Self-Hiding Behavior in Android Apps: Detection and Characterization

Zhiyong Shan
Wichita State University
Wichita, Kansas, USA
zhiyong.shan@wichita.edu

Iulian Neamtiu
New Jersey Institute of Technology
Newark, New Jersey, USA
ineamtiu@njit.edu

Raina Samuel
New Jersey Institute of Technology
Newark, New Jersey, USA
res9@njit.edu

ABSTRACT

Applications (apps) that conceal their activities are fundamentally deceptive; app marketplaces and end-users should treat such apps as suspicious. However, due to its nature and intent, activity concealing is not disclosed up-front, which puts users at risk. In this paper, we focus on characterization and detection of such techniques, e.g., hiding the app or removing traces, which we call "*self hiding behavior*" (SHB). SHB has not been studied per se – rather it has been reported on only as a byproduct of malware investigations. We address this gap via a study and suite of static analyses targeted at SH in Android apps. Specifically, we present (1) a detailed characterization of SHB, (2) a suite of static analyses to detect such behavior, and (3) a set of detectors that employ SHB to distinguish between benign and malicious apps. We show that SHB ranges from hiding the app's presence or activity to covering an app's traces, e.g., by blocking phone calls/text messages or removing calls and messages from logs. Using our static analysis tools on a large dataset of 9,452 Android apps (benign as well as malicious) we expose the frequency of 12 such SH behaviors. Our approach is effective: it has revealed that malicious apps employ 1.5 SHBs per app on average. Surprisingly, SH behavior is also employed by legitimate ("benign") apps, which can affect users negatively in multiple ways. When using our approach for separating malicious from benign apps, our approach has high precision and recall (combined F-measure = 87.19%). Our approach is also efficient, with analysis typically taking just 37 seconds per app. We believe that our findings and analysis tool are beneficial to both app marketplaces and end-users.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; **Software reverse engineering**; • **Software and its engineering → Automated static analysis**;

KEYWORDS

Android, static analysis, malware, mobile security

ACM Reference Format:
Zhiyong Shan, Iulian Neamtiu, and Raina Samuel. 2018. Self-Hiding Behavior in Android Apps: Detection and Characterization. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180214

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180214

1 INTRODUCTION

Mobile security research has mostly focused on malware activation, malicious payloads, permission abuse, or leaking sensitive data. Little attention has been paid to deceptive mechanisms that are essential for the success of malware, i.e., how malware manages to get installed, and continues operating on the phone without the users noticing anything suspicious. To do so, malware uses a range of SHB, e.g., hiding the app, hiding app resources, blocking calls, deleting call records, or blocking and deleting text messages. Surprisingly, extremely popular "benign" apps such as Airbnb, Truecaller, and Waze also employ certain SH techniques in the name of user convenience.

We believe that SHB is fundamentally deceptive and that having tools that perform accurate and early detection of SHB is key. First, app marketplaces, e.g., Google Play or Apple Store, should be able to detect SHB, so that SHB can be considered in the decision to publish an app or not. Even when an app with SHB is published on the marketplace, users should be forewarned about the SHB so they can decide whether to install the app on their phone or not.

We address these problems on the Android platform via several advances: (1) we shine a light on SHB via detailed characterization, (2) we construct an SHB-detecting tool based on static analysis,[1] and (3) we show how our approach for identifying SHB can be very effective at exposing malicious apps as well as deceptive practices in benign apps. We chose to focus on Android because Android clearly dominates the worldwide mobile market, with an 87% market share in the second quarter of 2017 [8]. While popular, Android's security could be improved: researchers from the antivirus firm G Data have discovered that more than 750,000 new malicious Android apps have sprung out during 2017's first quarter, and estimate that the total number will grow up to a staggering 3.5 million by the end of 2017 [1]. Therefore, there is plenty of evidence that the threat level for Android users remains high, and there is an impetus to detect and weed out malware before it is published on Google Play or it reaches users' devices.

We start by presenting a detailed characterization of SHB in Section 2. We group SHB into three main categories: SHB that involve app objects, i.e., hiding the presence of the app; SHB that block or remove traces of remote communication, e.g., blocking calls or deleting text messages; and subverting the system's reminders, e.g., hiding notifications or muting the phone.

[1] The tool and datasets are available at http://spruce.cs.ucr.edu/SelfHiding/

In Section 3 we present our approach: a tool, consisting of a suite of static analyses, that exposes potential SHB in a given Android app. Our tool works directly on APKs, i.e., the format Android apps are distributed in, and does not require access to app source code.

In Section 4 we evaluate our approach along several dimensions. First, we check our approach's accuracy via manual validation on a set of 198 malware samples; we found that it attains an 85.71% F-measure. Second, we check whether our approach can be used to triage benign from malicious apps: using a dataset of 9,452 benign and malicious apps, the attained F-measure is 87.19%.

In Section 5 we provide a detailed exposé of SHB in widely-used apps, and how these deceptive behaviors can affect users.

To summarize, we make the following contributions:

(1) An exposé of of SHBs, including novel SHBs, as employed by malware and widely-used benign apps.
(2) A static-analysis-based approach for detecting SHB.
(3) An evaluation of our approach on 9,452 sample apps, both benign and malicious.

2 SELF-HIDING BEHAVIORS

In this section, we provide a comprehensive description of SH behaviors. We define as *SH* a behavior meant to hide the app or its actions from being viewed (or heard!) *by the user*. Note that we exclude those behaviors meant to evade security mechanisms, e.g., anti-malware tools or access control mechanisms – they have been studied thoroughly and are outside the scope of this paper.

Our characterization is based on manual analysis of about 200 malicious apps and automated analysis of about 3,000 other malicious apps. We found 12 SHBs; few of these are even mentioned in the research community, let alone characterized thoroughly, and some, including "Hide icon" and "Hide activity", are not mentioned at all.

Users could employ three main approaches for identifying the presence of malicious apps: inspecting app objects (icon, app, activity), analyzing remote communication (SMS, MMS, and phone calls) or checking system reminders (system dialogs, sound, system logs, notifications, recent apps list, etc). There are two main issues with this approach, though: (1) it requires a highly knowledgeable user who performs such inspections periodically, and (2) malware actively attempts to escape (hide itself) from such identification.

To set up the discussion, in Figure 1 we show the number of SHBs in sample sets of 1,000 malicious and 1,000 benign apps, respectively; the 1,000 benign apps are a random sample extracted from the 6,233 benign apps, while the 1,000 malicious apps are a random sample extracted from the 3,219 malicious apps (a more thorough dataset description is provided in Section 4).

2.1 App Objects

2.1.1 *Hide icon.*

After installation, benign apps add their icon to the home screen. To hide itself, a malicious app removes the icon so the user cannot notice the app's presence. There are two methods for hiding the icon:

(a) Modifying the app's manifest file to remove the app from the default launcher, i.e., home screen. This can be done by deleting category android.intent.category.LAUNCHER from the app's main activity section in the manifest file.[2] For example, malware Fake-skype camouflages as the popular app Skype and runs in the background without an icon in the home screen.

