The dashed arrows show that the state of an execution is modified either by executing application code or when the event system fires an event handler. The solid arrows denote calls. An application may call the Android APIs, and if the call is to register a listener, the APIs in turn access the event system.

The architecture we use to compute a symbolic PEG is shown in the lower part of Figure 4. Each shaded box represents an engine in our implementation. The different engines interact to compute a set of event-modalities. We abstract application code with a static analyser, model event generation and destruction with an *event semantics engine*, and model APIs with an API *semantics engine*.

Our static analyser determines a set of preconditions, which specify event contexts. When an API call is encountered, the precondition and API name are given to the API semantics engine. If the API modifies the application state, a postcondition is returned to the static analyser. If the API modifies the system state, the name of the API is given to the event semantics engine. The event semantics engine computes the set of preconditions for that event handler to fire, and the static analyser had to determine whether to analyse the event handler code. By iterating between these three engines, we derive a set of event-modalities that symbolically encode a PEG for an application.

The functions below formalise these components.

$$\begin{aligned}
entry &: Handler \to \mathcal{P}(\mathsf{API}) \\
next &: Handler \times \mathsf{API} \to \mathcal{P}(\mathsf{API}) \\
event\text{-}sem &: Handler \to \mathcal{P}(Cube) \\
api\text{-}sem &: \mathsf{API} \times Cube \to \mathcal{P}(Cube) \\
app\text{-}sem &: Handler \times Cube \\
&\qquad \to \mathcal{P}(\mathcal{P}(\mathsf{API}) \times Cube)
\end{aligned}$$

The function $entry$ takes as input an event-handler name, retrieves the code, constructs the CFG for the event-handler, and retrieves the first set of API calls reachable from the entry of the CFG, without calling other APIs. The function $next$ is similar to $entry$. When invoked as $next(h, A)$ on an event handler h with API call A, $next$ will return the set

C of APIs in h that may be called after calling A, such that there is no API call between A and each API in C. These functions are implemented in the static analyser, by combining control-flow reachability with pointer analysis.

The function *event-sem* takes as input an event handler and returns as output a set of cubes representing preconditions for that event handler to fire. This function is implemented by the event semantics engine.

The function *api-sem* takes as input an API call A and a precondition p and returns a set Q of postconditions. The postconditions satisfy that executing A in a state satisfying p leads to a state satisfying some cube in Q. This function is implemented by the API semantics engine.

The function *app-sem* takes as input an event handler and a precondition, and returns a set of pairs of the form (A, q). Let h be an event handler. Towards formally defining *app-sem*, we define a function

$$\begin{aligned} \mathit{reach\text{-}sem}_h : &\mathcal{P}(\mathcal{P}(\mathsf{API}) \times \mathsf{API} \times \mathit{Cube}) \\ &\rightarrow \mathcal{P}(\mathcal{P}(\mathsf{API}) \times \mathsf{API} \times \mathit{Cube}) \end{aligned}$$

that maps a tuple (A, a, p) representing a set of APIs A previous executed, and the API a that will be executed with precondition p to the tuple $(A \cup \{a\}, b, q)$, where b is an API that can be executed after a and q is the postcondition of executing a when p holds.

$$\begin{aligned} \mathit{reach\text{-}sem}_h(R) = R \cup \{(A \cup \{a\}, \{b\}, q) \ \mid \ &\text{where} \\ &(A, a, p) \text{ is in } \mathit{reach\text{-}sem}_h(R), \text{ and} \\ &b \text{ is in } \mathit{next}(h, a), \text{ and} \\ &q \text{ is in } \mathit{api\text{-}sem}(b, p)\} \end{aligned}$$

Note that $\mathit{reach\text{-}sem}_h$ occurs on both sides of the definition. The function is implemented by fixed point iteration over the CFG to compute a set of action, postcondition pairs. The function *app-sem* computes event modalities by computing $\mathit{reach\text{-}sem}_h$ and projecting out the action, postcondition pairs that reach the exit point of the event handler. Formally, *app-sem* satisfies the condition below.

$$\begin{aligned} \mathit{app\text{-}sem}(h, p) = \{(A, q) | q \in \mathit{Cube}, \ &\text{and} \\ A = &\bigcup_{(B,b,q) \in R} B \cup \{b\}, \\ &\text{where } R = \{(\emptyset, a, p) \ \mid \ a \in \mathit{entry}(h)\}, \\ &\text{and } \mathit{next}(h, b) = \emptyset\} \end{aligned}$$

A pair (A, q) is produced by $\mathit{app\text{-}sem}(h, p)$ exactly if A is a set of APIs reachable in h, and executing h in a state satisfying p leads to the postcondition q at the exit point of h. We now describe how the precondition p is generated.

Fixed Point. We combine the functions above to compute a fixed point whose result is the PEG for a given application:

$$\begin{aligned} \mathit{EventModality} = \{(p, A, q) \mid &\ p \in \mathit{event\text{-}sem}(h), \\ &\text{and } (A, q) \in \mathit{app\text{-}sem}(h, p), \\ &\text{and } h \in \mathit{Handler}\} \end{aligned}$$

In words, we consider each event handler h in *Handler*, use the event semantics engine to generate preconditions for h to fire, and then combine *app-sem* and *api-sem* to determine the postconditions derived by firing h. The implementation of each function above is discussed below.

4.2 Implementation of the Engines

A contribution we make, en route to computing PEGs, is to engineer a static analyser, an event semantics engine, and an API semantics engine. We discuss implementation details below.

