Correlating UI Contexts with Sensitive API Calls: Dynamic Semantic Extraction and Analysis

Jie Liu, Dongjie He, Diyu Wu and Jingling Xue
School of Computer Science and Engineering, UNSW Sydney, Australia

***Abstract*—The Android framework provides sensitive APIs for Android apps to access the user's private information, e.g., SMS, call logs and locations. Whether a sensitive API call in an app is legitimate or not depends on whether the app has provided enough natural-language semantics to reflect the need for the permission. The prior efforts on analyzing description-to-permission fidelity in an app are all static. Some check whether the permissions requested (or sensitive APIs used) by the app are consistent with the functionalities described by the app. These app-level techniques are too coarse-grained, as they cannot tell if a sensitive API call under a certain runtime context, such as a UI state, is legitimate or not. Others attempt to establish this connection by performing a data-flow analysis, but such fine-grained API-level static analyses are too imprecise to handle a variety of dynamic language features used in Android apps, including dynamic class loading, reflection and code obfuscation.**

We introduce APICOG, an automated fine-grained API-level approach, representing the first dynamic description-to-permission fidelity analysis for an Android app that can check if a sensitive API call is legitimate or not under a given runtime context. APICOG relates each sensitive API call with a UI state, called its *UI context*, under which the call is made via dynamic analysis and then extracts the text-based semantics for each UI context from its associated text- and image-typed attributes by applying a natural language processing (NLP) technique. Finally, APICOG relies on machine-learning to deduce if a sensitive API call under a UI context is legitimate or not. We have evaluated APICOG with thousands of Android apps drawn from a third-party market and a malware dataset, achieving an accuracy of 97.7%, a precision of 94.1% and a recall of 92.8% overall, outperforming the prior art in all the three metrics.

I. INTRODUCTION

Mobile devices have become an integral part of our lives, with the Android accounting for 85.4% of the worldwide smartphone market share in 2020 Q2 [1]. Unfortunately, mobile devices store and process an enormous amount of personal information, such as SMS messages and contacts, raising many security and privacy concerns. In general, an app advertises its functionalities in its description to justify the permissions requested in order to access private user information (via sensitive APIs). For example, a photo editor app may access the user's album to beautify pictures and a map app may access the user's location to provide navigation services. However, an app often also abuses the permissions granted, by, for example, stealing private information via sensitive API calls in the circumstances unexpected and also unaware by the user.

Existing research efforts on analyzing description-to-permission fidelity for Android apps are static. There are two types of techniques: app-level techniques check to see if a sensitive API can be used legitimately by an app throughout [2]–[6] and API-level techniques decide if a sensitive API call can be used legitimately in a given runtime context [7], [8].

For app-level techniques [2]–[6], AUTOCOG [4] is a representative. For an app, AUTOCOG simply correlates its description (as required by Google Play) with its permissions requested (or implicitly the set of sensitive APIs used). However, such app-level techniques are coarse-grained. While effective in separating a map app from an alarm clock app (in terms of their legitimacy in accessing the user's location), app-level techniques cannot answer whether an app's behavior, under a given runtime context, should happen or not. For example, a game app may use `sendTextMessage()` in two different runtime contexts, one for providing a customized service with the user's knowledge, e.g., an SMS message verification with a send button in a login interface and the other for sending some personal information to the developers when the app stays in the game interface (without the user's knowledge). The correlation of `sendTextMessage()` with its runtime context is *positive* in the former but *negative* in the latter.

For API-level techniques [7], [8], ASDROID [8] is a representative, which attempts to discover statically the legitimate runtime contexts under which a sensitive API call can be made by performing a data-flow analysis. ASDROID relates a sensitive API call to some UI components (identified by their "text" attributes) and detects the malicious associations algorithmically. In practice, fine-grained static techniques such as ASDROID are ineffective, for two reasons. First, the "text" attribute of a UI component can be null, encrypted or statically unknown (e.g., downloaded dynamically at runtime). Second, static analyses for Android apps are often either unscalable [9] or imprecise (when applied to maximize their soundness), especially since dynamic features, such as dynamic class loading, reflection, code encryption, and code obfuscation, are widely used (particularly in malicious apps) [10]–[17].

With this vision, in this paper, we introduce APICOG, the first automated fine-grained API-level dynamic analysis for assessing description-to-permission fidelity in Android apps. APICOG is fine-grained, operating at the API level, since it correlates a sensitive API call with a (unique) *UI context*, e.g., the information in a map view interface. Intuitively, the main challenge faced by APICOG lies in how to extract such a UI context that is deeply embedded in an app relative to its associated sensitive API call. Our key insight is that a UI context for a sensitive API call is embedded in an Android UI state, consisting of text- and image-typed attributes, which

2332-6549/20/$31.00 ©2020 IEEE
DOI 10.1109/ISSRE5003.2020.00031

provide the semantics needed to justify the legitimacy of the call. For example, if a `getLastKnownLocation()` call exposes the user's location after the user has opened a map view, then its UI context can be obtained from the map view.

Given an app, APICOG first performs a dynamic analysis to build its UI state transition graph and associates a sensitive call executed with its most recent UI state. APICOG then maps a UI state to a UI context, characterized by the textual information distilled from a number of attributes found in the view tree of the UI layout in this UI state. Currently, we obtain such textual information from three text-typed attributes, "text", "resource_id" and "class name", and one image-typed attribute "view_str" (for images containing embedded texts). Then, we use an NLP technique to convert a UI context into a list of action-resource pairs [18]. As a result, we can represent every pair consisting of a sensitive API call and its UI context as a list of action-resource pairs. Finally, APICOG applies machine learning to compute automatically the semantic relatedness between a sensitive API call and a UI context. The call is likely to be legitimate if this association is positive and possibly illegitimate otherwise.

We have evaluated APICOG with a large number of Android apps. By correlating a sensitive API call with its UI context, APICOG outperforms the prior app- and API-level techniques (reshaped in our setting) in accuracy, precision and recall.

In summary, this paper makes the following contributions:

- We propose a novel fine-grained API-level approach, APICOG, for assessing automatically description-to-permission fidelity in Android apps, by checking the legitimacy of an API call under a particular UI context.
- APICOG (open-sourced at http://www.cse.unsw.edu.au/~corg/apicog) can be used either as a market-level vetting tool for App stores such as Google Play or by developers and/or researchers for finding information leaks in apps.
- We have evaluated APICOG with 3127 Android apps, including 1500 malware apps from Drebin, an Android malware dataset [19] and all the 1627 benign apps in the popular Android app repository F-Droid [20]. APICOG finds a total of 5099 (sensitive API call, UI context) pairs in 976 out of 3127 apps. With half of these pairs being used as the training data and the other half as the test data, APICOG achieves an accuracy of 97.7%, a precision of 94.1% and a recall of 92.8% overall, outperforming the prior art in all these metrics.

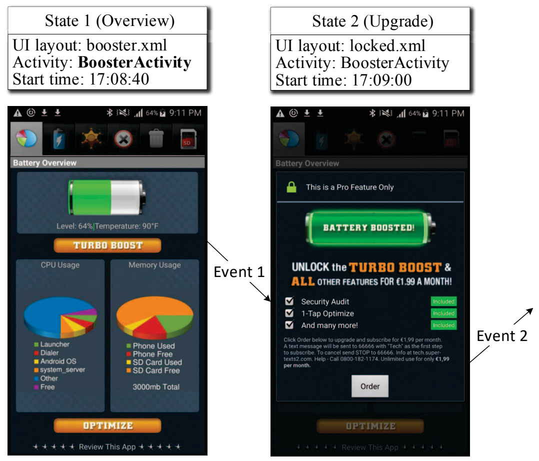
II. A MOTIVATING EXAMPLE

We illustrate the main challenges faced in correlating a sensitive API call with its UI context deeply embedded in the app and discuss how we address this challenge. We use an example abstracted from `Battery Supercharger` in Figure 1 to go through the key steps of APICOG. This malware app, which is obtained from Drebin [19], can help the user optimize battery usage and extend battery life.