(b) Calling an Android API method to disable the icon at runtime. This can be done by invoking method setComponentEnabledSetting(). For example, malware Facebook-otp (full package name: jgywwv.jvyjsd.sordvd), masquerades as the Facebook app but disables its icon immediately after installation. We show the segment of the code we reverse-engineered from this malware:

```
PackageManager pm = getPackageManager();
ComponentName cn = new ComponentName("jgywwv.jvyjsd.sordvd", "
        jgywwv.jvyjsd.sordvd.Activity1");
pm.setComponentEnabledSetting(cn, PackageManager.
        COMPONENT_ENABLED_STATE_DISABLED, PackageManager.
        DONT_KILL_APP);
```

2.1.2 *Hide app.*

When benign apps are running, they typically show up in the running app list. In contrast, a malicious app can run as a service, in the background, hence does not show up in the list. In order to automatically start the malware as a service without the user clicking the icon, a malicious app creates a BroadcastReceiver class and registers it to receive certain events like SMS_RECEIVED, BOOT_COMPLETED, etc. After receiving one of the registered events, the malware's BroadcastReceiver launches the malware as a service in the background. As a result, the user cannot see the malicious app in the running app list. For example, the spyware Candy_corn automatically records Google Voice calls in the background. As shown in the following code segment, Candy_corn monitors seven kinds of events and starts itself as a service (if the service is not running already):

```
public void onReceive(Context context, Intent intent) {
  ...
  String act = intent.getAction();
  if (Intent.ACTION_BOOT_COMPLETED.equals(act) |
      Intent.SMS_RECEIVED.equals(act) |
      Intent.NEW_OUTGOING_CALL.equals(act) |
      Intent.SCREEN_OFF.equals(act) |
      Intent.PACKAGE_INSTALL.equals(act) |
      Intent.PACKAGE_ADDED.equals(act) |
      Intent.SIG_STR.equals(act)) {
        if (isServiceRunning())
          return;
        Intent serviceIntent = new Intent(context, com.google.progress
                .AndroidClientService.class);
        serviceIntent.setAction("com.google.
                ACTION_START_CALL_RECORD");
        context.startService(serviceIntent);
  }
  ...
}
```

2.1.3 *Hide activity.*

Most malware runs as a background service. However, starting in Android version 3.1, apps cannot create a service without having an activity associated with that service. Therefore, a malicious app must first create an activity. Next, to hide the activity, the app can employ two approaches: making the activity transparent or destroying the activity before it becomes visible.

[2] The manifest file (AndroidManifest.xml), bundled with the app, contains a description of the app's capabilities, system requirements, resources, permissions, etc.

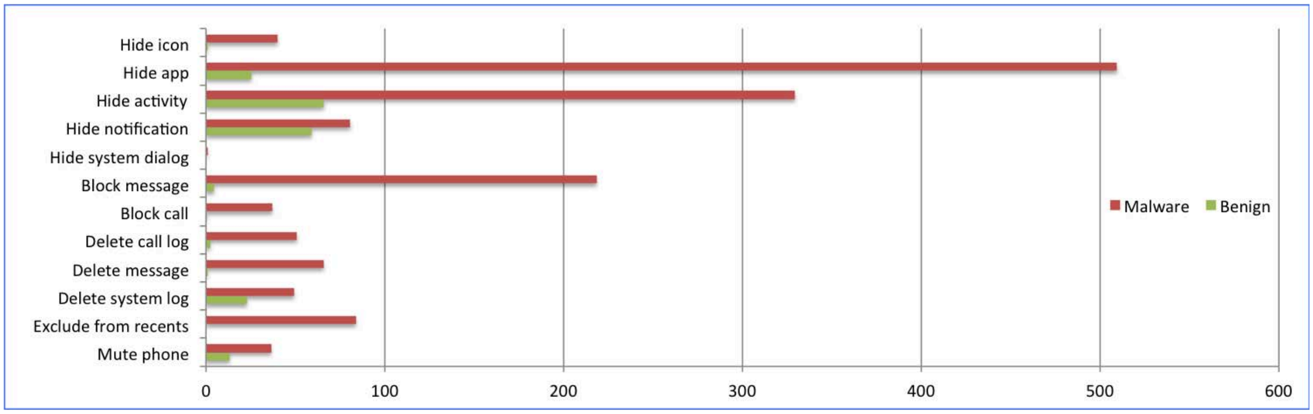


Figure 1: The numbers of SHBs in two sample sets of 1,000 malware apps and 1,000 benign apps, respectively.

To render an activity transparent, a malicious app needs to make the activity's main layout transparent, set the activity as full screen, then remove the action bar and window title. These values can be set in the app's manifest file. In addition, malware can also accomplish this via certain API methods. For example, Android API methods setBackground(), setBackgroundDrawable() or setBackgroundColor() can change the activity to transparent. Methods addFlags(), setFlags (), or requestWindowFeature() can change the window to full screen, as well as remove the action bar and the window title. We now illustrate this with malware DroidKungFu3. First, the manifest file is used to make the activity layout transparent:

```
<style name="Theme.NoTitle" parent="@android:style/Theme">
    <item name="android:windowBackground">@android:color/transparent</item>
    <item name="android:windowNoTitle">true</item>
    <item name="android:windowIsTranslucent">true</item>
    <item name="android:windowContentOverlay">@null</item>
</style>
```

Then, at runtime, the app sets the window flag to FLAG_NOT_TOUCH_MODAL, meaning that even when the window is focusable, it allows any pointer events outside the window to be sent to the windows behind it. As a result, the user cannot see the malware activity but can still use the activity of another app, which is just below the malware activity – de facto the malware activity has successfully inserted itself between the unsuspecting user and the app below:

```
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
   ...
  setContentView(R.layout.activity_main);
  getWindow().addFlags(WindowManager.LayoutParams.
        FLAG_NOT_TOUCH_MODAL);
   ...
}
```

To destroy an activity before showing it, a malicious app will call finish () in one of the three lifecycle callback methods: onCreate(), onStart (), or onResume(). The finish () method in turn calls onDestroy() to finish the activity. If finish () is called in onCreate() or onStart (), the activity is not shown at all. But if it is called in onResume(), the screen will flicker during the activity transition from the foreground to the background. To prevent this, the activity is set to transparent. We show this being accomplished in the SaveMe spyware:

```
protected void onCreate(final Bundle bundle)
{
  super.onCreate(bundle);
   ...
  this.getPackageManager().setComponentEnabledSetting(this.
        getComponentName(), 2, 1);
  // 2 = COMPONENT_ENABLED_STATE_DISABLED; 1 = DONT_KILL_APP
  this. finish ();
   ...}
```

2.2 Remote Communication

2.2.1 *Delete Message.*

Sending SMS/MMS messages furtively, in the background, is a common behavior in malware. Therefore, several anti-malware products focus on this to recognize malware. After sending or receiving SMS/MMS in the background, Android saves a copy of the SMS/MMS in the outbox or inbox, respectively. To cover its tracks, malware needs to delete this copy. The malware usually calls delete () on a content URI, i.e., "content :// sms/inbox/" and "content :// sms/outbox/", respectively. Furthermore, malware can also delete SMS/MMS associated with a certain message ID, time, or phone number. An example is malware XTaoAd.A that deletes a message upon receipt:

```
void onReceive(android.content.Context context,android.content.Intent
      intent){
   ...
  if (android.os.Build.VERSION.SDK_INT >= android.os.Build.
        VERSION_CODES.KITKAT) {
    if (!Telephony.Sms.getDefaultSmsPackage(context).equalsIgnoreCase(
          context.getPackageName())) {
      context.getContentResolver().delete(Uri.parse("content :// sms/"),
            null, null);
       ...
}}}
```

2.2.2 *Delete Call Log.*

After making or receiving a phone call in the background, Android will generate a record in the call log. To cover its traces, malware has to delete this entry from the call log. An example is the SaveMe spyware. SaveMe has a service that can make a call in the background (e.g., to a premium number) as dictated by the malware's server. We show this in Figure 4. In the left code snippet EXT_CALL is the number to be called. After the phone call, the malware deletes the corresponding call log entry. In the right code snippet, the string s contains the number that was just called.