Static Analysis. We implement a partially context-sensitive points-to analysis serving two purposes. First, the analysis overapproximates the targets of the method call. Overapproximation arises due to dynamic dispatch, where different executions of a given method call may invoke different methods. The second purpose is computing information about method arguments. For example, consider a call to `Button.setOnClickListener`. The first argument to this method is the event handler to attach as the `onClick` listener. We use the points-to analysis to disambiguate arguments and to overapproximate the set of event handlers the application will attach. Resolving arguments is necessary to derive sufficient information for verification, because an API call can map to different permissions depending on the values of its arguments. For example, a call to the `ContentResolver.query(URI)` method will access the phone's contacts if the URI points to the contacts content provider, while the same API will access the phone's SMS messages for a different URI. The two operations require different permissions.

We augment the context-insensitive analysis for event handlers by propagating the points-to information for method call parameters from the caller to the callee. This provides partial context-sensitivity. In particular, it allows the analysis of sub-functions to reason about values which are computed in parent functions. Our experience shows that this is important to handle, since many applications pass arguments for system APIs through helper functions or wrappers for those APIs. For flow-sensitivity, we use flow-insensitive analysis for class fields and flow-sensitive analysis for local variables to balance efficiency and precision. Our hybrid approach to points-to analysis is similar to the use of object representatives or instance keys [5, 21, 30].

Event Semantics Engine. The event semantics engine implements the *event-sem* function. It receives a method han-

Class name	Methods
android.app.Activity	onCreateOptionsMenu, onKeyDown, onOptionsItemSelected, onPrepareOptionsMenu, <init>, onActivityResult, onConfigurationChanged, onCreate, onCreateContextMenu, onDestroy, onPause, onRestart, onResume, onSaveInstanceState, onStart, onStop, onWindowFocusChanged
android.app.Dialog	<init>, onCreate
android.app.ListActivity	<init>, onCreate
android.app.Service	onBind, <init>, onCreate, onDestroy, onLowMemory, onStart, onStartCommand
android.content.BroadcastReceiver	<init>, onReceive
android.content.ContentProvider	query, insert, onCreate, delete, update, getType, <init>
android.content.ServiceConnection	onServiceConnected, onServiceDisconnected
android.os.AsyncTask	doInBackground, onPostExecute, onPreExecute
android.os.Handler	handleMessage
android.preference.PreferenceActivity	onPreferenceTreeClick, <init>, onCreate, onDestroy, onStop
android.preference.Preference.OnPreferenceChangeListener	onPreferenceChange
android.preference.Preference.OnPreferenceClickListener	onPreferenceClick
android.telephony.PhoneStateListener	onCallStateChanged
android.view.View.OnClickListener	onClick
android.view.View.OnTouchListener	onTouch
android.webkit.WebChromeClient	onProgressChanged
android.webkit.WebViewClient	shouldOverrideUrlLoading, onPageFinished, onPageStarted, onReceivedError
android.widget.AdapterView.OnItemClickListener	onItemClick
android.widget.AdapterView.OnItemLongClickListener	onItemLongClick
java.lang.Runnable	run, run
java.lang.Thread	run

Table 1. Event handling APIs supported by the event semantics engine.

dler name as input and returns the preconditions for the handler to execute. This engine models the semantics of events by capturing the context in which an event may fire. We implemented the engine by examining the effect of event handling mechanism as specified in the Android documentation and in the Android platform code. A list of 63 event handlers we model is given in Table 1.

API semantics engine. The *api-sem* receives as input a method call and a precondition, and generates as output the event-modalities generated by executing the method when that precondition is satisfied, and the postcondition in which the method terminates. Though these event-modalities are determined by the implementation of Android APIs, we *do not* analyse the Android API source code. Instead, we model every API call we have found necessary to support during analysis. Figure 5 summarises the API coverage of the API semantics engine. We support 1200 API calls, which covers over 90% of the call-sites we found on a data set of over 95,000 applications. The entire list of methods we support is too long to recall here.

5 Pegasus

We design and implement Pegasus, an analysis system that combines the abstraction procedure in Section 4 with analysis of PEGs and rewriting of Android applications.

System Overview. Figure 6 presents an overview of the Pegasus architecture. Pegasus takes as input an Android application and a specification expressed as safety property over events and actions. It uses a *translation tool* to convert Dalvik bytecode to Java bytecode. Using Java bytecode allows us to use off-the-shelf analysis frameworks.

The *abstraction engine* takes as input Java bytecode and generates an PEG as output. The PEG is fed to the *verification tool*, along with a specification to check for conformance. If certain application behaviour cannot be analysed (for instance, due to unresolved reflection), the *rewriting tool* generates a new application that contains dynamic checks when reflective calls are made.

If the PEG satisfies the specification, and the implementation of the API and event semantics engines, and the verification procedure is sound, the application is guaranteed to satisfy the specification as well. If the PEG does not satisfy the specification, it may be because the application violates the specification, or because the overapproximation creates false positives. For each violation, Pegasus produces a counterexample trace which can be used to determine if the violation corresponds to a feasible execution.