In this example, we focus on explaining how we determine the sensitive API call `getLastKnownLocation()` made reflectively in line 25 of Figure 1(a) (with its name encrypted) is illegitimate while the sensitive API call

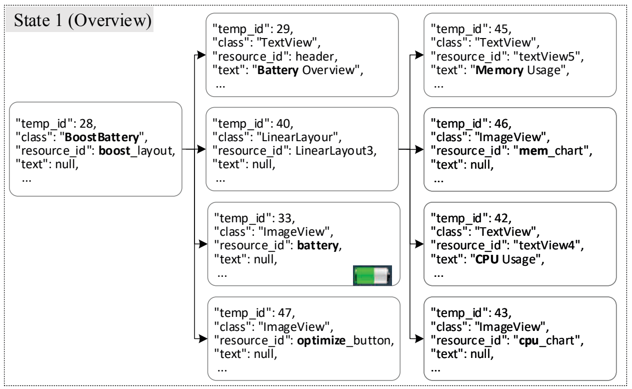
```
 1 final class ClickOrderListener implements OnClickListener {
 2   final Map map;
 3   ClickOrderListener(Map map) { this.map = map;}
 4   public void onClick(View view) {
 5     String message = buildMessageFromMap(this.map);
 6     SmsManager smsManager = SmsManager.getDefault();
 7     smsManager.sendTextMessage(phoneNum, null, message, pendingIntent, null);
 8 }}
 9 public class BoosterActivity extends Activity implements OnClickListener {
10   public static void showOrderDialog(Context context, Map map) {
11     Dialog dialog = new Dialog(context); dialog.setContentView(inflate);
12     Button button = (Button) inflate.findViewById(R.id.buttonOrder);
13     button.setOnClickListener(new ClickOrderListener(map));
14     dialog.show();
15   }
16   public void onClick(View view) {
17     switch (view.getId()) {
18     case R.id.boost_button :
19       String service = CipherUtil.decrypt("bG9jYXRp");// service = "location";
20       Object lm = ctx.getSystemService(service); // get LocationManager object
21       // mtdName = "getLastKnownLocation";
22       String mtdName = CipherUtil.decrypt("WpMUG6kCL/VztBsv");
23       Method mtd = lm.getClass().getMethod(mtdName, String.class);
24       // get the current location
25       Object location = mtd.invoke(lm, provider);
26       map.put(serviceName, location);
27       showOrderDialog((Context) this, map);
28     ...
29   }}
30   public void onCreate(Bundle bundle) {
31     findViewById(R.id.boost_button).setOnClickListener(this);
32 }}
```

(a) Adapted from the malware app Battery Supercharger


State 1 (Overview)
UI layout: booster.xml
Activity: **BoosterActivity**
Start time: 17:08:40

State 2 (Upgrade)
UI layout: locked.xml
Activity: BoosterActivity
Start time: 17:09:00

(b) Part of the UI state transition for (a) ("Event 1": click the boost button and "Event 2": click the order button)


State 1 (Overview)

"temp_id": 28,
"class": "**BoostBattery**",
"resource_id": **boost**_layout,
"text": null,
...

"temp_id": 29,
"class": "TextView",
"resource_id": header,
"text": "**Battery** Overview",
...

"temp_id": 40,
"class": "LinearLayour",
"resource_id": LinearLayout3,
"text": null,
...

"temp_id": 33,
"class": "ImageView",
"resource_id": **battery**,
"text": null,
...

"temp_id": 47,
"class": "ImageView",
"resource_id": **optimize**_button,
"text": null,
...

"temp_id": 45,
"class": "TextView",
"resource_id": "textView5",
"text": "**Memory** Usage",
...

"temp_id": 46,
"class": "ImageView",
"resource_id": "**mem**_chart",
"text": null,
...

"temp_id": 42,
"class": "TextView",
"resource_id": "textView4",
"text": "**CPU** Usage",
...

"temp_id": 43,
"class": "ImageView",
"resource_id": "**cpu**_chart",
"text": null,
...

(c) The simplified UI layout for State 1 in (b)

Fig. 1: A motivating example.

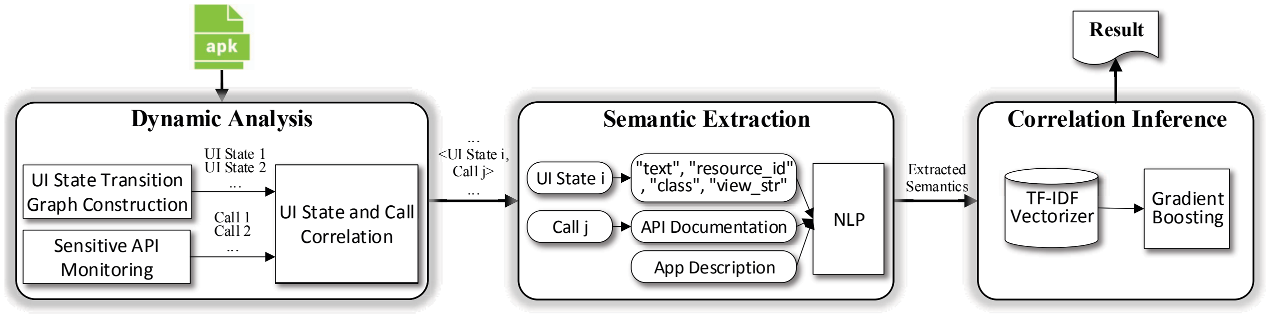


Fig. 2: A schematic overview of APICOG.

`sendTextMessage()` in line 7 is legitimate. Section IV-G discusses a few other legitimate/illegitimate associations.

In Figure 1(b), the left diagram depicts a state ("State 1 (Overview)") when `BoosterActivity` becomes the top activity. Once the user clicks its "TURBO BOOST" button, the callback `onClick()` in line 16 will be triggered (Event 1). This event first invokes the sensitive API `getLastKnownLocation()` through string decryption [21] and reflection [22], [23] (lines 19 – 25) and then opens an order dialog (lines 26 – 27) as the top UI to transit the app to another state ("State 2 (Upgrade)"), as depicted in the right diagram of Figure 1(b). If the user clicks its "Order" button (by requesting an upgrade to a professional version), the callback `onClick()` in line 4 will be triggered (Event 2), which will send an order message (including the user's location) to a remote server (lines 5 – 7).

A. *Prior Work*

There are two types of static techniques for assessing description-to-permission fidelity in Android apps (Section I). The app-level techniques [2]–[6] are ineffective for vetting this app. Even if its description states the need for accessing the user's location due to the navigation services provided, these techniques do not know what API call will be made reflectively in line 25 of Figure 1(a) and whether it is legitimate or not when `BoosterActivity` becomes the top activity.

On the other hand, the API-level techniques [7], [8] attempt to fix this limitation by performing a data-flow analysis. Due to code encryption, however, their static techniques [10]–[15] cannot find the API call made reflectively in line 25 either.

B. *The* APICOG *Approach*

Figure 2 gives an overview of APICOG. There are three components used in analyzing an app (given in the form of an APK file). "Dynamic Analysis" first associates each call with a unique UI state (known as its UI context). "Semantic Extraction" then maps each $\langle$UI State, Call$\rangle$ pair into a list of action-resource pairs by applying an NLP algorithm. Finally, "Correlation Inference" applies machine learning to determine if $\langle$UI State, Call$\rangle$ is positive or negative.

1) Dynamic Analysis: We use a GUI testing tool [24] to analyze our example app dynamically. The objective here is to build a UI state transition graph and associate every sensitive API call executed with its most recent UI state seen.

The *UI state transition graph* is a directed graph, where:

- a node represents a UI state, recording the information about its start time, the top activity, and the view tree of the UI layout in this state, and
- an edge between two nodes represents a UI event (e.g., a mouse or button click) leading to a state transition from the source UI state to the target UI state.

Figure 1(b) depicts a portion of the UI state transition graph, which is built on the fly during the app execution. Currently, there are two nodes, representing two different UI states: (1) the "Battery Overview" state, State 1, identified as $\mathcal{S}_{\text{overview}}$ and (2) the "Order Dialog" state, State 2, identified as $\mathcal{S}_{\text{upgrade}}$. Note that the "Battery Optimization" state after State 2, identified as $\mathcal{S}_{\text{optimize}}$, has been elided. Consider Figure 1(b). Given $\mathcal{S}_{\text{overview}} \xrightarrow{\text{Event 1}} \mathcal{S}_{\text{upgrade}}$, the user can switch from the "Battery Overview" state to the "Order Dialog" state by clicking the "TURBO BOOST" button (Event 1). Given $\mathcal{S}_{\text{upgrade}} \xrightarrow{\text{Event 2}} \mathcal{S}_{\text{optimize}}$, the user can click the "Order" button to upgrade the app to a professional version and then switch to the "Battery Optimization" state (Event 2).