2.2.3 Block Message. After a malicious app sends SMS/MMS to sign up for a premium-rate service in the background, it will receive a confirmation SMS/MMS sent from the service provider. To prevent users from knowing this, the malware has to filter the received SMS/MMS by calling abortBroadcast(). An example is Trojan:Fakebank.B, shown below (please note that method a checks the intent to see whether the SMS message has been received).

```
void onReceive(android.content.Context context, android.content.Intent
        intent) {
    ...
    if (!Telephony.Sms.getDefaultSmsPackage(context).equalsIgnoreCase(
            context.getPackageName())) {
        a(intent.getExtras(), context);
        abortBroadcast();
          ...
} }
```

2.2.4 Block Call. For malware that is part of a botnet, the command and control (C&C) server could call the infected phone to ask the bot (malware app) to perform certain services; the C&C server is encoded in the phone number. To prevent users from realizing this, the malware needs to block the phone call. If the malware received an intent android.intent.action.PHONE_STATE and the number of the caller is the C&C server, then the ringer mode is set to silent to suppress the notification of the incoming call and the phone call is disconnected. Its corresponding entry from the call logs is also removed, removing all traces of the phone call. An example is malware fakeAV that uses endCall() to cancel the incoming call:

```
void onReceive(android.content.Context context, android.content.Intent
        intent) {
    ...
    TelephonyManager tm = (TelephonyManager) context.getSystemService(
            context.TELEPHONY_SERVICE);
    try {
        Class c = Class.forName(tm.getClass().getName());
        Method m = c.getDeclaredMethod("getITelephony");
        m.setAccessible(true);
        com.android.internal.telephony.ITelephony telephonyService = (
                ITelephony) m.invoke(tm);
        telephonyService.endCall();
    } catch (Exception e) {
        e.printStackTrace();
    }
    ...
}
```

2.3 System Reminders

2.3.1 Hide Alert. System dialogs could reveal the presence of malware by displaying alarms, user account balances, or other abnormal behaviors to the user. To avoid this, malware has to dismiss the system dialog by broadcasting the intent ACTION_CLOSE_SYSTEM_DIALOGS, as shown in the following code snippet:

```
public void onWindowFocusChanged(boolean hasFocus) {
    super.onWindowFocusChanged(hasFocus);
    ...
    if (! hasFocus) {
        Intent closeDialog = new Intent(Intent.
                ACTION_CLOSE_SYSTEM_DIALOGS);
        sendBroadcast(closeDialog);
    }
    ...
}
```

2.3.2 Hide Notification. Apps can send alerts to the user by generating a notification on the notification bar. But the malware can delete notifications by calling NotificationManager's methods cancel() or cancelAll() when receiving notifications. An example is malware Bios.NativeMaliciousCode.apk:

```
void clearNotify(android.content.Context context){
    ...
    ((android.app.NotificationManager) context.getSystemService("
            notification")).cancel(1);
    ...
}
```

2.3.3 Mute Phone. To cover their presence, malicious apps often resort to muting the phone or disabling the vibrate function, to prevent the user from hearing the sound of alarms, notifications, phone calls or incoming SMSs. This can be accomplished in a variety of ways: switching to silent mode, calling the vibrator service, setting the phone to mute, or adjusting the volume to the lowest level. An example is the Trojan iBanking:

```
void a(android.content.Context context){
    ...
    ((android.media.AudioManager)context.getSystemService("audio")).
            setStreamMute( AudioManager.STREAM_ALARM, true);
    ...
}
```

2.3.4 Exclude From Recent Apps List. After an app has run, the system puts its activities into the recent apps list. To prevent this, malware can set the flag excludeFromRecents in the manifest file, or by calling ActivityManager.setExcludeFromRecents(). An example is Trojan:Malapp.

```
<activity android:name="com.yangccaa.chengaa.WEYY"
    android:label="@string/notification_name"
    android:taskAffinity=". NotificationActivity "
    android:excludeFromRecents="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
</activity>
```

2.3.5 Delete System Log. Android saves system activity into the system log, which can be viewed via the logcat. Malware can call `adb logcat -c` to delete the logs if the phone is rooted or if the Android API is lower than 16. We present an example of such an action extracted from the trojan SMSblocker:

```
r = getRuntime();
r.exec("logcat -c");
```

3 DETECTING SH BEHAVIORS

Our approach relies on a suite of static analyses to detect SHBs. Figure 2 shows an overview of our tool's design. The input is an APK file (APK is the format Android apps are distributed in). We pass the bytecode to Soot [12]/FlowDroid [13] which perform basic tasks such as alias analysis, call graph analysis, as well as fixpoint computations to deal with loops and recursion. Next, we perform

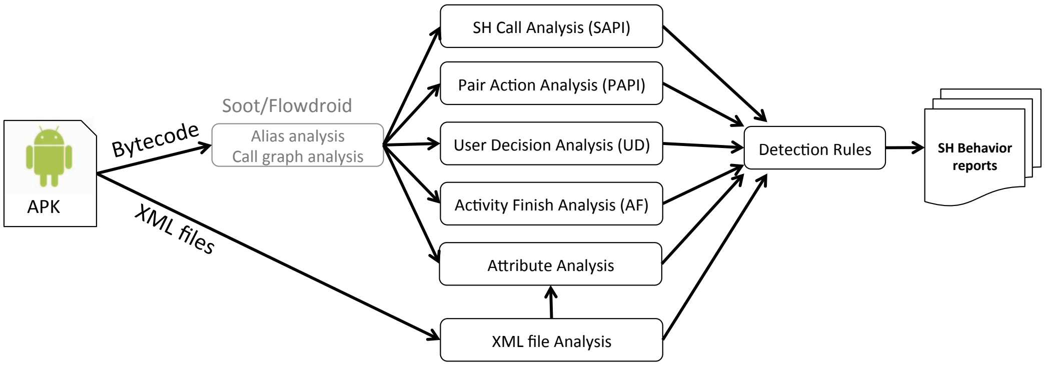


Figure 2: Tool overview.

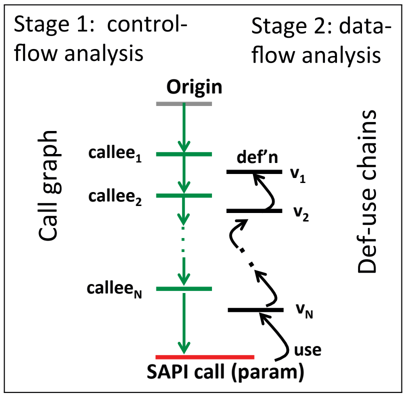


Figure 3: SAPI analysis.

our core analyses (described shortly) on both bytecode and XML files. Finally, a report detailing the potential SHBs is produced.