Specifications. Recall that *safety properties* assert that certain undesirable behaviours never occur. Researchers have

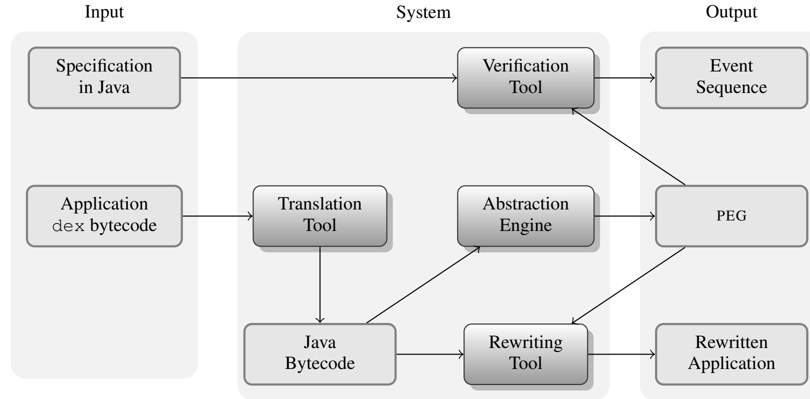

Figure 6. Pegasus architecture. The system consists of a translation and a verification tool, and an abstraction engine.

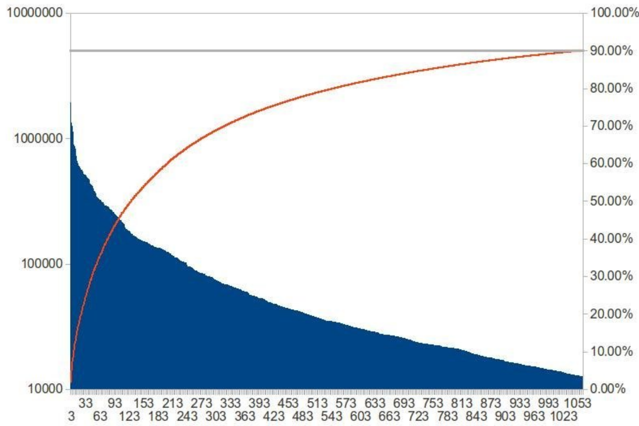

Figure 5. Coverage of API calls in the API semantics engine. API calls are represented by numbers on the x-axis. A vertical blue line represents the number of applications in which an API call occurs. The red line represents the cumulative distribution of API calls across call-sites. 1062 APIs make up for 90% of the calls in 95910 applications, and are part of those supported by our API semantics engine.

```
1  public class RunningExample implements SpecificationChecker {
2    // is the application currently recording?
3    private boolean isRecording = false;
4    // is the application allowed to record? (i.e., did the user
5    // press the Record button and not yet press the Stop button?)
6    private boolean recordingAllowed = false;
7
8    public boolean checkEvent(EventModality event) {
9      if (event.getEventHandler() ==
10           getClickHandlerForButtonByLabel(''REC''))
11       recordingAllowed = true;
12     else if (event.getEventHandler() ==
13           getClickHandlerForButtonByLabel(''STOP''))
14       recordingAllowed = false;
15
16     for (Action action : event.getActions()) {
17       if (action == RECORD_START)
18         isRecording = true;
19       else if (action == RECORD_STOP)
20         isRecording = false;
21     }
22
23     boolean violation = (isRecording && !recordingAllowed);
24     return violation;
25   }
26 }
```

Figure 7. Specification for the running example.

developed a numerous languages for safety properties. The SLAM project used a C-like specification language called SLIC [3], because it was convenient to use specification language with similar syntax to the analysed programs. Similarly, we write specification monitors in Java.

A specification for the running example is shown in Figure 7. The callback function `checkEvent` is used to determine if the current event modality corresponds to the button click handler for REC. If the current event modality corresponds to the REC button click event, the specification checker sets class field `recButtonClicked` to `true`. It then scans all the behaviours associated with the current event modality. If it finds the `RecordStart` behaviour and the record button has not been previously clicked, it signals a violation by returning `true`.

A user of our system implements specifications using the `SpecificationChecker` interface, which defines the callback function `checkEvent`. The verification algorithm calls this function for each event modality reached during exploration. Depending on the specification, the `checkEvent` function inspects the event type, the actions associated with the event, or both. The `checkEvent` returns

`true` if a violation has occurred based on the current event modality, and `false` otherwise. The specification checker can maintain a specification state in its class fields. The specification state is stored and restored by the verification tool using Java serialization.

To ease the task of writing specifications, we also implement a mapping from low-level API calls to high-level actions, such as maps from API calls to permissions [1, 18], and other security relevant actions, such as the start and the stop of recording. Pegasus enumerates the application's sequences of actions. It uses a Java interface to pass these sequences to the Java specification, which updates the state of the specification until a violation state is reached. If no sequence in a PEG leads to a violation, the application satisfies the specification.

Verification. Pegasus includes a verification algorithm that uses a bounded, breadth-first graph search with pruning, to check security properties written as Java checkers. Once the PEG has been generated, specifications can also be checked using other model checkers.

Rewriting. Our analysis is designed to successfully analyse many common-case uses of potentially problematic Java constructs such as reflection and dynamic invoke dispatching. The semantics of these constructs depends on information that is available only when the program executes, so static analyses may be unable to precisely analyse programs that use them. We rewriting applications to include runtime checks to account for cases where static analysis does not succeed.

When the abstraction engine fails to analyse part of an application, three strategies can be applied. The first one is to introduce a havoc statement and assume anything can happen. This strategy usually leads to a high false positive rate, and consequently, a tool that is not usable in practice.

The second strategy is to add runtime checks to only allow executions that respect the conditions computed by static analysis. For example, we can add runtime checks only allow a reflective call if the target call was already derived by static analysis. Static analysis may also fail to determine all the sensitive resources an application accesses. We can similarly add runtime checks to only permit accesses to URIs that were either statically determined, or are not considered sensitive, such as contact lists or SMSes.