As shown in the top of Figure 1(b), every state consists of three pieces of information: its start time, the class name of its top activity, and its UI layout. Take $\mathcal{S}_{\text{overview}}$ as an example. Its top activity is `BoosterActivity`, which is displayed on the screen at 17:08:40. Its UI layout, `booster.xml`, shown as a simplified view tree, is given in Figure 1(c). Each view in the tree is represented as a node, tagged with a variety of attributes. Currently, we consider three text-typed attributes, "text", "resource_id" and "class name", and one image-typed attribute, "view_str", as they are sufficient collectively to identify the functionality of a view. In particular, "view_str" contains a dynamically generated string, representing the name of the image used (if any) for the corresponding view. In future work, some other attributes can be included, if necessary.

During the dynamic analysis, we also monitor the sensitive API calls made. As shown in Figure 1(a), when the app runs in $\mathcal{S}_{\text{overview}}$, the customer callback `onClick()` of `BoosterActivity` will trigger the API call `getLastKnownLocation()` in line 25, $\mathcal{I}_{25}$, to be executed at time 17:08:42, identified as $\mathcal{I}_{25}^{17:08:42}$. As $\mathcal{S}_{\text{overview}}$ is the most recent UI state causing its execution, we have obtained a state-call pair $\langle \mathcal{S}_{\text{overview}}, \mathcal{I}_{25}^{17:08:42} \rangle$. Similarly, we obtain another state-call pair $\langle \mathcal{S}_{\text{upgrade}}, \mathcal{I}_{7}^{17:09:12} \rangle$ to associate the sensitive API call of `sendTextMessage()` made in line

TABLE I: The action-resource pairs extracted from the UI state $\mathcal{S}_{\text{overview}}$ in Figure 1.

Attribute	Values	Action-Resource Pairs
Text	**Battery** Overview, **Memory** Usage, **CPU** Usage	⟨null, battery⟩, ⟨null, memory⟩, ⟨null, CPU⟩
Resource_id	**boost**_layout, header, LinearLayout3, **optimize**_button, **battery**, textView5, **mem**_chart, textView4, **cpu**_chart	⟨boost, null⟩, ⟨optimize, null⟩, ⟨null, memory⟩, ⟨null, cpu⟩, ⟨null, battery⟩
Class name	**BoostBattery**, TextView, LinearLayout, ImageView	⟨boost, battery⟩

7 with its most recent UI state $\mathcal{S}_{\text{upgrade}}$.

2) Semantic Extraction: To justify that $\mathcal{I}_{25}$ is called illegitimately under its triggering UI state $\mathcal{S}_{\text{overview}}$, which represents its UI context, i.e., that $\langle \mathcal{S}_{\text{overview}}, \mathcal{I}_{25}^{17:08:42} \rangle$ is negative, we will extract the semantics from $\mathcal{S}_{\text{overview}}$ and $\mathcal{I}_{25}$ by applying some NLP techniques.

APICOG first processes an app's description and the API documentation related to `getLastKnownLocation()` and its containing class `LocationManager`, and converts all the crawled raw texts into a list of action-resource pairs [18], denoted $\mathcal{C}(\mathcal{S}_{\text{overview}})$. In our motivating example, we will obtain ⟨optimize, battery⟩ from the app description and ⟨update, location⟩ and ⟨access, location⟩ from the API documentation (among others). We then extract the texts from three text-typed attributes, "text", "resource_id" and "class name" from $\mathcal{S}_{\text{overview}}$, (by converting them into English, if necessary). Based on $\mathcal{C}_{\text{syn}}(\mathcal{S}_{\text{overview}})$, which contains the words in $\mathcal{C}(\mathcal{S}_{\text{overview}})$ and their synonyms, we finally extract the action-resource pairs from $\mathcal{S}_{\text{overview}}$, as shown in Table I.

Consider now the UI layout of $\mathcal{S}_{\text{overview}}$ in Figure 1(c). Let us examine the three text-typed attributes one by one. From the "text" attributes of its views, we obtain "Battery Overview", "Memory Usage" and "CPU Usage" and thus ⟨null, battery⟩, ⟨null, memory⟩ and ⟨null, CPU⟩, as $\{\text{battery}, \text{CPU}, \text{memory}\} \subseteq \mathcal{C}_{\text{syn}}(\mathcal{S}_{\text{overview}})$. Moving to "resource_id", we find that "boost_layout", "optimize_button", "mem_chart", "cpu_chart" and "battery" are relevant with respect to $\mathcal{C}_{\text{syn}}(\mathcal{S}_{\text{overview}})$. Thus, we obtain ⟨boost, null⟩, ⟨optimize, null⟩, ⟨null, memory⟩, ⟨null, cpu⟩ and ⟨null, battery⟩. Note that "mem" will be recognised as a synonym of "memory" (Section III). Finally, we look at "class name". The two class names are relevant (with respect to $\mathcal{C}_{\text{syn}}(\mathcal{S}_{\text{overview}})$): `BoosterActivity` (the class of the top activity) and `BoostBattery` (for the view with temp_id = 28). This results in two more pairs, ⟨boost, battery⟩ and ⟨null, booster⟩.

The semantics for $\langle \mathcal{S}_{\text{upgrade}}, \mathcal{I}_{7}^{17:09:12} \rangle$ is obtained similarly.

3) Correlation Inference: APICOG determines if $\langle \mathcal{S}_{\text{overview}}, \mathcal{I}_{25}^{17:08:42} \rangle$ is positive or negative by applying SCIKIT-LEARN, a machine learning tool [25]. APICOG first applies the bag-of-words model [26] to convert the action-resource pairs in $\mathcal{C}(\mathcal{S}_{\text{overview}})$ and for $\mathcal{S}_{\text{overview}}$ (Table I) into a text vector and applies the term-frequency inverse document-frequency (TF-IDF) model to convert the text vector into a numerical feature vector [27]. APICOG then feeds such feature vectors to SCIKIT-LEARN to obtain a classifier. In our example, $\langle \mathcal{S}_{\text{overview}}, \mathcal{I}_{25}^{17:08:42} \rangle$ is found to be negative, since ⟨update, location⟩ and ⟨access, location⟩ from the API documentation are irrelevant to the action-resource pairs found from $\mathcal{S}_{\text{overview}}$ (Table I).

Similarly, we check the state-call pair $\langle \mathcal{S}_{\text{upgrade}}, \mathcal{I}_{7}^{17:09:12} \rangle$. Since the semantics of `sendTextMessage()` are highly related to action-resource pairs like ⟨send, message⟩ and ⟨send, null⟩ abstracted from $\mathcal{S}_{\text{upgrade}}$, APICOG predicts this call to be positive.

III. THE APICOG DESIGN

We describe the three components of APICOG depicted in Figure 2 for assessing description-to-permission fidelity in Android apps dynamically. Given an app, our "Dynamic Analysis" component associates every sensitive API call $I_j^{t_j}$ invoked in line j at time t_j to the most recent UI state S_i (i.e., its UI context), forming a state-call pair $\langle \mathcal{S}_i, \mathcal{I}_j^{t_j} \rangle$. For each $\langle \mathcal{S}_i, \mathcal{I}_j^{t_j} \rangle$, our "Semantic Extraction" component extracts the call-specific semantics from $\mathcal{S}_i$, based on the app's description and the API documentation related to I_j, by applying NLP. Our "Correlation Inference" component applies machine learning to answer if $\langle \mathcal{S}_i, \mathcal{I}_j^{t_j} \rangle$ is legitimate or not.

A. Dynamic Analysis

While an app is being executed on the device, a sequence of UI states are generated. A UI state represents a dynamic snapshot of a view (i.e., activity rendering) displayed on the device's screen. To build a UI state transition graph, APICOG relies on DROIDBOT [24], an automatic input generation tool.

1) Sensitive API Monitoring: When an app is executed, we also monitor all the sensitive API calls triggered. Our API monitor, which is developed in the Xposed framework [28], intercepts and records the sensitive API calls executed while preserving the original Android user experience.

Table II gives the list of APIs monitored. These include (1) the APIs for handling data collection and transmission, e.g., those accessing the user's location, querying the user's contacts, and sending SMS messages (the first five rows), and (2) the APIs for obtaining the phone state changes (the last two rows). In the latter case, a WIFI connection change can be used to infer the user's location [29] and a top activity switching event can frequently act as a trigger for a variety of malicious behaviors, without user awareness [30], [31].

There has been an extensive study on the sensitive APIs that are frequently exploited by malware [32]–[34]. Our current list, which is built based on the prior work, is not intended to be comprehensive but is sufficient for this work.

2) Correlating API Calls with UI States: Our dynamic analysis produces the following two sets as desired. $\mathcal{ST} = \{\mathcal{S}_1, \cdots, \mathcal{S}_n\}$ contains all the UI states encountered, where $\mathcal{S}_i.stime$ gives the start time of the UI state $\mathcal{S}_i$. $\mathcal{CI} = \{\mathcal{I}_{\ell_1}^{t_1}, \cdots, \mathcal{I}_{\ell_m}^{t_m}\}$ contains all the sensitive API calls triggered, where $\mathcal{I}_{\ell_j}^{t_j}$ represents a call appearing in line ℓ_j executed at time t_j.