3.1 Static Analysis

3.1.1 Finding SH Call Invocations (SAPI Analysis). Our first analysis finds whether *SH API calls* are invoked (we name this *SAPI analysis* for short). We present the analysis in Figure 3. Specifically, the analysis starts at an **Origin** (app or activity start). In the first stage, we use control-flow and call graph analysis to find whether a certain **SAPI call** is invoked – green nodes and edges on the left represent methods and call graph edges, respectively. In the second stage, we use backward dataflow analysis to find if the call is invoked with certain SH-indicating **param**eters; more precisely, we walk the def-use chains backwards (shown in black) until we can find the parameter definition, e.g., a constant or an alias. In Table 1 we show the origins and SAPI calls for each SHB. For example, to detect the “Hide app” SHB, our analysis will check whether the call to Context. startService () is reachable when starting in BroadcastReceiver.onReceive(). To check for “Delete message” on the other hand, we start tracking from BroadcastReceiver.onReceive(SMS_RECEIVED/ACTION_VIEW) to see if we can reach ContentResolver.Delete(); next, we walk the def-use chains backwards to see if the argument is "content :// sms". For certain behaviors, e.g., “Hide activity”, we use all the app’s entry points as origin; app entry points are provided by FlowDroid.

We now provide an example from the real malware DroidKungFu1, which deletes all SMS messages. In this case we show simplified disassembled code, from which we have removed irrelevant instructions.

```
1  specialinvoke $r3.<java.lang.StringBuilder: void <init>(java.lang.
         String)>("content :// sms/")
2  $r4 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.String
         toString () >()
3  $r5 = staticinvoke <android.net.Uri: android.net.Uri parse(java.lang.
         String)>($r4)
4  virtualinvoke $r2.<android.content.ContentResolver: int delete(android
         .net.Uri,java.lang.String,java.lang.String []) >($r5, null, null)
```

Let us assume that our analysis has determined that an invocation of ContentResolver.Delete() is reachable from BroadcastReceiver.onReceive(PHONE_STATE). To check the value of Delete()’s parameter, we walk the def-use chains backwards starting at $r5 (that is used at line 4 and defined at line 3). As line 3 calls method parse(), we proceed further on $r4 and then $r3. Eventually on line 1 we see the definition, i.e., "content :// sms/". This concludes our analysis and we report the potential “Delete message” behavior.

3.1.2 Pair Action (PAPI) Analysis. Another broad self-hiding category consists of *pair actions*, where an app first performs a malicious action then deletes traces of this action, e.g., deleting a text message after sending it. Our analysis (we name this *PAPI analysis* for short) detects six types of pair actions: send message/delete message log, receive message/delete message log, receive/block message, make phone call/delete call log, receive phone call/delete call log, receive/block phone call.

Our pair action detector uses data flow analysis in a manner similar to taint analysis to see if data flows from a *pair start* to a *pair end*. The paired methods and the SHB are listed in Table 2. Figure 4 shows a code snippet from the real malware Saveme that deletes a call from the call log. Our tool detects data flow from startActivity () to contentResolver.delete(). Actually EXT_CALL on line 2 (left) and s on line 4 (right) have the same value – the phone number called. Therefore, the pair (Context. startActivity (),contentResolver.delete()), is detected, indicating the SHB “Delete call log”.

3.1.3 User-decision (UD) Analysis. To reduce potential false positives in cases where SAPI methods are also used by benign apps, we perform a *user-decision* analysis that checks whether an API

Table 1: SH call (SAPI) analysis.

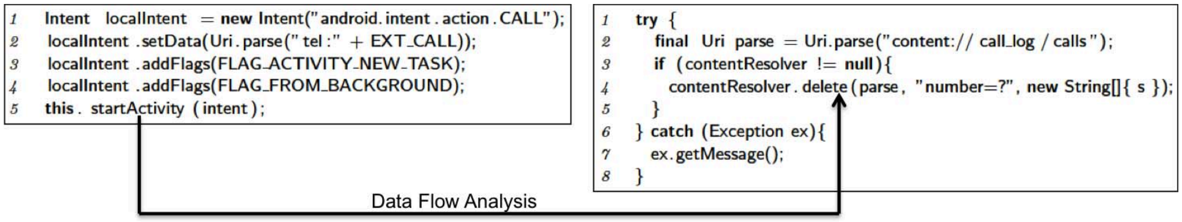
SH Behavior	Origin	SH Call (SAPI)
Hide app	BroadcastReceiver . onReceive(ACTION_BOOT_COMPLETED /SMS_RECEIVED/NEW_OUTGOING_CALL/SCREEN_OFF/ PACKAGE_INSTALL/PACKAGE_ADDED/SIG_STR)	Context. startService ()

Hide activity	any entry point	Window.addFlags(FLAG_NOT_TOUCH_MODAL)
		Window.setFlags(FLAG_NOT_TOUCH_MODAL,*)
		Window.requestFeature(FLAG_NOT_TOUCH_MODAL)

Delete message	BroadcastReceiver . onReceive(SMS_RECEIVED/ACTION_VIEW)	ContentResolver. Delete ("content :// sms")
Delete call log	BroadcastReceiver . onReceive(PHONE_STATE) PhoneStateListener . onCallStateChanged(TelephonyManager. CALL_STATE_RINGING)	ContentResolver. Delete ("content :// cal_log / calls ")
Block message	BroadcastReceiver . onReceive(SMS_RECEIVED)	BroadcastReceiver . abortBroadcast ("content :// sms")
Block call	BroadcastReceiver . onReceive(PHONE_STATE) PhoneStateListener . onCallStateChanged(TelephonyManager. CALL_STATE_RINGING)	ITelephony . endCall ()
Hide alert	any entry point	Context.sendBroadcast(Intent . ACTION_CLOSE_SYSTEM_DIALOGS)
Hide notification	BroadcastReceiver . onReceive()	NotificationManager. cancel () NotificationManager. cancelAll ()

Mute phone	any entry point	Vibrator . Cancel()
		AudioManager.setRingerMode(RINGER_MODE_SILENT)
		AudioManager.setStreamMute(**true**)
		AudioManager.adjustStreamVolume(ADJUST_LOWER)

Delete system log	any entry point	Runtime.Exec(" logcat −c")



```
Intent  localIntent  = new Intent("android. intent . action . CALL");
 localIntent .setData(Uri . parse(" tel :" + EXT_CALL));
 localIntent .addFlags(FLAG_ACTIVITY_NEW_TASK);
 localIntent .addFlags(FLAG_FROM_BACKGROUND);
this . startActivity ( intent );
```

```
try {
   final  Uri  parse  = Uri.parse("content:// call_log / calls ");
   if  (contentResolver  != null){
     contentResolver . delete (parse ,  "number=?", new String[]{ s });
   }
} catch (Exception ex){
   ex. getMessage();
}
```

Figure 4: Code snippets of malware Saveme that implement the “Delete call log” SHB.

Table 2: Pair action (PAPI) analysis.

SH Behavior	Pair start	Pair end

Delete message	SmsManager.sendTextMessage()/sendMultipartTextMessage()/ sendMultimediaMessage()
	SmsMessage.createFromPdu()
		ContentResolver. Delete ()

Delete call log	Context. startActivity () PhoneStateListener . onCallStateChanged() BroadcastReceiver . onReceive()	ContentResolver. Delete ()
Block message	SmsMessage.createFromPdu()	BroadcastReceiver . abortBroadcast ()
Block call	BroadcastReceiver . onReceive() PhoneStateListener . onCallStateChanged()	ITelephony . endCall ()

method invocation is the *result of an user decision.* We name this *UD analysis* for short.