We use the second strategy. In specific cases, where we have manually scrutinised an application, we permit executions even if they have not been analysed statically. This occurs when we believe the behaviour that was not statically analysed is benign. Such manually aided rewriting allows us to reduce the overhead of runtime checks.

In Section 6 we evaluate the number of unresolved transitions in the applications we analysed. The number is generally small, a fact we attribute to the simple coding patterns used by most applications (e.g., using a constant string as the argument to a reflection call), as well as our per-event context-sensitive analysis which allows us to propagate information through method calls within each event handler.

We do not support unknown native code. Known native code is modelled by the API semantics engine. Known dynamic class loading is supported by analysing the class and treating its loading point as a reflective call.

Implementation. Pegasus is implemented in 11,626 lines of Java code, including the code for the abstraction, model generation, and verification phases. The API semantics engine models 1218 APIs and the event semantics engine supports 62 different types of events. We developed a translation framework to translate Dalvik bytecode to Java bytecode; the dataflow analysis and rewriting are implemented in Soot [31], a compiler and static analysis framework for Java bytecode.

6 Evaluation

This section describes our experiments using Pegasus to demonstrate that PEGs can be used to automatically check and enforce policies in Android applications. All experiments are performed on an Intel Core i7 CPU machine with 4GB physical memory.

6.1 Generic Specification Checking

We run Pegasus on 152 malicious and 117 benign Android applications, and measure the execution and the size of the PEG. Figure 8 presents the Cumulative Distribution Function (CDF) of the time spent on PEG generation. On over 80% of the applications, the abstraction phase terminates within 600 seconds. The abstraction phase also always terminated within 2 hours. Figure 9 presents the CDF of PEG verification time, on a logarithmic scale. The verification phase terminates within 1000 seconds, for over 80% of the inputs, and the verification phase always terminates within 3.6 hours. To boost efficiency, we heuristically bounded verification to terminate after 50000 states were explored. The justification behind this heuristic is shown in Figure 10, where most of the applications have at most 10000 unique states, so the probability of an unsound result is low.

In the verification phase, we check 6 application-independent properties to determine if sensitive operations are guarded by user interaction. The three sensitive operations we consider are reading the GPS location, accessing the SD card, and sending SMSes. The properties we check are that the three behaviours above are always bracketed by user interaction, such as a button click. The result of verification is shown in Table 2. We see that malicious applications performing sensitive operations without user consent more frequently than benign applications.

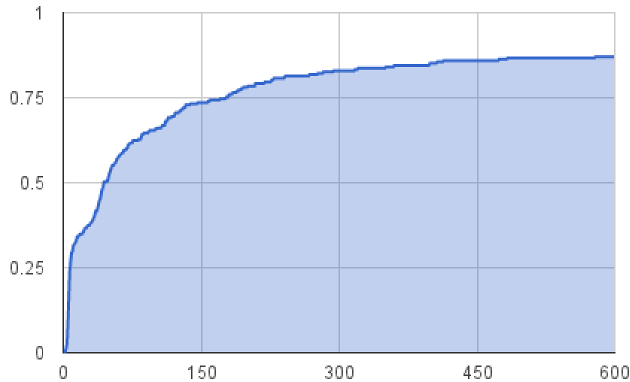

Figure 8. CDF of abstraction time.

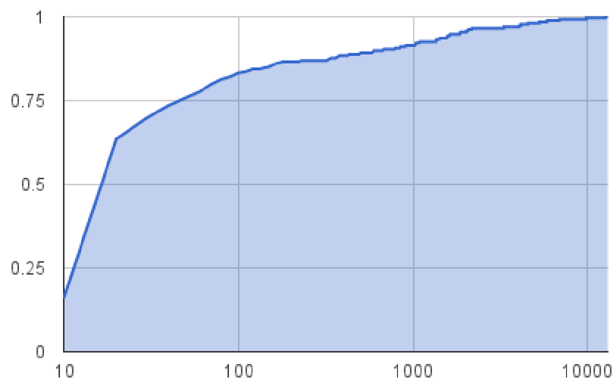

Figure 9. CDF of the verification time.

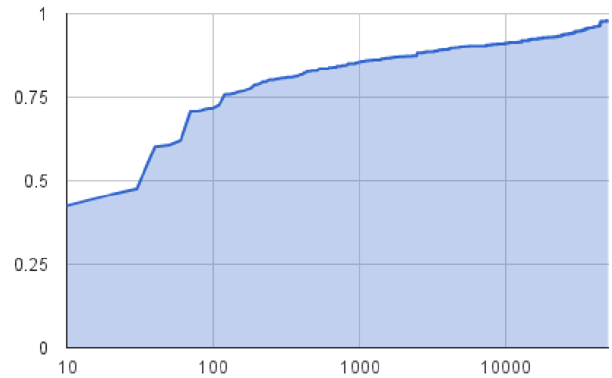

Figure 10. CDF of PEG size measured by the number of states.

Sensitive Operation	Malicious NoUI	Malicious Total	Benign NoUI	Benign Total
GPS	15	15	18	30
SD card	25	26	25	32
SMS	10	11	0	1

Table 2. Results of checking application-independent specifications. The first column is the name of the sensitive resource accessed. The "NoUI" columns list the number of applications accessing these resources without user consent.

6.2 Application-Specific Properties

Sample Applications. In this section, we check application-specific properties on 17 sample applications, including 8 benign applications and 9 applications with known malicious behaviours, to demonstrate Pegasus used as a diagnostic tool. Table 4 in Appendix A lists these sample applications and presents a short description for each application. The first 8 applications are benign samples selected from the official Android Market and third-party application stores. We selected these applications to represent a broad variety of different application classes that together exercise most of the core functionality supported in the Android system. These applications implement a variety of behaviours such as recording audio, accessing the phone's contacts, sending SMS messages, and accessing the device GPS location. The remaining 9 samples are malware which exhibit a variety of malicious behaviours.