TABLE II: The list of currently monitored APIs.

Resource Types	Monitored APIs	Description
Microphone and Camera	APIs in AudioRecord, MediaRecorder, and Camera, e.g., startRecording(), start(), and takePicture()	Recording audio and video, and taking pictures
Location	APIs in LocationManager and GoogleMap, e.g., getMyLocation(), requestLocationUpdates(), and getLastKnownLocation()	Accessing device location by querying GPS directly or registering for a GPS listener
Accounts	APIs in AccountManager, e.g., getAccountsByType()	Providing access to a centralized registry of the user's online accounts
Resources provided by ContentProvider	query() in ContentResolver with URI referring to contents such as contacts, SMS messages, and call logs	Providing access to the data maintained by system and applications
Telephony and SMS	APIs in TelephonyManager, BluetoothDevice, and SmsManager, e.g., getLine1Number(), getAddress(), and sendTextMessage()	Accessing the information about the telephony services on the device (e.g., by stealing the user's money via sending SMS messages to a premium-rate number)
Connectivity States	APIs in WifiManager and WifiService, e.g., getScanResults(), getDhcpInfo(), getWifiState(), and getConnectionInfo()	Triggering undesirable behaviors or inferring the user's location (based on the WIFI state changes)
Application States	APIs in ActivityManager, ApplicationPackageManager, and UsageStatsManager, e.g., getInstalledPackages(), getRunningTasks(), and queryUsageStats()	Checking for the availability of an anti-virus or financial app on the device to trigger malicious behaviors, e.g., phishing (based on the installed app's state)

Algorithm 1: Correlating an API call with its UI context.

```
Input  : $\mathcal{ST} = \{\mathcal{S}_1, ..., \mathcal{S}_n\}$
         $\mathcal{CI} = \{\mathcal{I}^{t_1}_{\ell_1}, ..., \mathcal{I}^{t_m}_{\ell_m}\}$
Output: $\mathcal{P}$
1 Function CALLTOSTATE($\mathcal{ST}$, $\mathcal{CI}$)
2   $\mathcal{P} = \emptyset$;
3   Let $\mathcal{S}_{n+1}$ be a pseudo UI state, where
      $\mathcal{S}_{n+1}.stime = \infty$;
4   foreach $\mathcal{I}^{t_j}_{\ell_j} \in \mathcal{CI}$ do
5     foreach $\mathcal{S}_i \in \mathcal{ST}$ do
6       if $\mathcal{S}_i.stime < t_j < \mathcal{S}_{i+1}.stime$ then
7         $\mathcal{P}$.add($\langle \mathcal{S}_i, \mathcal{I}^{t_j}_{\ell_j} \rangle$);
8   return $\mathcal{P}$;
```

For each call $\mathcal{I}^{t_j}_{\ell_j} \in \mathcal{CI}$, we correlate it with a unique UI context, which is the most recent state in $\mathcal{ST}$, i.e., $\mathcal{S}_i \in \mathcal{ST}$ such that $\mathcal{S}_i.stime$ is the largest satisfying $\mathcal{S}_i.stime < t_j$, yielding a state-call pair $\langle \mathcal{S}_i, \mathcal{I}^{t_j}_{\ell_j} \rangle$. For completeness, Algorithm 1 is given to generate all such state-call pairs. In our example illustrated in Figure 1, there are two state-call pairs, $\langle \mathcal{S}_{\text{overview}}, \mathcal{I}^{17:08:42}_{25} \rangle$ and $\langle \mathcal{S}_{\text{upgrade}}, \mathcal{I}^{17:09:12}_{7} \rangle$.

As APICOG is a dynamic analysis, we may have $\ell_j = \ell'_j$ for two dynamic calls $\mathcal{I}^{t_j}_{\ell_j}$ and $\mathcal{I}^{t'_j}_{\ell'_j}$, implying that the same function is called twice from the same call site. However, these two calls may or may not be associated with the same UI context.

B. Semantic Extraction

Given a state-call pair $\langle \mathcal{S}, \mathcal{I} \rangle$, we will obtain its semantics as a list of action-resource pairs. Note that the execution time of $\mathcal{I}$ is dropped as it is not relevant in semantic extraction.

As motivated in Section II, the action-resource pairs for $\langle \mathcal{S}, \mathcal{I} \rangle$ come from three sources: (1) the app description, (2) the $\mathcal{I}$-related API documentation, and (3) the UI context $\mathcal{S}$ (Table I). Below we describe how to handle each and then give an NLP-based algorithm.

1) The App Description: Apps' natural-language descriptions provided in most app markets (e.g., Google Play and F-Droid) by developers, are streamlined product definitions, which provide an intuitive idea about their functionalities including some security-related information. These app descriptions are also a source of semantics leveraged by existing coarse-grained approaches [2]–[5], as further discussed in Section IV-H. If an app does not come with a description, APICOG will distill a textual description from the UI states preceding $\mathcal{S}$ as a substitute, since this is what the user sees before $\mathcal{S}$ is reached, as discussed in Section III-B4.

Apparently, different apps may target users from different countries. Thus, app descriptions may be written in different languages, sometimes containing paragraphs in multiple languages. For a given app, we use Google's Language Translator [35] to obtain a description in English only.

2) The API Documentation: Google provides an API documentation for all the methods/classes in the Android framework [36]. For a given API call $\mathcal{I}$, we consider only the part of the API documentation related to $\mathcal{I}$ and its containing class.

3) The UI State: For a UI state $\mathcal{S}$, we extract its text-based semantics from its UI layout. Again, we rely on Google's Language Translator [35] to handle non-English texts. After having experimented with a large number of UI layouts, we have identified three attributes in a UI layout to be the mostly relevant. These include three text-typed attributes, "text", "resource_id" and "class name", and one image-typed attribute, "view_str". For each attribute, its values in all the views in the entire view tree of the UI layout of $\mathcal{S}$ are considered. For a text-typed attribute in a view, its text can be directly read off. For an image-typed attribute in a view, we will extract its embedded texts (if any) from its associated image. The other attributes can be included, if necessary, in future work.

Below we discuss why we have made these selections.

- **"text".** For the "text" attribute in a view, its associated text will be displayed on the screen and thus reflects its purposes. For the UI layout of $\mathcal{S}_{\text{overview}}$ depicted in Figure 1(c), the text for the view with temp_id = 29 is "Battery Overview".

Algorithm 2: Extracting a list of action-resource pairs from a state-call pair by applying an NLP technique.

```
   Input  : $\langle \mathcal{S}, \mathcal{I} \rangle$
   Output: $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$
1  Function ActResExtraction($\mathcal{S}$, $\mathcal{I}$)
2      $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$ = nlpExtractor(App Description);
3      $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$.addAll(nlpExtractor($\mathcal{I}$-related API Doc));
4      $textSet$ = all texts extracted from $\mathcal{S}$;
5      $(actSet, resSet)$ = nlpSynonym($words$ in
         $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$);
6      foreach $text \in textSet$ do
7          $Acts = \{act \in actSet \mid$
             $act$ is a subtring of $text\}$;
8          $Rscs = \{res \in resSet \mid$
             $res$ is a subtring of $text\}$;
9          foreach $act \times res \in Acts \times Rscs$ do
10             $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$.add($\langle$act, res$\rangle$);
11     $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$.remove($\langle$null, null$\rangle$);
12     return $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$;
```

- **"resource_id".** For many views, such as images and layouts, their "text" attributes are often null. In this case, their "resource_id" attributes may serve to describe their purposes. Consider again the UI layout of $\mathcal{S}_{\text{overview}}$ in Figure 1(c). For the `ImageView` view with temp_id = 47, text = null, but resource_id = "optimize_button", reflecting adequately its purposes. For the `ImageView` view with temp_id $= 33$, only its image icon is displayed. However, we can still understand its meaning if we exploit the information provided in its resource_id, "battery".
- **"class name".** For class names, including Android system views (e.g., `com.google.android.gms.maps.MapView` and `com.google.android.cameraview.CameraView`) and custom classes (implemented to serve some special needs for the developers), their keywords usually reflect their purposes. For the UI layout of $\mathcal{S}_{\text{overview}}$ in Figure 1(c), its top activity is named `BoosterActivity` and its view with temp_id = 28 is named `BoostBattery`, with both class names containing the keyword "boost".
- **"view_str".** The embedded texts in images may reflect the purposes of these images. We use an optical character recognition (OCR) engine to convert the printed texts in images into editable texts [37], as described in Section IV.