The user’s GUI actions can be *decision-related* or *decision-unrelated*, as explained next. Decision-related actions include clicking a button, checking a checkbox or selecting a menu item; in other words, the user takes decisions (and acts accordingly) in a way meant to change the app state. Examples of decision-unrelated actions include scrolling down a window or changing focus. If an SAPI is invoked by a decision-related action, we rule that call as legitimate, rather than an SH attempt. However, if invoked by a decision-unrelated action, it can be an SHB. Note that existing research can only detect whether an API is invoked by a GUI [15, 17], whereas we further consider whether the GUI can reflect decisions.

In order to present the user-decision analysis approach, we introduce several definitions. User-Decision-GUI (*UDG*) is an interactive GUI element, e.g., Button, Checkbox, Radio Button, Toggle Button, Spinner, Picker, or menu. User-Decision-Callback (*UDC*) is a top-level callback method directly invoked as a result of the user action, e.g., onClick(), onCheckedChanged(). In contrast, some callback methods are due to decision-unrelated actions, e.g., onBackPressed(), onScroll(), onEditorAction().

Android offers two ways for creating a correspondence between a callback method and a GUI element: statically defining the callback as the handler of an event in the GUI element's layout file or dynamically defining a callback for the GUI element by registering a listener object – we handle both.

We determine that a given callback is an UDC if either of these two conditions is satisfied:

- The corresponding GUI of the callback is an UDG, *and* the event to be handled by the callback is a decision-related event, e.g., click. Note that there exist decision-unrelated events, e.g., scroll and focus change.
- The corresponding GUI of the callback is an UDG, *and* the listener of the callback is decision-related, e.g., onCheckedChange Listener. Note that there exist decision-unrelated listeners, e.g., onCreateContextMenuListener and onFocusChangeListener.

Finally, we infer that an API invocation is user-decided if all of its callbacks are UDC, which includes callbacks within the same component and callbacks in other components. If any callback is not UDC, we infer that the API call is not invoked by the user – further, if this call is an SAPI, it is potentially an SHB.

3.1.4 Activity Finish Analysis. This analysis detects activity hiding, i.e., whether an activity is terminated prematurely, before being displayed, as illustrated in Section 2.1.3. To achieve this, the activity calls finish() within onCreate(), onStart(), or onResume() (or their descendants in the call graph). Therefore, our analysis starts at the beginning of these three callback methods. We perform a control flow analysis to check whether there exists a path from the beginning of the callback to the callback's end that includes finish(); if such a path exists, it indicates potential activity hiding. We name this *AF analysis* for short.

3.1.5 Attribute Analysis. The purpose of this analysis is to check whether the app attempts to manipulate activity attributes in order to deceive the user. The analysis checks both the XML manifest file and the attribute-related API methods. For example, the liner layout of an activity has an attribute "background color". If the attribute value is #00000000, the activity is transparent. An app can set the value of an attribute in the manifest or layout files, or by calling certain API methods, e.g., setBackgroundDrawable(), setBackgroundColor() or setBackgroundResource(). Another example is the attribute excludeFromRecents which can be specified in the manifest file, or set via the API methods setFlags() and addFlags().

3.2 Detection Rules

We use *SAPI*, *PAPI*, *UD*, *AF*, and *Attribute* to denote the five static analyses. The detection rules for the SHBs are shown in Figure 5. If *any of the rules fires*, the tool will report the app as malicious. We now explain each rule.

Rule 1 reports "Hide icon" when the main activity is removed from the home screen without user involvement. Rule 2 detects "Hide app" if starting an app as a service without user involvement. Rule 3 reports "Hide activity" when the activity finish analysis returns true or the main activity is transparent. Rule 4 infers behavior "Delete message" when deleting occurs after receiving or sending a message. Rule 5 reports "Delete call log" when deleting occurs after making or receiving a phone call. Rule 6 detects "Block message" if blocking received or sent messages. Rule 7 reports "Block call" when blocking incoming or outgoing phone calls. Rule 8 reports "Hide alert" if the app closes a system dialog without user involvement. Rule 9 infers behavior "Hide notification" when canceling a notification without user intervention. Rule 10 detects "Mute phone" when muting the phone surreptitiously. Rule 11 finds SHB "Exclude from recent apps list" if the attribute EXCLUDE_FROM_RECENTS is set without user's involvement. Rule 12 reports "Delete system log" if that specific shell command is not launched by the user.

3.3 Implementation

We implemented our tool on top of the Soot and FlowDroid static analysis frameworks. These frameworks only analyze bytecode, so we added modules to analyze XML files (e.g., categories and attributes in `AndroidManifest.xml`, `style.xml`, etc). Our static analysis modules use both data-flow and control-flow analyses. Finally, the analysis results are produced using the detection rules.

3.4 Limitations

Limitations/false negatives. Our tool has several analysis limitations. First, if an SHB is invoked by GUI interaction but the GUI text does not reflect the invocation of the SHB, the tool will not report it; Huang et al.'s idea of finding mismatches between user interface and app behavior [17] could be used to address this limitation. Second, there were a few apps that, due to obfuscation, could not be analyzed, e.g., TripAdvisor (`com.tripadvisor.tripadvisor.apk`) and KCLS
(`com.bibliocommons.kcls.apk`).

Improving precision/reducing false positives. Our analysis, built on top of FlowDroid, is based on over-approximation, and handles reflection/native code conservatively – this can be a source of false positives. Also, the SAPI functions with zero parameters tend to have more false positives – a more precise alias and flow analysis would improve precision.

Broader concerns. There could be other classes of SHBs, beyond the ones we have discovered. Nevertheless, our list of SHBs: (1) is effective at malware discrimination, and (2) exposes questionable practices in benign apps.

1	Hide icon	!UD ∧ RemoveFromHomeScreen(MainActivity)
2	Hide app	!UD ∧ SAPI(Start_service)
3	Hide activity	ActivityFinish () ∨ (!UD ∧ AttributeAnalysis(Transparent_Main_Activity) ∧ SAPI(Set_Flags))
4	Delete message	!UD ∧ SAPI(Delete_Sms_Mms) ∧ (PAPI(Receive_Sms_Mms) ∨ PAPI(Send_Sms_Mms))
5	Delete call log	!UD ∧ (PAPI(Make_Phone_Call) ∨ PAPI(Receive_Phone_Call)) ∧ SAPI(Delete_Call_Log)
6	Block message	!UD ∧ SAPI(Block_Sms_Mms) ∧ PAPI(Receive_Sms_Mms) ∧ PAPI(Send_Sms_Mms)
7	Block call	!UD ∧ SAPI(Block_Phone_Call) ∧ PAPI(Receive_Phone_Call) ∧ PAPI(Make_Phone_Call)
8	Hide alert	!UD ∧ SAPI(Close_System_Dialog)
9	Hide notification	!UD ∧ SAPI(Cancel_Notification)
10	Mute phone	!UD ∧ (SAPI(Cancel_Vibrate) ∨ SAPI(Mute) ∨ SAPI(Adjust_Volume) ∨ SAPI(Chang_Ringer_Mode))
11	Exclude from recent apps list	!UD ∧ AttributeAnalysis(EXCLUDE_FROM_RECENTS)
12	Delete system log	!UD ∧ SAPI(Delete_Logcat)

Figure 5: Detection rules.