Specifications. For these applications, we constructed application-specific properties after installing each application, reading its documentation to understand its intended functionality, looking at the list of events and GUI widgets used by the application, then determining a security policy which an analyst might reasonably wish to impose on the application. We wrote 23 application-specific properties.

PEG Generation. Table 3 summarises the results of PEG generation. The last two columns of Table 3, show how often the analysis can resolve the targets of intent calls and reflective calls. We also manually inspected the decompiled source code to resolve values that were not automatically determined, then used the rewriting tool to enforce those values at runtime.

Verification. We used Pegasus to check the generated PEGs for conformance to their properties. When Pegasus found a violation, we manually executed the applications and used the counterexample trace to determine if the violation represented a feasible behaviour of the application. If source code was available, we also inspected the code.

Our results show that Pegasus completes verification of most applications in less than a second, with a maximum verification time of 10 seconds. The length of the counterexample traces for violations ranges from 4 to 10 events.

Name	Size KB	EM #	Intents UR	Intents T	Reflection UR	Reflection T
Who's Calling?	148	83	0	2	0	0
Share Contacts	135	359	0	9	1	2
Geotag	117	226	0	19	1	2
Find My Phone	285	29	0	2	6	11
Simple Recorder	20	16	0	0	0	0
Diet SD Card	304	155	0	14	2	5
SMS Cleaner Free	159	175	0	14	0	3
SyncMyPix	425	300	4	12	1	3
SMS Replicator	63	18	0	0	0	0
ADSms	41	38	0	0	0	0
ZitMo	20	19	0	0	0	0
HippoSMS	404	434	0	38	0	0
DroidDream	204	89	12	15	0	0
Zsone	241	30	0	0	1	3
Geinimi	558	51	4	4	4	6
Spitmo	20	6	0	0	0	0
Malicious Recorder	20	25	0	0	0	0

Table 3. Summary of the evaluation results. The columns are: (1) name of application, (2) size of the .apk file, (3) number of event-modalities, (4) number of (unresolved) intents, (5) number of (unresolved) reflection calls.

The properties we checked are discussed in detail in Section 6.3. We checked 8 benign applications, against a total of 16 properties. For 11 of these properties, Pegasus determined that the application satisfied the property. For 3 of the remaining properties, we determined that Pegasus found a property violation due to legitimate but unexpected application behaviour. We believe that analysts would find such information valuable.

For 1 of the remaining 2 properties, Pegasus determined that an infeasible path involving dead code violated the property. For the last property, imprecision in the analysis caused Pegasus to determine that the application violated the property. Consequently, we conclude that Pegasus has false positives for 2 of the 16 properties.

For all 9 malicious applications, Pegasus correctly reported violations of at least one of the 7 properties. In other words, there were no false negatives for these properties.

6.3 Case Studies

We now present case studies illustrating properties one can check with Pegasus. Table 5 in Appendix A provides detailed information about the security actions and events used in the specifications in this section.

Specification Format. For brevity, we present specifications as LTL formulae. Note that Pegasus does not actually take LTL formulae as input but requires specifications to be encoded as a Java checker. A specification for the Find My Phone application is shown below.

$$\neg\text{Send-SMS}\ \mathbf{U}\ \text{Receive-SMS}$$

The symbol **U** is the *until operator*, while Send-SMS is an *action* and Receive-SMS is an *event*. The specification is read in English as asserting that

> The application does not send an SMS until it receives an SMS.

A important technical clarification is in order: the temporal logics supported by standard model checkers are usually *state-based*, meaning the propositions occurring in formulae describe properties of states. In our specification, mode-variables describe states and action labels describe properties of transitions. In technical temporal logic parlance, our specifications are both *state and event-based*. Consult [9] for an in-depth discussion of such issues.

6.3.1 Benign Applications

Simple Recorder. The simple recorder contains two buttons to start and stop recording, and behaves as expected. We check that the application only records audio when the REC is clicked, and stops when STOP is clicked. This property is expressed in LTL as

$$(\neg\text{Start-Recording}\ \mathbf{U}\ \text{REC.onClick}) \wedge (\text{Stop-Recording} \iff \text{STOP.onClick})$$

The application satisfies this property.

Diet SD Card. We check that the application accesses the SD card only after a button labelled Clean is clicked.

$$\neg\text{Access-SD}\ \mathbf{U}\ \text{Clean.onClick}$$

The application satisfies this property, showing that the SD card is only accessed after Clean is clicked.

Geotag. We check that the application accesses geolocation information only after the user clicks the Locate button.

$$\neg\text{Access-GPS}\ \mathbf{U}\ \text{Locate.onClick}$$

Pegasus discovers a property violation. On examining the counterexample, we determined that the main activity's `onCreate` event handler initialises the Google Ads library, which spawns a new background task and accesses the geolocation information. If we refine our property to

$$\neg\text{Access-GPS}\ \mathbf{U} \begin{pmatrix} \text{Locate.onClick} \\ \vee\ \text{AdTask.onPreExecute} \end{pmatrix}$$

Pegasus no longer reports a violation. We have also learnt that the only way the geolocation can be accessed without the user's consent is via the Google Ads library.