4) NLP: For a state-call pair $\langle \mathcal{S}, \mathcal{I} \rangle$, we apply now NLP to clean the raw texts obtained by converting them into a list of action-resource pairs, $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$, according to Algorithm 2. Note that two state-call pairs, $\langle \mathcal{S}_i, \mathcal{I}_\ell \rangle$ and $\langle \mathcal{S}_{i'}, \mathcal{I}_{\ell'} \rangle$, are naturally indistinguishable if $\mathcal{S}_i$ and $\mathcal{S}_{i'}$ yield the same set of action-resource pairs (due to typically being the same UI state) and $\ell = \ell'$ (same function call).

We explain the key steps of Algorithm 2 below.

In line 2, we add to $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$ the action-resource pairs found from the app description (Section III-B1) by applying *nlpExtractor()*, which can be implemented using any NLP algorithm [18]. For a description-less app, we will use the texts extracted from all the UI states preceding $\mathcal{S}$ as a substitute, as explained in Section III-B1. This is done exactly as in lines 5 – 11 but omitted. In line 3, we add to $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$ the action-resource pairs found from the $\mathcal{I}$-related API documentation (Section III-B2). At this point, we filter out all action-resource pairs that are found in more than half of all distinct state-call pairs analyzed in an app since they are considered as non-informative. For example, "application" and "Android" are universal in all the apps and will thus be ignored.

In the rest of this algorithm, we add to $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$ the action-resource pairs distilled from the UI state $\mathcal{S}$ (Section III-B3). We cannot do so by applying *nlpExtractor()* directly to the texts extracted from $\mathcal{S}$ for two reasons. First, the texts obtained from $\mathcal{S}$ do not usually form grammatically English sentences. For the view with temp_id = 19 in Figure 1(c), resource_id = "mem_chart". Second, many words obtained from the four attributes,"text", "resource_id", "class name" and "view_str", are non-informative as they are universal in the UI layouts (e.g., "LinearLayout" and "TextView" in Figure 1(c)).

For a state-call pair $\langle \mathcal{S}, \mathcal{I} \rangle$, we propose a new way to extract the action-resource pairs from $\mathcal{S}$ by using the action-resource pairs already found from the app description and $\mathcal{I}$-related API documentation in lines 2 – 3, which are recorded in $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$, as the initial seeds. This is done in lines 4 – 11. In line 4, we extract all the texts from $\mathcal{S}$, as described in Section III-B3, and store them in $textSet$. In line 5, $actSet$ ($resSet$) is initialized as the set of all synonyms of sensitive words, i.e., actions (resources) in $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$, based on Word2Vec [38]. For example, if "map" is a resource in $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$, its synonyms include "location", "coordinate", and "latitude" (among others). In lines 6 - 11, we map each text in $textSet$ separately into a list of action-resource pairs. (If $Acts = \emptyset$ ($Rscs = \emptyset$), then we assume $Acts = \{\text{null}\}$ ($Rscs = \{\text{null}\}$). This allows APICog to capture as much sensitive semantics as possible from a UI state precisely.

C. Correlation Inference

Given $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$ for a state-call pair $\langle \mathcal{S}, \mathcal{I} \rangle$, we can apply machine learning to determine if the pair is positive (i.e., if $\mathcal{I}$ is legitimate) or negative (i.e., if $\mathcal{I}$ is illegitimate).

We first convert the action-resource pairs in $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$ to a text vector by applying the bag-of-words model [26]. We then convert the text vector obtained into a numeric feature vector by applying the term-frequency inverse document-frequency (TF-IDF) model [27]. In the feature vector obtained, the TF-IDF weight of an action-resource pair in $\mathcal{L}(\langle \mathcal{S}, \mathcal{I} \rangle)$ indicates its relative importance in identifying $\langle \mathcal{S}, \mathcal{I} \rangle$. Finally, we use Gradient Boosting (GB) to predict the semantic relatedness of $\langle \mathcal{S}, \mathcal{I} \rangle$ by taking its feature vector as input. As evaluated in Section IV, GB is the best performer among a large number of learning-based classifiers considered.

IV. Evaluation

We evaluate the effectiveness of APICog by addressing the following three research questions (RQs):

- **RQ1.** Is APICog effective in assessing description-to-permission fidelity in Android apps measured in terms of three metrics, accuracy, precision and recall?

- **RQ2.** Is Gradient Boosting (GB) more effective than other learning-based algorithms in APICOG?
- **RQ3.** Is APICOG more effective than the prior static approaches (reshaped in our setting)?

A. Implementation

APICOG consists of the three components shown in Figure 2. We describe how we have implemented each below.

To perform our dynamic analysis, we use DROIDBOT [24], an automatic input generation tool, to execute an app in order to construct its UI state transition graph. In addition, we have implemented our sensitive API monitor in the Xposed framework [28], a popular code-injection framework for rooted Android devices. Our monitor can hook arbitrary functions of the app (in Java) at runtime without the need to modify the behaviors of the system or app, by being able to run on all types of Android devices supported by the Xposed framework.

For our semantic extraction component, we have implemented it in Python. We have developed a crawler [39] to crawl an app's description from the app market and the relevant API documentation for all the 159 APIs monitored (Table II). We use mtranslate [35], which wraps the Google Translate API, to translate non-English texts into English. We use Stanford Parser [18] to cleanse these raw texts and transform them into a set of valid verb-noun, i.e., action-resource pairs. To find the synonyms for these verbs and nouns, we resort to Word2Vec in the Gensim library [40]. Based on the sensitive words thus found, we have written scripts to parse the UI layout files generated during the dynamic analysis and extract their active-resource pairs. To extract texts from an image, we rely on pytesseract [37], a Google's tesseract-OCR engine.

For correlation inference, we use SCIKIT-LEARN [25], a free Python machine learning library, which integrates all the machine learning algorithms used in our evaluation.

B. Experiment Setup

Our desktop computer runs on a 64-bit Ubuntu 16.04 with 8 cores (3.20GHz Intel Xeon(R) CPU) and 256GB RAM. All apps are performed on a real rooted Android device, namely a Galaxy S5 smartphone. We use a physical device to bypass anti-analysis, since malware often includes checking code to prevent malicious sensitive data access when the app is being experimented on emulator environments [41].

To analyze an app, we use DROIDBOT to generate 1000 events, including clicks, touches, and system-level events. According to prior studies [32], [42], 500 events are enough to provide effective coverage in identifying sensitive behaviors.

C. Android Apps

We have selected a large number of benign and malware apps, totaling 3127. For benign apps, we use all the 1627 Android apps (latest versions) from F-Droid, a popular Android app repository [20]. For malware, we select a set of 1500 apps from Drebin, a popular Android malware dataset [19] (used widely in the literature [43]–[47]), by sampling randomly but proportionally from its 179 malware families.

By running our dynamic analysis on these apps (with 1000 events generated by DROIDBOT per app), we have obtained 5099 state-call pairs, with 805 residing in 251 out of 1627 benign apps and 4294 residing in 725 out of 1500 malware apps. Note that doubling the number of events generated per app does not alter the number of state-call pairs found much.

For the 5099 state-call pairs, whether a sensitive API call can be made legitimately under a UI state is subjective due to the lack of ground truth. Thus, manual labeling is needed. To this end, we invited three participants who are familiar with Android programming to annotate each state-call pair $\langle \mathcal{S}, \mathcal{I} \rangle$ as being positive or negative, based on the app's description, the $\mathcal{I}$-related API documentation, and the action-resource pairs extracted from $\mathcal{S}$. In the end, a pair $\langle \mathcal{S}, \mathcal{I} \rangle$ is considered as being positive if at least two participants agree with this outcome and negative otherwise. In practice, the three participants have annotated the 5099 pairs quite consistently.

Table III shows the ground truth thus obtained. Among the 805 pairs for the benign apps, 696 are positive and 109 are negative. As for the 4294 pairs for the malicious apps, 191 are positive and 4103 are negative. These results show intuitively that illegitimate sensitive API calls mostly appear in malware.

Finally, we divide the set of 5099 state-call pairs roughly equally into a training data set and a test data set. To obtain a balanced training data set [48], [49], we select randomly training state-call pairs so that they come from roughly half of the benign apps and roughly half of the malware apps. This yields a training data set consisting of 2549 state-call pairs (with 395 from benign apps and 2154 from malware apps) and a test data set consisting of the remaining 2550 pairs (with 410 from benign apps and 2140 from malware apps).