Finally, our approach cannot recognize specific *malware families*: certain SHBs might span multiple malware families. This is expected, as our design goal was at a lower level, automatic SHB identification, rather than clustering malware by family.

4 EVALUATION

In this section, we present an evaluation of our approach along several dimensions: Is the approach effective at identifying SHBs? Is the approach efficient? What are the main causes of false positives/false negatives? We begin by describing the two datasets used in our evaluation.

Datasets. Our first dataset, which we name **MA-198**, contains 198 malware samples that were decompiled and analyzed manually, in detail. The 198 samples come from the Malware Genome Project [31], Drebin [11], and AndroZoo [10].

The second dataset, which we name **ALL-9452**, consists of 6,233 benign apps[3] and 3,219 malicious apps.[4] These apps were analyzed automatically. To ensure that the benign set does not contain malware, we sent all the apps in this set to VirusTotal [7], a public malware scanning service. If an app is reported by at least one common anti-virus tool as malicious, we removed it from the benign set. For the malware samples, we performed a quick and simple static analysis to eliminate the samples without any possibility to have SHBs. This is done by searching requested permissions, major SAPI calls and intent actions. For example, if an app does not have permissions SEND_SMS and RECEIVE_SMS, it is impossible to have the SHB "Delete message". Moreover, in order to make sure that the samples are malware, we sent them to VirusTotal. If an app was reported malicious by less than two scanners, we removed it from the malware set.

Platform. The static analysis tool ran on an 8-core Intel Xeon i7-4770 (8MB Cache, 3.4 GHz) with 32GB of RAM. The system ran Ubuntu 14.04.1, Linux kernel version 3.13.0-32-generic.

Table 3: Effectiveness results on MA-198.

True SHBs	Over-reported SHBs (FP)	Under-reported SHBs (FN)
219	46	27
Precision: $\frac{219}{219+46} = 82.64\%$		*Recall*: $\frac{219}{219+27} = 89.02\%$
F−measure: $2 * \frac{82.64*89.02}{82.64+89.02} = 85.71\%$		

4.1 Effectiveness

The test for evaluating effectiveness consists of two steps: SHB detection validation (manual) and large scale measurement (automatic).

4.1.1 Manual Cross-checking on **MA-198**. As there is no existing oracle to determine SHB, we manually verified each static analysis report. Specifically, we reverse-engineered each app – decompiled the app (to source code) via the JADX decompiler [4]. Note that decompilation is not always possible due to obfuscation, so some of our manual analysis was based on source code inspection, some on Dalvik bytecode inspection. The results are shown in Table 3. Our tool has reported 265 SHBs in total; of these 219 were true SHBs, while 46 SHBs were over-reported (false positives) and 27 were false negatives, i.e., our tool missed those SHBs (the reasons will be discussed in Sections 4.1.3 and 4.1.4, respectively). This yields an F-measure of 85.71%, indicating that our tool is quite effective.

4.1.2 Automated Analysis on **AA-9452**. We now turn to discussing the large-set results, shown in Table 4. Note that here the numbers in columns 2–4 indicate *apps*, not SHBs (since we only have ground truth for app nature, not SHBs).

A sample is identified as malicious if it exhibited any one of the SH behaviors. The tool missed (false negatives, or 'FN') 311 samples from the malware set, hence the recall value is 90.62%. The tool also reported 996 benign apps as having SHB (false positives, or 'FP') from the benign set, hence the precision is 84.02%. While these 996 apps were not malicious, their use of SHB is questionable – we discuss such uses at length in Section 5.

[3] The benign app samples are from Google Play and AndroZoo [10]; specifically, 4,970 (70%) of the benign apps are from Google Play and span all 33 app categories, as well as games.

[4] The malware samples are from Drebin [11], DroidCat [3], Kharon [5], AndroMalShare [2], Malware Genome Project [31], and Offensive Computing [6].

Table 4: Effectiveness results on AA-9452.

	Apps	Apps reported as Malicious	Apps reported as Benign	SHBs	SHBs /app
Malicious set	3,219	2,908	311 (FN)	4,843	1.5
Benign set	6,233	996 (FP)	5,237	1,241	0.2

$Precision$: $\frac{5,237}{5,237+996} = 84.02\%$ $Recall$: $\frac{2,908}{2,908+311} = 90.62\%$

$F\!-\!measure$: $2 * \frac{84.02*90.62}{84.02+90.62} = 87.19\%$

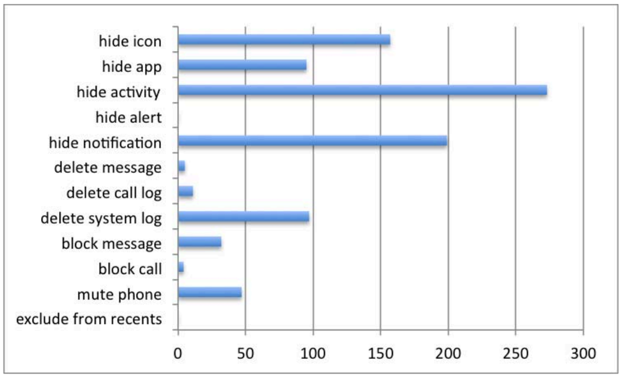


Figure 6: FPs generated by each SHB.

Finally, the F-measure is 87.19%; the malware set exhibited 1.5 SHB per sample on average[5] while the benign set exhibited only 0.2 SHB per sample. *We believe that the high F-measure value and the per-app figures of 1.5 SHB (malicious) vs 0.2 (benign) indicate that our approach is* ***effective*** *for detecting SHB (and perform SHB-based triaging) in Android apps.*

4.1.3 False Positives. To better understand the causes of false positives, in Figure 6 we have grouped them by SHBs. Five SHBs, "hide activity", "hide notification", "hide icon", "hide app" and "delete system log", generated the most false positives. We investigated this and found that the false positives were due to several reasons:

(1) Certain apps employ SHB, such as running in the background without the user having started the app, or without the user being able to see that running app, in the name of improving user experience (see Section 5).
(2) Static Analysis: alias, data-flow, and control-flow analyses are over-approximating, which is inherent in static analysis.

4.1.4 False Negatives. We have categorized the false negative sources as follows:

(1) Parameters of an SHB are dynamically sent from a remote control server, hence our static analysis cannot identify the behavior. For example, spyware Saveme has a remote server that sends the id, time or phone number through the network to delete certain SMS/MMS messages.
(2) SHBs are launched by GUI interaction, but the behavior mismatches the content shown on the GUI. For example, apps Pure girl and iCalendar employ this behavior.
(3) Some malware samples do not have SHB, though they do invoke SAPI calls, e.g., Towelroot and FakeCMCC. Our tool did not identify these samples as malicious.

4.1.5 Behavior statistics. Figure 1 shows the number of each type of SHBs detected per 1,000 malware samples and 1,000 benign samples, respectively.[6] The total number of SHBs detected in the malware set is 1502 per 1000 samples while in the benign set it is only 185 per 1000 samples. "Hide app", "Hide activity" and "Block message" are the three most common SHBs in the malware set.