Who's Calling?. Similar to the Geotag application, we check the that contacts information is only accessed after a phone call is received.

$$\neg\text{Access-Contacts}\ \mathbf{U}\ \text{PhoneCall.onReceive}$$

Pegasus returns a counterexample which shows that the application retrieves and caches the contact list when it starts up. We verified this behaviour manually by running the application in an emulator and observing its API calls.

Share Contacts. We check whether the application accesses contacts if an SMS is not sent.

$$\neg\text{Send-SMS}\ \mathbf{U}\ \text{Access-Contacts}$$

This property holds. We then checked that SMSes are sent in response to user input.

$$\neg\text{Send-SMS}\ \mathbf{U}\ \text{Send.onClick}$$

This property also holds. Finally, we check if adding a new contact requires user consent.

$$\neg\text{Insert-Contacts}\ \mathbf{U}\ \text{Insert.onClick}$$

This property too is satisfied.

SMS Cleaner Free. This application allows users to delete SMS messages that match a user-provided contact name. We check if accessing contacts is user-driven.

$$\neg\text{Access-Contacts}\ \mathbf{U}\ \text{Select-Contact.onClick}$$

Pegasus reports a property violation. We manually investigated the counterexample generated and concluded that the counterexample was not feasible. The reported violation is a false positive. This application uses a switch statement to register the same event handler for different button clicks. Our abstraction engine does not consider branch conditions, so in the generated PEG every button click can trigger every event handler.

Find My Phone. This application responds to the receipt of an SMS containing a specific keyword by sending the phone's GPS location. We check if an SMS can be sent without any being received.

$$\neg\text{Send-SMS}\ \mathbf{U}\ \text{Receive-SMS}$$

Pegasus reports a violation. We manually verified that the violation was a false positive.

SyncMyPix. The SyncMyPix application specifies the target of an intent by looking at the configuration file and using a default value if the user does not specify the target. The set of possible runtime targets is not arbitrary, because they must be components in the application, but static analysis does not have this information and is inconclusive. We rewrite the application to force the target of the intent to be the default one. The rewriting takes 5 seconds and the rewritten application works correctly. We check if the rewritten application can access contacts without the Sync being clicked.

$$\neg\text{Access-Contacts}\ \mathbf{U}\ \text{Sync.onClick}$$

Pegasus reports a property violation. The counterexample revealed that the application may access the contacts if the user clicks the Result button. We verified manually that clicking Result displays the results of synchronising pictures. This behaviour is innocuous, so we refined the property as below.

$$\neg\text{Access-Contacts}\ \mathbf{U}\left(\begin{array}{ll} & \text{Result.onClick} \\ \vee & \text{Sync.onClick}\end{array}\right)$$

The application satisfies the refined property.

6.3.2 Malicious Applications

Malicious Recorder. This application contains the same functionality as the benign recording application. However, it also records audio for 15 seconds whenever a new SMS is received. We check the same specifications as the Simple Recorder application and verification fails. The counterexample shows that that a timer can trigger recording.

ZitMo. In most benign applications, the SMS messages received should either be passed on to the next Broadcast Receiver or displayed to the user. Thus we check

$$\neg\text{BroadcastAbort}$$

to see whether the ZitMo application discards SMSes without notifying the user. Pegasus reports a violation, which we confirmed manually.

SMS Replicator Secret. We checked two properties of this application.

$$\neg\text{Send-SMS}\ \mathbf{U}\ \text{Button.onClick}$$
$$\neg\text{BroadcastAbort}$$

Both properties are violated, because the application malicious sample sends SMS messages without user interaction, and deletes certain incoming SMS messages.

ADSms. We use Pegasus to study this application and understand how it uses permissions. We check three proper-

ties.

$\neg$Kill-Background-Processes

$\neg$Read-IMEI

$\neg$Send-SMS

The counterexamples show that this application registers a broadcast receiver to kill anti-virus processes. We also discovered that the broadcast receiver starts a new process, which reads IMEI information and sends SMS messages to premium-rate numbers.

7 Related Work and Discussion

We build upon work at the intersection of specification languages, program analysis, model checking, and Android security. These areas are all mature and a comprehensive survey is beyond the scope of this paper. We only attempt to place our work in the context of either seminal or very recent papers in each area.

Specification Languages. A specification language may be external to a program, as with a temporal logic or internal to a program as in design-by-contract mechanisms. See [32] for a hardware-oriented survey of industrial formats for temporal logics, and [6] for an overview of JML and ESC/Java2, which is well known, but only one of many specification mechanisms for Java.

Our use of rewriting achieves a form of in-line monitoring, an idea articulated in [17]. Monitoring for security policies has been implemented in EFSA [29] and PSLang [16], and with a focus on mobile security. in the S3MS project [10], The Apex [27] system uses Android permissions to guide runtime monitoring, while our monitoring policies are defined by custom security properties.

We use runtime monitoring to deal with cases in which static analysis is ineffective, such as in the presence of dynamic class loading or running native code. This combination can also be viewed as an optimisation that reduces the overhead of runtime checks, and leverages the strengths of both. JAM, developed concurrent to our work, combines model checking with abstraction-refinement to alleviate monitoring overheads [22]. Techniques from JAM can be used to improve our rewriting tool.

Static Analysis. The design of our abstraction engine combines ideas from abstract interpretation [7] and software model checking. The closest related work is the SLAM toolkit [2], for checking properties of device drivers. Similar to SLAM, we check policies about the interaction of an application and the operating system, and use cubes over Boolean variables for symbolically encoding . Unlike SLAM, we abstract the operating system context of an event. Moreover, instead of a theorem prover, we use a domain specific event- and API-semantics engines to determine how mode variables are transformed.