D. RQ1: Accuracy, Precision and Recall

Based on the manually annotated ground truth, we measure the true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) produced by a tool for assessing description-to-permission fidelity as follows [4], [7]:

- *TP*: A state-call pair correctly identified as being positive.
- *FP*: A state-call pair incorrectly identified as being positive.
- *TN*: A state-call pair correctly identified as being negative.
- *FN*: A state-call pair incorrectly identified as being negative.

The following four metrics are widely used:

$$\begin{aligned} \mathit{Precision} &= \frac{TP}{TP+FP} \\ \mathit{Recall} &= \frac{TP}{TP+FN} \\ \mathit{Accuracy} &= \frac{TP+TN}{TP+FP+TN+FN} \\ F1\ \mathit{Score} &= \frac{2 \times \mathit{Precision} \times \mathit{Recall}}{\mathit{Precision} + \mathit{Recall}} \end{aligned}$$

Table IV gives the results of APICOG for classifying the 2550 state-call pairs in the test data set. Overall, APICOG has achieved an accuracy of 97.7%, a precision of 94.1%, and a recall of 92.8%. For either benign or malware apps alone, we have also given the performance measurements of APICOG in terms of the same three metrics. APICOG's accuracy (98.6%)

TABLE III: Manually-annotated ground truth.

App Category	# of Apps	# of Apps with State-Call Pairs	# of State-Call Pairs	# of Positive State-Call Pairs	# of Negative State-Call Pairs
Benign	1627	251	805	696	109
Malware	1500	725	4294	191	4103
Total	3127	976	5099	887	4212

TABLE IV: Overall performance of APICOG on the test data set consisting of 2550 state-call pairs against the ground truth.

App Category	# of State-Call Pairs (in the Test Data Set)	TP	FN	FP	TN	Accuracy	Precision	Recall	F1 Score
Benign	410	339	12	15	44	93.4%	95.8%	96.6%	96.2%
Malware	2140	72	20	11	2037	98.6%	86.7%	78.3%	82.3%
Total	2550	411	32	26	2081	97.7%	94.1%	92.8%	93.4%

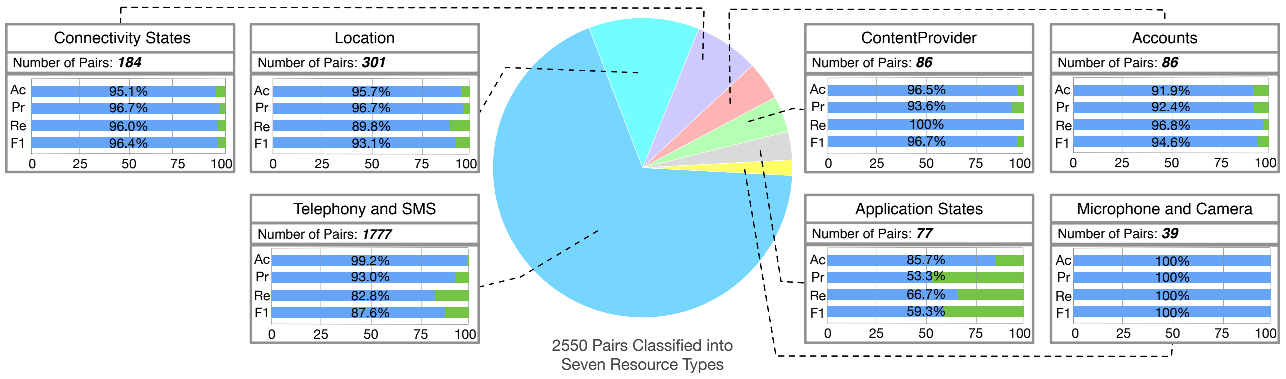


Fig. 3: Performance of APICOG for seven different resource types, i.e., seven different types of APIs monitored as classified in Table II. As notational shorthands, Ac, Pr, Re, and F1 denote accuracy, precision, recall, and F1 score, respectively.

for malicious apps is slightly higher than APICOG's accuracy (93.4%) for benign apps, due to mainly the existence of a larger number of negative pairs in malicious apps. On the other hand, APICOG achieves a higher precision and a higher recall for benign apps (95.8% and 96.6%, respectively) than for malicious apps (86.7% and 78.3%, respectively), due to mainly the existence of a larger number of positive pairs in benign apps. As a result, the F1 scores for benign apps, malware apps, and both together are 96.2%, 82.3% and 93.4%, respectively. Such inconsistent results between benign and malicious apps are well-known in the literature, as they are caused mainly by imbalanced training sets [48], [49].

To analyze APICOG further, Figure 3 illustrates the effectiveness of APICOG in handling seven types of sensitive APIs given in Table II. For "Microphone and Camera" (with the smallest number of state-call pairs), APICOG achieves an accuracy of 100%, the highest among the seven resource types, with neither false positives nor false negatives. This suggests that the texts extracted for this type of API calls can accurately reflect their purposes. In the case of "ContentProvider" and "Accounts", which both have 86 pairs each, APICOG achieves a higher accuracy in the former case, since the former happens to have a more balanced training data set. With "Microphone and Camera" and "ContentProvider" excepted, the larger the training data set for a resource category is, the better the accuracy achieved. In the case of "Application States", APICOG obtains a fairly low precision (53.3%) and recall (66.7%), since the number of its positive pairs, totaling 12, is small.

Finally, let us examine "Telephony and SMS" in more detail. For this resource type, APICOG achieves the second highest accuracy (after "Microphone and Camera" (100%)) and the second lowest F1 score (above "Application States" (59.3%)). Of the seven resource types, "Telephony and SMS" contains the largest number of state-call pairs, totaling 1777, with 64 positive and 1713 negative. APICOG has classified them into FP: 53, FN: 11, FP: 4 and TN: 1709. For the 1713 negative pairs, 1709 are correctly identified as being negative. For the 64 positive pairs, however, only 53 are correctly identified as being positive (implying that the corresponding calls are regarded as legitimate) but 11 still as negative (implying that the corresponding calls are regarded as illegitimate conservatively). As a result, APICOG achieves a high accuracy and a high precision, but a low recall and a low F1 score.

E. RQ2: Learning Effectiveness

Let us justify why we have applied Gradient Boosting (GB) in APICOG. During the course of this research, we have also experimented with nine other machine learning algorithms, Support Vector Machine (SVM), Neural Net (NN), Logistic Regression (LR), Gaussian Naive Bayes (GNB), Adaptive Boosting (AB), Gaussian Process (GP), K-Nearest Neighbors (KNN), Random Forest (RF), Decision Tree (DT).

As shown in Figure 4, APICOG's accuracy (97.7%), precision (94.1%) and recall (92.8%) under GB are the best. With GB and SVM excepted, APICOG achieves less than 89% in either its precision or recall under every other algorithm. SVM

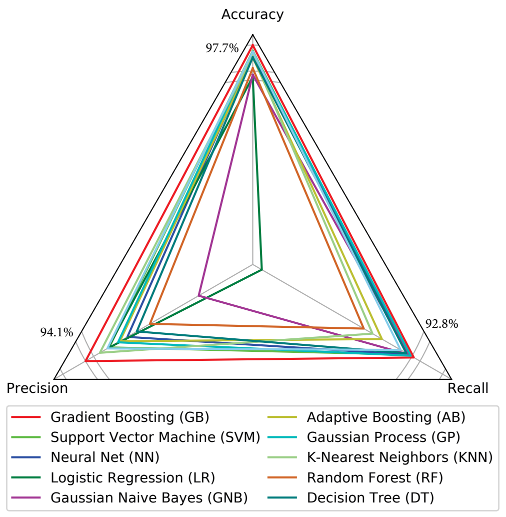


Fig. 4: APICog under GB and nine other learning algorithms (measured in terms of accuracy, precision and recall).

TABLE V: Comparing APICog with prior work.

Tool	Accuracy	Precision	Recall	F1 Score
APICog$_{\mathrm{des}}$	94.8%	91.4%	77.2%	83.7%
APICog$_{\mathrm{asd}}$	95.7%	91.1%	83.5%	87.2%
APICog	97.7%	94.1%	92.8%	93.4%

is the second best, enabling APICog to achieve an accuracy of 96.9%, a precision of 90.3% and a recall of 92.3%.

F. RQ3: Comparing with the Prior Work

As discussed in Sections I and II, there are two types of existing techniques, which are all static, for assessing description-to-permission fidelity. AutoCog [4] is a representative of app-level techniques [2]–[6] while AsDroid [8] is a representative of API-level techniques [7], [8].