4.2 Efficiency

Running our tool on the 9,452 apps took about 10 days. We show the detailed efficiency results in Table 5. The "Bytecode size" grouped columns show that the datasets had substantial variety in terms of app size, and some apps' bytecode size was as large as 24 MB. The "Time" grouped columns show running time statistics for each dataset. We focus on **AA-9452** as it is larger, hence more representative. The mean analysis time was 84 seconds while the median was 37 seconds, which shows that our analysis is practical. Finally, we believe that even the maximum analysis time of 15,290 seconds (i.e., 4 hours 15 minutes) is acceptable for a static analysis. To conclude, *with a median analysis time of 37 seconds on a median app size of 2.4MB we believe that our approach is* ***efficient*** *at SHB analysis.*

5 SELF-HIDING BEHAVIOR IN BENIGN APPS

For each SHB category our tool has found in benign apps, we performed a two-part targeted manual investigation: first, we analyzed the disassembled bytecode, and then ran the app with instrumentation to confirm the SHB. We focused this investigation on two categories of apps: (1) apps that are very popular, e.g., with more than 100 million installs; or (2) less popular apps which displayed severe cases of SHB. Ultimately we aimed to answer the questions " Why does this SH behavior occur and what are the consequences for the user?" This section summarizes some of our findings; we limit the discussion to 8 SHBs for brevity.

5.1 Hide App

Many popular benign apps, such as Airbnb and BBM start themselves as an automatic service after receiving the BOOT_COMPLETED event. This event, which requires the permission RECEIVE_BOOT_COMPLETED , notifies the app that the system has rebooted. In conjunction with this event and permission, there is a function which launches the auto-start service. Our tool reports this as "Hide app" SHB. Apps employ this technique as a means to initialize app-specific information and functions upon startup. While it could be argued that the app is not hiding in the malicious sense (rather it is running in the background to have access to certain types of data – most commonly, location services), we believe that users should know when such

[5] Median = 1, min = 0, max = 5.

[6] The 1,000 benign apps are a random sample extracted from the 6,233 benign apps, while the 1,000 malicious apps are a random sample extracted from the 3,219 malicious apps.

Table 5: Efficiency results.

Dataset	Bytecode size (KB)				Time (seconds)			
	min	max	average	median	min	max	average	median
MA-198	32	15,021	5,326	1,995	13	6,596	140	32
AA-9452	5	24,218	4,496	2,459	2	15,290	84	37

apps are running: (1) so they understand why the battery is draining, and (2) so they understand the privacy implications of apps accessing and transmitting sensitive information (e.g., location) in the background.

5.2 Hide Notification

Certain apps, such as Waze, Truecaller, All in One Toolbox, Quick Heal Mobile Security, and MiniFetion use NotificationManager.cancel() or NotificationManager. cancelAll () to block notifications without user intervention. As a result these apps have been marked as having the "Hide notification" SHB. This is due to the nature of cancel() and cancelall (), which cancel all previously-shown notifications. Apps employ this technique as a means to update the user to the most recent notification or to consolidate notifications, especially in communication apps such as MiniFetion and TrueCaller. Lately, many "clean up" and "device maintenance" apps have started to exhibit this behavior for the same reasons. Consolidated notifications may appear convenient to the user, however the app does not have a means to show high-priority notifications first (other than through chronological order). Therefore, users might prefer to receive notifications for all messages to reduce the risk of missing an important notification. However in the case of Waze, the app blocks certain notifications using Vanagon Notification Manager which cancels all app notifications when the user is not driving. While the app might be trying to appear helpful, notification cancellation and blocking without user's consent/awareness is questionable at best.

5.3 Mute Phone

Our tool discovered the use of AudioManager.setRingerMode() in the benign app Camera360. As its name states, this is a camera app which edits and takes photos; it has more than 100 million installs and was "Best App of 2016 on Google Play in several countries".[7] Many camera apps use volume controls when recording audio. Our tool also discovered the "Mute phone" SHB in certain benign popular apps like Smart Truck Route and All in One Toolbox due to the use of Vibrator.cancel() and AudioManager.setRingerMode(). Regarding Smart Truck Route, the app directly checks and manipulates the device's audio settings, including its ringer mode. As for All in One Toolbox, the app mutes the phone based on the SDK version of the device. This is dubious behavior for a utility app aimed at optimizing the Android device. To sum up, even though it seems reasonable in some of these cases, we believe that muting the phone should be done *by the user through a system-wide control* rather than *silently by the app.*

5.4 Block Message

As the BroadcastReceiver is usually a dormant app component, it is not surprising that its methods can be categorized as SHBs, especially abortBroadcast(). As a result, many benign apps can exhibit this behavior. Interestingly, these apps are not limited to those which rely heavily on BroadcastReceiver. For example, the popular navigation app Waze uses abortBroadcast() which can be construed as the "Block Message" SHB. The abortBroadcast() method is used to prevent other receivers from obtaining the broadcast, thus blocking the communication. It might be justified that Waze employs this tactic as a means to prevent itself from getting location-based alerts that may be irrelevant or annoying to the user. While the intentions of message-blocking apps might appear benign, such blocking removes decision-making from the user and can interfere with usability.

5.5 Block Call

Apps which use ITelephony.endCall() are considered to have the 'Block Call" SHB. The benign app Truecaller has the sole purpose to identify and block spam calls, hence it was obviously marked to have this behavior. Despite explicitly stating that it automatically blocks calls, an app which decides for the user which calls are spam can be maliciously manipulated against the user's interest.

5.6 Hide Icon

"Hide icon" achieves its goal by deleting an activity's category.LAUNCHER from the Android manifest. While this deletion merely indicates that that activity should appear as an initial activity of a task, it is evident that a deceitful app can use hide-icon to promote other activities, masking the deceitful app beneath. Many popular benign apps such as ES File Explorer and Next Launcher 3D Shell Lite have this behavior. For example, app Next Launcher 3D Shell Lite is a premium launcher for Android's home screen, but one of its key features is that it draws 3D icons and widgets over their original counterparts. App ES File Explorer has permissions to draw over apps, which is surprising and might be regarded as excessive for a file manager. By having the ability to promote certain activities and controlling the launcher's top level apps, apps with this SHB should be treated with caution.

5.7 Delete Call Log

The app Quick Heal Mobile Security exemplifies this SHB. The app uses ContentResolver.Delete() to delete the call logs on the device. The app has call filtering capabilities and has explicit permissions to read and write call logs on the user's device. Nevertheless, (1) users may not be aware of the security implications of log deletion, and

[7] https://play.google.com/store/apps/details?id=vStudio.Android.Camera360&hl=en

(2) the user does not initiate call deletion. These two factors make this particular SHB instance quite problematic.

5.8 Delete System Log

MiniFetion, an app from the Baidu app marketplace, sends free SMS to the user's contacts. Despite the seemingly straightforward nature of the app, we found two highly questionable behaviors. First, the app deletes the system log via `"logcat -c"`. Second, the app has an activity MobClickAgent which uploads device logs to a third party server. Thus the app is able to manipulate, as well as exfiltrate, the system logs without the user's awareness. While not many popular apps have this SHB, users need to be extremely suspicious of any app which send device logs and user information to third-party servers.