The ideas in SLAM has been extended to *lazy abstraction* [24], which interleaves the abstraction and checking process, and to YOGI [4], which combines testing and theorem proving to construct abstractions. Most developments that follow SLAM, such as those surveyed in [26] focus on improving either model checking or abstraction. Our work is orthogonal because we compute a different type of abstraction. Much work successive to SLAM, can be lifted to Android and used to improve the construction or verification of PEGs.

Though program analysis is a mature field, event driven programs have only recently received attention. Interprocedural data-flow analysis with a finite-height, powerset lattice is EXPSPACE-hard [25]. In a language like Java, all analysis depends on the quality of points-to analysis. Points-to-analysis in the presence of an event-queue faces similar complications as with function pointers [11]. These are obstacles that will have to overcome if we wish to improve our analysis.

Android Security. TaintDroid [13] supports dynamic taint-tracking for Android applications. It explores one execution at a time, while our system checks all the possible behaviors of an application. Security analysis based on control-flow patterns and simple static analysis has been used in [14] to detect a range of malicious behaviour. A semantically richer static analysis has been applied to Android in [28], but the focus is on common bugs rather than security properties.

The Stowaway system [18] combines static analysis with a permission map to identify applications requesting more permissions than they use. Permission-based approaches [1, 15, 27] use a map from API calls to Android permissions. Pegasus uses use a map from API calls and arguments to a custom-defined set of actions. This set of actions extends the abstraction of APIs provided by permissions.

Discussion. Analysis of PEGs provides richer semantic information than is available in standard program representations. PEGs abstract the operating system context in which event handlers execute, and model checking of PEGs provides more information than pattern matching on syntactic program artefacts. We can detect permissions used in background tasks, or in event-handlers triggered by invisible buttons. We are not aware of other techniques that can detect such behaviour.

Analysis of PEGs is not a panacea for malware detection. However, our analysis gives security analysts an advantage in their arms race against new malware, by aiding in identifying a new class of malicious behaviour. While attackers can work to evade our analysis mechanisms, e.g., using native code or dynamic class loading, such evasion requires code to adopt a more convoluted structure or exhibit more circuitous behavior, compared to benign applica-

tions — thereby making the code more conspicuous. Thus, much as the ZOZZLE defense against heap spraying attacks in Javascript [8], our analysis can support and facilitate the identification of malware. Even simple pattern matching of syntactic structure, or triggered runtime checks, may be sufficient to reveal anomalies that indicate malicious intent.

8 Conclusion

We have presented a new approach to specifying and detecting malicious behaviour in Android applications. Our conceptual contribution is the Permission Event Graph (PEG) a new, domain-specific program abstraction, which captures the context in which events fire, and the context-sensitive effect of event-handlers. We devised a new static analysis procedure for constructing PEGs from Dalvik bytecode, and our implementation models of the Android event-handling mechanism and several APIs. Our system Pegasus can detect and prevent security specifications that characterises safe interaction between user-driven events and application actions. Given the rapidly increasing popularity and sophisticated functionality of mobile applications, we believe that analysis systems such as Pegasus will improve the capabilities of security analysts.

Our work leads to several questions. One question is to incorporate existing techniques to improve the precision and efficiency of analyses used to construct and analyse PEGs. A particularly interesting question is to determine if counterexample-driven refinement can be used to improve both verification and rewriting. A second problem is to identify applications PEGs in other contexts, such as to measure the complexity and usability of user-interfaces, and statically provided permission information. Answering such questions is left as future work.

9 Acknowledgements

We are deeply indebted to Úlfar Erlingsson for his assistance, feedback and encouraging support. We thank the anonymous reviewers for their comments.

This material is based upon work partially supported by the National Science Foundation under Grants No. 0842695, No. 0831501 and the Air Force Office of Scientific Research (AFOSR) under MURI award FA9550-09-1-0539. Any opinions,findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] K. Au, Y. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android permission specification. In *Proc. of the ACM Conference on Computer and Communications Security*, pages 217–228. ACM, 2012.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. of the Symposium on Programming Language Design and Implementation*, pages 203–213. ACM, 2001.

[3] T. Ball and S. K. Rajamani. SLIC: A Specification Language for Interface Checking (of C). Technical Report MSR-TR-2001-21, Microsoft Resesarch, 2001.

[4] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proc. of the International Symposium on Software Testing and Analysis*, pages 3–14. ACM, 2008.

[5] E. Bodden, P. Lam, and L. Hendren. Object representatives: a uniform abstraction for pointer information. In *Proc. of the 1st International Academic Research Conference of the British Computer Society (Visions of Computer Science)*, pages 391–405, 2008.

[6] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Proc. of the International Symposium on Formal Methods for Components and Objects*, pages 342–363. Springer, 2005.

[7] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2:511–547, 1992.

[8] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. ZOZZLE: fast and precise in-browser JavaScript malware detection. In *Proc. of the USENIX conference on Security*, pages 3–3. USENIX Association, 2011.

[9] R. De Nicola and F. Vaandrager. Action versus state based logics for transition systems. In *Proc. of the LITP Spring school on theoretical computer science on Semantics of systems of concurrent processes*, pages 407–419. Springer, 1990.

[10] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. The S3MS.NET run time monitor. *Electronic Notes in Theoretical Computer Science*, 253(5):153–159, 2009.

[11] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of the Symposium on Programming language design and implementation*, PLDI '94, pages 242–256. ACM, 1994.