As AutoCog and AsDroid are static (without being open-sourced) and APICog is the first dynamic solution, we have decided to compare APICog with both indirectly in our setting as follows. We consider two configurations of APICog. APICog$_{\mathrm{des}}$ is configured from APICog by considering only the app descriptions and API documentation. Note that this configuration is always more effective than AutoCog, which considers only the app descriptions. APICog$_{\mathrm{asd}}$ is configured from APICog by considering only the API documentation and the UI states/contexts for the state-call pairs being analyzed. Note that this configuration is expected to be more effective than AsDroid [8] for two reasons. First, we apply APICog$_{\mathrm{asd}}$ to the same set of state-call pairs discovered dynamically by APICog, but many of these pairs are expected to be missed by AsDroid (as illustrated in Figure 1). Second, AsDroid relates a call to its UI state by considering only the "text" attribute but APICog$_{\mathrm{asd}}$ considers all the four attributes described in Section III-B3.

Table V shows that APICog is better than APICog$_{\mathrm{asd}}$, which is better than APICog$_{\mathrm{des}}$, in each of the four metrics measured (as expected). Given that the API documentation is leveraged by both APICog$_{\mathrm{des}}$ and APICog$_{\mathrm{asd}}$, this means that exploiting additionally the information from call-related UI states is more effective than just relying on additionally the app descriptions. In addition, considering both types of additional information is even more effective.

G. Case Studies

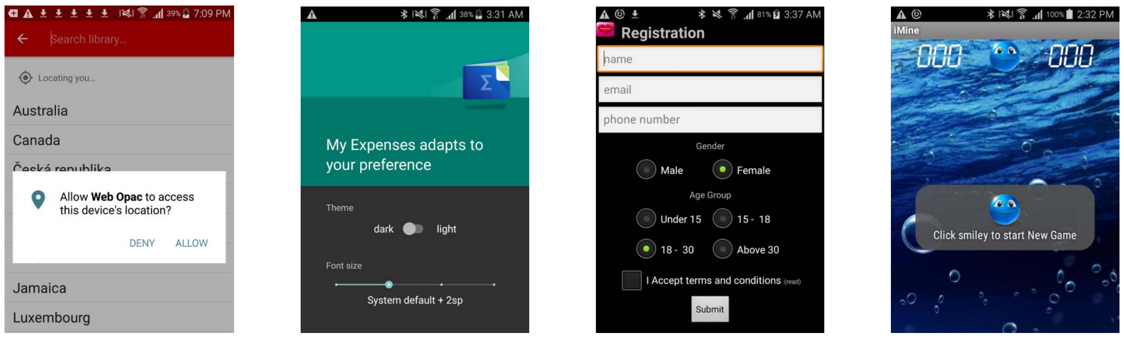
Let us examine four types of state-call pairs analyzed by APICog in the test data set, as illustrated in Figure 5.

- **A Positive State-Call Pair in a Benign App.** `Web Opac`, shown in Figure 5(a), is a benign app developed to provide access for using academic libraries. This is a pair acquiring the user's location via `requestLocationUpdates()`. However, the app description introduces only the library usage but nothing related to the location access. As the UI context extracted from this call includes "access location" and "locating", APICog has correctly classified the pair as positive. This illustrates the importance of leveraging UI contexts in addition to the app description for vetting purposes.
- **A Negative State-Call Pair in a Benign App.** `My Expenses`, shown in Figure 5(b), is a benign app, which leaks ISO country codes via `getNetworkCountryIso()` to the developers. The app describes how to track the user's expenses and the corresponding UI context includes texts such as "My expenses adapts to your preference", "theme" and "Font size" that are irrelevant to the call. Thus, APICog can only correlate the call with its UI context negatively.
- **A Positive State-Call Pair in a Malware App.** `Sweet Heart`, shown in Figure 5(c), is a malicious app. However, this state-call pair, under which `getLine1Number()` is called, is positive and identified as such correctly by APICog. The phone number provided is used to register for a new account. Thus, this call correlates positively with its UI context, with its extracted texts including "phone number".
- **A Negative State-Call Pair in a Malware App.** `iMine`, shown in Figure 5(d), is a malicious app. In this negative pair, `sendTextMessage()` is called to leak the information by SMS without the user's knowledge. This happens when the user clicks the "Click smiley to start New Game" button in its UI context, which includes only this text and "iMine". So APICog has classified this as negative correctly.

H. Limitations

While already more effective than the prior work in assessing description-to-permission fidelity, APICog can still be improved in a number of aspects. First of all, APICog is a dynamic analysis, as it relies on DroidBot [24] to discover all the sensitive API calls made in an app dynamically. To improve code coverage, we can combine static and dynamic analyses to discover more state-call pairs to be analyzed.

APICog correlates a sensitive API call with its UI context (explicitly available) by exploiting the description of an app for some justification information. Such app descriptions (as required by, e.g., Google Play) are assumed to be accurate, as the apps with misleading descriptions are usually removed quickly from an app repository. Correlating sensitive API calls



(a) Web Opac (b) My Expenses (c) Sweet Heart (d) iMine

Fig. 5: Screenshots of four apps in the case studies.

with some implicit UI contexts (e.g., font sizes, colors, and UI element blocking) will be investigated in future work.

In addition to "text", "resource_id", "class name" and "view_str", more attributes may be introduced, if needed.

Currently, APICog exploits the semantic relatedness between an API call and a UI state only. How to relate an API call to a possibly non-UI state can be similarly exploited.

Our sensitive API monitor is implemented in the Xposed framework [28] but Xposed can currently hook Java code only. However, how to extend Xposed to discover sensitive API calls executed in non-Java code can be pursued orthogonally.

Finally, the Android system keeps evolving [50], [51]. However, except for data relabeling, which will be required as in any ML-based approach, APICog grows easily as its text selection and matching tasks are all done automatically by applying NLP and machine learning techniques.

V. Related Work

NLP Techniques for Android. NLP techniques have been used in performing semantic analysis for improving the security of Android apps. Whyper [3] and AutoCog [4] relate the textual description of an app to the permissions requested. CHABADA [2] goes slightly beyond by checking if the permission for a particular API call is consistent with the app description. TAPVerifier [6] makes improvements by also exploiting an app's privacy policy. These tools are too coarse-grained, as they cannot conclude if an API call is legitimate or not under a certain runtime context.

AsDroid [8] applies static analysis to determine if a sensitive operation matches the "text" attribute of its related UI component. FlowCog [7] introduces a flow-level system to extract and analyze semantics for each information flow [10]–[15] of an Android app based on existing static analysis frameworks [10], [52]. FlowCog first applies FlowDroid [10] to discover all information flows. FlowCog then conducts a data-flow analysis to find the views related to a given flow. Finally, FlowCog correlates a given flow with the semantics extracted from its corresponding views. However, static analyses for Android apps are usually imprecise, especially in the presence of dynamic language features, such as dynamic class loading, reflection, code encryption, and code obfuscation.

Unlike these earlier static techniques, APICog is dynamic by correlating a sensitive API call with a UI state (i.e., its UI context) and exploiting a number of attributes in its UI layout whose values often become known at runtime.

Taint Analysis. Both static [10]–[16] and dynamic [53], [54] analysis approaches have been proposed to detect information leaks. FlowDroid [10] focuses on tracking intra-component data flows while AmandRoid [13], DroidSafe [11], IccTA [12] and DIALDroid [14] consider also inter-component data-flows. Apart from these static tools, some dynamic tools also exist. TaintDroid [53] performs real-time taint tracking in a customized Android framework. On the other hand, Uranine [54] can detect information leaks by instrumentation without modifying the Android system.

However, these tools aim to find information flows in an app, without understanding whether they are legitimate or not. In contrast, APICog assesses whether a given API call is legitimate or not under a particular runtime context.

GUI Testing. Google's Monkey [55] and DynoDroid [56] perform automated app testing, with randomly generated test inputs. To improve their coverage via optimizing test sequences, Sapienz [57] embraces multi-objective search-based testing while GUICC [58] and Stoat [59] resort to automated model-based testing. PUMA [60] is a programmable UI automation framework for implementing various state-based test strategies. These tools cannot be used directly for detecting illegal API calls. DroidBot [24] represents a UI-guided input generation tool based on a state transition model. Therefore, DroidBot can work with obfuscated or encrypted code, by building the UI state transition graph for an app on the fly.

VI. Conclusion

We have introduced APICog, a new fine-grained API-level framework for assessing description-to-permission fidelity in Android apps dynamically. We have evaluated APICog with a large number of benign and malware apps, achieving an accuracy of 97.7%, a precision of 94.1%, and a recall of 92.8% overall. APICog is shown to outperform the prior work (evaluated in our experimental setting).