6 RELATED WORK

Behavior-based malware detection for Android has been long been studied due to the prevalence of malware in the Android ecosystem; a variety of methods for characterization and behavior-based detection have been proposed.

Malware behavior characterization. CopperDroid characterizes malware behavior based on how it is initiated, either through Java, JNI, or native code execution [20]. SmartDroid uses a combination of static and dynamic analysis to detect conditions as a way to expose the behavior of Android malware in UI-based triggers [29].

Machine learning. Crowdroid uses crowdsourcing to obtain traces of an app's behavior [14]; it distinguishes between benign and malicious apps of the same name and version by detecting anomalous behavior using k-means; some limitations include having to rely on the Android user community as a source for app traces, as well as having high energy consumption on devices. PUMA [21] evaluates the scope and use of permissions. DREBIN [11] uses both static analysis and machine learning to optimize analysis and detection patterns; MAMA [22] uses classifiers based on features to detect malware. Yerima et al. [28] use static analysis to build Bayesian models as a way to detect evasive malware. Andromaly applies machine learning (anomaly detectors) to classify features and events as benign or malicious [25]. DroidRanger uses a permission-based footprinting scheme to detect malware followed by a heuristic-based filtering scheme to identify behavior of unknown malware families [32]. Droid Detective detects malware based on permission combination [19]; by obtaining permission combinations which have been requested frequently by malware, it auto-generates sets to be used as a means of identifying malware. MONET combines runtime behavior with static structures to detect malware variants and to generate a runtime signature of the malware [27]. AVclass performs massive-scale malware labeling through clustering anti-virus labels, and identifying the most likely family names for each sample [24]. Similarly, Euphony categorizes malware samples based on clustering the anti-virus labels produced by anti-virus vendors [18].

Static analysis. Apposcopy uses static analysis to extract malware properties, and takes a more semantics-oriented approach to classify malware based on its signature [16]. However one of its biggest limitations is that it cannot detect obfuscation or self-hiding. Another signature-based tool is DroidAnalytics which collects, manages, and extracts malware and analyzes mutations and repackaging methods [30]. Similarly, DroidAPIMiner uses classifiers based on semantic information from the bytecode of apps, namely API calls [9]. Using dataflow analysis and frequency analysis, it captures the most common and relevant API calls used by malware. Feng et al. focus on a structure of information flows gathered through the sequence of API calls and the patterns of behavior present to identify malware [26]. Similarly, we have employed such API and dataflow analyses in our work to analyze SHB. MADAM is a host-based malware detection system for Android devices [23]; it analyzes features at 4 levels – kernel, app, user, and package – which enables it to detect 125 existing malware families.

Rather than focusing on general behavior-based analysis of malware, we implement a range of static analyses to detect SHB and malware. Machine learning, while useful in better understanding malware behavior, has several disadvantages, e.g., the models it learns are opaque whereas we have a static analysis report that helps users/developers/marketplaces trace the SHB precisely; furthermore, machine learning may oversimplify SHB resulting into large numbers of false positives. Most static analyses of malware are focused on behavior and do not employ attribute analysis as we do; this analysis is key in identifying SHB such as transparent activities. Finally, our work differs from dynamic analysis-based approaches in the standard static vs. dynamic analysis way: due to static analysis our approach is prone to false positives, but does not require running the app. Dynamic analysis is prone to false negatives and requires high-quality inputs to ensure good coverage.

7 CONCLUSION

Motivated by the common tendency of Android malware to self-hide in order to deceive users and cover malicious traces, we define a set of self-hiding behaviors and construct a suite of static analyses to reveal such behavior. Our experiments indicate that the presence of self-hiding behavior is strongly associated with malice in a given app. Nevertheless, we also found plenty of benign, widely-popular apps that employ hiding techniques, which suggests that end-users and marketplaces would benefit from using an approach like ours to shed light on potential nefarious behavior in Android apps and improve user experience.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1617584. Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] 2017. 8,400 new Android malware samples every day. (April 2017). https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day.

[2] 2017. AndroMalShare. (August 2017). http://sanddroid.xjtu.edu.cn:8080.

[3] 2017. The DroidCat Dataset. (June 2017). http://www.people.vcu.edu/~rashidib/Res_files/DroidCatDataset.htm.

[4] 2017. JADX. (August 2017). http://skylot.github.io/jadx/.

[5] 2017. Kharon project. (May 2017). http://kharon.gforge.inria.fr/index.html.

[6] 2017. Open Malware. (August 2017). http://www.offensivecomputing.net/search.cgi?search=android.

[7] 2017. VirusTotal. (August 2017). https://www.virustotal.com.

[8] 2018. Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2017. (February 2018). https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/.

[9] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. Droidapiminer: Mining api-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems*. Springer, 86–103.

[10] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[11] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.. In *NDSS*.

[12] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. 2017. The Soot-based Toolchain for Analyzing Android Apps. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft '17)*. IEEE Press, Piscataway, NJ, USA, 13–24. https://doi.org/10.1109/MOBILESoft.2017.2

[13] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[14] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 15–26.

[15] Karim O. Elish, Danfeng (daphne Yao, and Barbara G. Ryder. 2012. User-Centric Dependence Analysis For Identifying Malicious Mobile Apps *(MOST'12)*.

[16] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. 2014. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 576–587.

[17] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. 2014. AsDroid: Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1036–1046. https://doi.org/10.1145/2568225.2568301

[18] Mederic Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawende F. Bissyande, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. 2017. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for Android malware. In *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE, 425–435.

[19] Shuang Liang and Xiaojiang Du. 2014. Permission-combination-based scheme for android mobile malware detection. In *Communications (ICC), 2014 IEEE International Conference on*. IEEE, 2301–2306.

[20] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. 2013. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April* (2013).

[21] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. 2013. Puma: Permission usage to detect malware in android. In *International Joint Conference CISIS´12-ICEUTE´ 12-SOCO´ 12 Special Sessions*. Springer, 289–298.

[22] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Javier Nieves, Pablo G Bringas, and Gonzalo Álvarez Marañón. 2013. MAMA: manifest analysis for malware detection in android. *Cybernetics and Systems* 44, 6-7 (2013), 469–488.

[23] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. 2016. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing* (2016).

[24] Marcos Sebastian, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 230–253.

[25] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. 2012. Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems* 38, 1 (2012), 161–190.

[26] Feng Shen, Justin Del Vecchio, Aziz Mohaisen, Steven Y Ko, and Lukasz Ziarek. 2017. Poster: Android Malware Detection using Multi-Flows and API Patterns. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 171–171.

[27] Mingshen Sun, Xiaolei Li, John CS Lui, Richard TB Ma, and Zhenkai Liang. 2017. Monet: a user-oriented behavior-based malware variants detection system for android. *IEEE Transactions on Information Forensics and Security* 12, 5 (2017), 1103–1112.

[28] Suleiman Y Yerima, Sakir Sezer, Gavin McWilliams, and Igor Muttik. 2013. A new android malware detection approach using bayesian classification. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*. IEEE, 121–128.

[29] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 93–104.

[30] Min Zheng, Mingshen Sun, and John CS Lui. 2013. Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*. IEEE, 163–171.

[31] Yajin Zhou and Xuxian Jiang. 2012. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 95–109. https://doi.org/10.1109/SP.2012.16

[32] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets.. In *NDSS*, Vol. 25. 50–52.