[12] W. Enck. Defending users against smartphone apps: Techniques and future directions. In *International Conference on Information Systems Security*, pages 49–70. Springer, 2011.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the USENIX Conference on Operating Systems Design and Implementation*, pages 1–6. USENIX, 2010.

[14] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proc. of the USENIX conference on Security*, pages 21–21. USENIX Association, 2011.

[15] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proc. of the*

ACM conference on Computer and communications security, pages 235–245. ACM, 2009.

[16] Ú. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, 2004.

[17] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *Proc. of the Workshop on New security paradigms*, pages 87–95. ACM, 2000.

[18] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. of the Conferenc. on Computer and Communication Security*, pages 627–638. ACM, 2011.

[19] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proc. of the workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14. ACM, 2011.

[20] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: attacks and defenses. In *Proc. of the USENIX Security Conference*. USENIX Association, 2011.

[21] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):9, 2008.

[22] M. Fredrikson, R. Joiner, S. Jha, T. Reps, P. Porras, H. Saïdi, and V. Yegneswaran. Efficient runtime policy enforcement using counterexample-guided abstraction refinement. In *Proc. of the Conference on Computer Aided Verification*, pages 548–563. Springer, 2012.

[23] M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. RiskRanker: scalable and accurate zero-day Android malware detection. In *Proc. of Mobile Systems, Applications, and Services*, pages 281–294. ACM, 2012.

[24] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of the Symposium on Principles of Programming Languages*, POPL, pages 58–70. ACM, 2002.

[25] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. In *Symposium on Principles of Programming Languages*, volume 42 of *POPL*, pages 339–350. ACM, Jan. 2007.

[26] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, Oct. 2009.

[27] M. Nauman, S. Khan, and X. Zhang. Apex: extending Android permission model and enforcement with user-defined runtime constraints. In *Proc. of the Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.

[28] É. Payet and F. Spoto. Static analysis of Android programs. In *Proc. of the Conference on Automated Deduction*, pages 439–445. Springer, 2011.

[29] R. Sekar, V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. *ACM SIGOPS Operating Systems Review*, 37(5):15–28, 2003.

[30] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, 2008.

[31] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Proc. of the Conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–. IBM Press, 1999.

[32] M. Y. Vardi. From philosophical to industrial logics. In *Proc. of the Indian Conference on Logic and Its Applications*, pages 89–115. Springer, 2009.

A Application, Action and Event Details

Table 4 lists the sample applications used to evaluate Pegasus. Table 5 lists the security actions and events used in the specifications.

Name	Description
Who's Calling?	An application that uses text-to-speech to announce the caller ID of an incoming call.
Share Contacts	An application that allows users to send and receive contacts via SMS.
Geotag	An application that embeds the users current GPS location into uploaded pictures.
Find My Phone	Responds with the GPS location of the device upon receiving a message containing a specific keyword.
Simple Recorder	An application that recording audio from the microphone.
Diet SD Card	An application that recommends files to delete from the SD card.
SMS Cleaner Free	An application that allows the user to delete SMS messages based on a variety of features.
SyncMyPix	An application that syncs pictures from Facebook friends to the contact list in the phone.
SMS Replicator Secret	Spyware that automatically forwards SMS messages to a number selected by the user installing the application.
ADSms	A malicious application that stealthily sends premium SMS messages and kills certain background processes.
ZitMo	A malicious application that intercepts SMS messages in order to harvest two-factor authentication codes issued by banks.
HippoSMS	A malicious application that stealthily deletes all incoming SMS messages.
DroidDream (Steamy Window)	A trojan packaged with a benign application that stealthily sends SMS messages, installs new applications and receives commands from a C&C server.
Zsone (iMatch)	A malicious application that sends SMS messages to premium numbers.
Geinimi (Monkey Jump 2)	A trojan packaged with a benign application that stealthily sends SMS messages, installs new applications and receives commands from a C&C server.
Spitmo	A trojan that intercepts and filters incoming SMS messages.
Malicious Recorder	Mimics a benign sound recorder but also records audio when an SMS message is received.

Table 4. Sample applications used in the evaluation: 8 benign applications (top) and 9 malicious applications (bottom).

Name	Description
Access-Contacts	Access the contacts list.
Insert-Contacts	Insert to the contacts list.
Send-SMS	Send SMS.
Access-GPS	Access GPS location information.
Start-Recording	Start recording audio.
Stop-Recording	Stop recording audio.
BroadcastAbort	Abort an ordered broadcast (used by many malicious applications to prevent SMS messages from being displayed to the user).
Access-SD	Access the SD card.
Kill-Background-Processes	Kill background processes.
Read-IMEI	Read IMEI information.
Access-Internet	Access the Internet.
REC.onClick	The REC button's `onClick` handler.
STOP.onClick	The STOP button's `onClick` handler.
Clean.onClick	The Clean button's `onClick` handler in Diet SD Card.
Locate.onClick	The Locate button's `onClick` handler in Geotag.
Send.onClick	The Send button's `onClick` handler in Share Contacts.
Insert.onClick	The Insert button's `onClick` handler in Share Contacts.
Select-Contact.onClick	The Select-Contact button's `onClick` handler in SMS Cleaner Free.
Button.onClick	The general `onClick` handler for any button.
AdTask.onPreExecute	The `onPreExecute` handler from a ASyncTask created by Google Ads library in Geotag.
AdTask.doInBackground	The `doInBackground` handler from a ASyncTask created by Google Ads library in Geotag.
PhoneCall.onReceive	The `onReceive` event from a broadcast receiver in the Who's Calling? application.

Table 5. Security actions and events used in specifications.