Acknowledgement

We wish to thank all the reviewers for their comments and feedback. This research is supported by Australian Research Grants ((DP170103956 and DP180104069).

References

[1] IDC, "Smartphone os market share, q2 2020," https://www.idc.com/promo/smartphone-market-share/os, 2020.

[2] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1025–1035.

[3] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, "WHYPER: Towards automating risk assessment of mobile applications," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 527–542.

[4] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "AutoCog: Measuring the description-to-permission fidelity in Android applications," in *Proceedings of the Conference on Computer and Communications Security*, ser. CCS. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1354–1365.

[5] S. Zimmeck, Z. Wang, L. Zou, R. Iyengar, B. Liu, F. Schaub, S. Wilson, N. Sadeh, S. Bellovin, and J. Reidenberg, "Automated analysis of privacy requirements for mobile apps," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS, San Diego, California, USA, 2017.

[6] L. Yu, X. Luo, C. Qian, S. Wang, and H. K. N. Leung, "Enhancing the description-to-behavior fidelity in Android apps with privacy policy," *IEEE Transactions on Software Engineering*, vol. 44, no. 9, pp. 834–854, 2018.

[7] X. Pan, Y. Cao, X. Du, B. He, G. Fang, R. Shao, and Y. Chen, "FlowCog: Context-aware semantics extraction and analysis of information flow leaks in Android apps," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, pp. 1669–1685.

[8] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 1036–1046.

[9] F. Pauck, E. Bodden, and H. Wehrheim, "Do Android taint analysis tools keep their promises?" in *Proceedings of the 26th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE. New York, NY, USA: Association for Computing Machinery, 2018, pp. 331–341.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proceedings of the 35th Conference on Programming Language Design and Implementation*, ser. PLDI. New York, NY, USA: Association for Computing Machinery, 2014, pp. 259–269.

[11] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of Android applications in DroidSafe," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS, San Diego, California, USA, 2015.

[12] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE. Florence, Italy: IEEE Press, 2015, pp. 280–291.

[13] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," *ACM Trans. Priv. Secur.*, vol. 21, no. 3, 2018. [Online]. Available: https://doi.org/10.1145/3183575

[14] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Android collusive data leaks with flow-sensitive DIALDroid dataset," 2017.

[15] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd International Workshop on the State of the Art in Java Program Analysis*, ser. SOAP. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1–6.

[16] D. He, H. Li, L. Wang, H. Meng, H. Zheng, J. Liu, S. Hu, L. Li, and J. Xue, "Performance-boosting sparsification of the IFDS algorithm with applications to taint analysis," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 267–279.

[17] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 176–186.

[18] https://github.com/dasmith/stanford-corenlp-python, 2020.

[19] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of Android malware in your pocket," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS, San Diego, California, USA, 2014, pp. 23–26.

[20] F-Droid, *Free and Open Source App Repository*, 2018, https://f-droid.org/.

[21] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang, "Understanding Android obfuscation techniques: A large-scale investigation in the wild," in *Proceedings of the International Conference on Security and Privacy in Communication Systems*, ser. SecureComm. Singapore: Springer, 2018, pp. 172–192.

[22] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. dAmorim, and M. D. Ernst, "Static analysis of implicit control flow: Resolving Java reflection and Android intents (t)," in *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, ser. ASE, Washington, DC, USA, 2015, pp. 669–679.

[23] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "DroidRA: Taming reflection to support whole-program analysis of Android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA. New York, NY, USA: Association for Computing Machinery, 2016, pp. 318–329.

[24] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: A lightweight ui-guided test input generator for Android," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C. Buenos Aires, Argentina: IEEE, 2017, pp. 23–26.

[25] https://scikit-learn.org/stable/, 2020.

[26] https://en.wikipedia.org/wiki/Bag-of-words_model, 2020.

[27] http://www.tfidf.com/, 2020.

[28] https://repo.xposed.info/module/de.robv.android.xposed.installer, 2020.

[29] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Android permissions: A perspective combining risks and benefits," in *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, ser. SACMAT '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 13–22.

[30] J. Liu, D. Wu, and J. Xue, "TDroid: Exposing app switching attacks in Android with control flow specialization," in *Proceedings of the 33rd International Conference on Automated Software Engineering*, ser. ASE. New York, NY, USA: Association for Computing Machinery, 2018, pp. 236–247.

[31] C. Ren, P. Liu, and S. Zhu, "WindowGuard: Systematic protection of gui security in Android," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS, San Diego, California, USA, 2017.

[32] Y. Li, F. Yao, T. Lan, and G. Venkataramani, "SARRE: Semantics-aware rule recommendation and enforcement for event paths on Android," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 12, pp. 2748–2762, 2016.

[33] Y. Li, F. Chen, T. J.-J. Li, Y. Guo, G. Huang, M. Fredrikson, Y. Agarwal, and J. I. Hong, "Privacystreams: Enabling transparency in personal data processing for mobile apps," *ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 3, pp. 76:1–76:26, 2017.

[34] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: automatically detecting potential privacy leaks in Android applications on a large scale," in *Proceedings of the International Conference on Trust and Trustworthy Computing*, ser. TRUST, Berlin, Heidelberg, 2012, pp. 291–307.

[35] https://github.com/mouuff/mtranslate, 2020.

[36] Google, "Android framwork class index," https://developer.android.com/reference/classes.html, 2019.

[37] https://pypi.org/project/pytesseract/, 2020.

[38] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems*, ser. NIPS. Lake Tahoe, Nevada: Curran Associates, Inc., 2013, pp. 3111–3119.

[39] https://www.crummy.com/software/BeautifulSoup/bs4/doc/, 2020.

[40] https://radimrehurek.com/gensim/models/word2vec.html, 2020.

[41] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in Android applications that feature anti-analysis techniques," in *Proceedings of the Network and Distributed System Security Symposium*, ser. NDSS, San Diego, California, USA, 2016.

[42] Z. Meng, Y. Xiong, W. Huang, F. Miao, T. Jung, and J. Huang, "Divide and conquer: Recovering contextual information of behaviors in Android apps around limited-quantity audit logs," in *Proceedings of the 41st International Conference on Software Engineering: Companion*

Proceedings, ser. ICSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, p. 230–231.

[43] Z. Zhu and T. Dumitraş, "FeatureSmith: Automatically engineering features for malware detection by mining the security literature," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 767–778.

[44] W. Yang, D. Kong, T. Xie, and C. A. Gunter, "Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ser. ACSAC 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 288–302.

[45] M. Hammad, J. Garcia, and S. Malek, "A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 421–431.

[46] H. Cai, N. Meng, B. Ryder, and D. Yao, "DroidCat: Effective Android malware detection and categorization via app-level profiling," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1455–1470, 2019.

[47] S. Chen, M. Xue, L. Fan, S. Hao, L. Xu, H. Zhu, and B. Li, "Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach," *computers & security*, vol. 73, pp. 326–344, 2018.

[48] Y.-W. Chen and C.-J. Lin, *Combining SVMs with Various Feature Selection Strategies*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 315–324.

[49] Z. Wang, "Practical tips for class imbalance in binary classification," https://towardsdatascience.com/practical-tips-for-class-imbalance-in-binary-classification-6ee29bcdb8a7, 2018.

[50] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and detecting evolution-induced compatibility issues in android apps," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 167–177.

[51] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 1–41, 2017.

[52] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13–23.

[53] W. Enck, P. Gilbert, S. Han, V. Tendulkar, 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," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, 2014. [Online]. Available: https://doi.org/10.1145/2619091

[54] V. Rastogi, Z. Qu, J. McClurg, Y. Cao, and Y. Chen, "Uranine: Real-time privacy leakage monitoring without system modification for Android," in *Proceedings of the International Conference on Security and Privacy in Communication Systems*, ser. SecureComm, Dallas, United States, 2015, pp. 256–276.

[55] Google, "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey.html, 2019.

[56] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE. New York, NY, USA: Association for Computing Machinery, 2013, pp. 224–234.

[57] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA, Saarbr&#252;cken, Germany, 2016, pp. 94–105.

[58] Y.-M. Baek and D.-H. Bae, "Automated model-based Android gui testing using multi-level gui comparison criteria," in *Proceedings of the 31st International Conference on Automated Software Engineering*, ser. ASE. New York, NY, USA: Association for Computing Machinery, 2016, pp. 238–249.

[59] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of Android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 245–256.

[60] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, ser. MobiSys. New York, NY, USA: Association for Computing Machinery, 2014, pp. 204–217.