# Demystifying Hidden Sensitive Operations in Android Apps

XIAOYU SUN, XIAO CHEN, and LI LI, Monash University
HAIPENG CAI, Washington State University
JOHN GRUNDY, Monash University
JORDAN SAMHI, TEGAWENDÉ BISSYANDÉ, and JACQUES KLEIN, University of Luxembourg

Security of Android devices is now paramount, given their wide adoption among consumers. As researchers develop tools for statically or dynamically detecting suspicious apps, malware writers regularly update their attack mechanisms to hide malicious behavior implementation. This poses two problems to current research techniques: static analysis approaches, given their over-approximations, can report an overwhelming number of false alarms, while dynamic approaches will miss those behaviors that are hidden through evasion techniques. We propose in this work a static approach specifically targeted at highlighting hidden sensitive operations (HSOs), mainly sensitive data flows. The prototype version of HiSenDroid has been evaluated on a large-scale dataset of thousands of malware and goodware samples on which it successfully revealed anti-analysis code snippets aiming at evading detection by dynamic analysis. We further experimentally show that, with FlowDroid, some of the hidden sensitive behaviors would eventually lead to private data leaks. Those leaks would have been hard to spot either manually among the large number of false positives reported by the state-of-the-art static analyzers, or by dynamic tools. Overall, by putting the light on hidden sensitive operations, HiSenDroid helps security analysts in validating potentially sensitive data operations, which would be previously unnoticed.

CCS Concepts: • **Security and privacy** → Domain-specific security and privacy architectures;

Additional Key Words and Phrases: Android application, privacy leak, hidden sensitive operations, program analysis

**ACM Reference format:**
Xiaoyu Sun, Xiao Chen, Li Li, Haipeng Cai, John Grundy, Jordan Samhi, Tegawendé Bissyandé, and Jacques Klein. 2023. Demystifying Hidden Sensitive Operations in Android Apps. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 50 (March 2023), 30 pages.
https://doi.org/10.1145/3574158

This work was partly supported by the **Australian Research Council** (**ARC**) under a Laureate Fellowship project FL190100035, a **Discovery Early Career Researcher Award** (**DECRA**) project DE200100016, and a Discovery project DP200100020, by the **Luxembourg National Research Fund** (**FNR**) (under project CHARACTERIZE C17/IS/11693861), by the SPARTA project, which has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 830892.

Authors' addresses: X. Sun, X. Chen, L. Li (corresponding author), and J. Grundy, Monash University, Wellington Rd, Clayton VIC 3800, Australia; emails: {xiaoyu.sun, Xiao.chen}@monash.edu, lilicoding@ieee.org, john.grundy@monash.edu; H. Cai, Washington State University, Wilson Rd, Pullman, WA 99164-5910; email: haipeng.cai@wsu.edu; J. Samhi, T. Bissyandé, and J. Klein, University of Luxembourg, 2 Avenue de l'Universite, 4365 Esch-sur-Alzette, Luxembourg; emails: {jordan.samhi, tegawende.bissyande, jacques.klein}@uni.lu.

1049-331X/2023/03-ART50 $15.00
https://doi.org/10.1145/3574158

## 1 INTRODUCTION

Android is the most adopted mobile operating system in terms of users, applications, and developers [9]. However, its popularity means that legitimate developers must co-exist with malware writers. Reports on many different kinds of attacks are presented in the technology and lay media. For example, security researchers have reported a malicious "clicker trojan"[1] which has been bundled into 34 different Google Play apps that have already been installed more than 100 million times.[2] On a larger scale, antivirus engines have been flagging a large number of apps as potential threats. For example, as of October 2020, the popular AndroZoo dataset [8] has recorded more than 226,000 Android GooglePlay apps than have been flagged as adware/malware by at least 5 Antivirus products, and this number is still growing. Those adware/malware often do not work alone but collaborate with many third parties over the internet. Some of the representative malicious behaviors include leading users to malicious websites through devious advertisements [22, 23, 46, 71], distributing malicious apps in the mobile network through drive-by downloads [19], leaking users' sensitive data to web servers through HTTP connections [28, 37, 47, 65], and so on.

To protect Android users against the rapid spread of malware, the research and practice communities have implemented a variety of measures and proposed several approaches to detect malware [11, 43, 52, 66, 72, 74]. These include static code analysis-based approaches [39, 41], dynamic testing-based approaches [34], and learning-based approaches [49]. Unfortunately, the emergence of many different malware detection techniques has also stimulated malware attackers into being more innovative to increasingly better hide malicious behavior, in order to bypass static code analysis (e.g., via obfuscation) and even dynamic detection (e.g., sensing of sandbox execution). In practice, sophisticated code obfuscation techniques [53] are being leveraged by attackers to hide their malicious program behavior, leading to false negatives in most static analyses thus resulting in imprecise and unsound results. Camouflage techniques have been frequently leveraged by attackers to evade dynamic testing approaches [25, 62]. Attackers often introduce a so-called logic bomb or time bomb to set off malicious functions only after certain conditions are met. For instance, after knowing that Google applies a dynamic analysis tool called *bouncer* to scan every app submitted to Google Play for five minutes, as revealed by Oberheide et al. [55], a bunch of malicious apps have been created and been demonstrated to be capable of penetrating Google's bouncer vetting system by simply waiting five minutes before triggering their malicious behavior.

To cope with such hidden malicious behaviors, researchers have devised new detection approaches. For example, Fratantonio et al. [27] have proposed an approach called TriggerScope to detect hidden behaviors triggered by predefined circumstances such as events related to location, time, and SMS. However, TriggerScope is not capable of detecting such malicious activities hidden behind other trigger types, such as the existence of other services (i.e., other than location, time, and SMS). In line with this research, Pan et al. [57] have proposed a machine learning-based approach aiming to discover unknown trigger types. Their approach, however, needs to manually label a dataset for training, which is known to be resource-intensive and error-prone.

Static analyzers suffer less than dynamic approaches from evasion techniques such as logic bomb or time bomb. In particular, regarding sensitive flow detection (also called privacy leak detection), numerous static analysis tools have been proposed such as FlowDroid [12] (and its extension IccTA [38]), Amandroid [70], or DroidSafe [30]. Although these tools are able to

[1] Such as the *Android.Click.312.origin* trojan and its modified variant Android.Click.313.origin trojan. This aims to generate fraudulent click-through and subscription revenues.

[2] https://www.forbes.com/sites/zakdoffman/2019/08/13/android-warning-100m-users-have-installed-dangerous-new-malware-from-google-play/#1956f51c22a9.

track sensitive flows (which are often hidden) by bringing key new contributions to the research community, they still face some well-known limitations [60]: their inherent over-approximations inevitably lead to false alarms, which, for some analyzers, occur at a high rate, making them impractical. Consequently, when building on static analysis, a manual investigation is often required. Unfortunately, such efforts cannot scale. Dynamic validation then appears as an alternative. Unfortunately, runtime execution often misses hidden sensitive flows due to the implementation of evasion techniques by attackers. While some effort (e.g., [27, 57]) has been put to *characterize* **Hidden Sensitive Operations** (**HSOs**) in Android apps, our community has not yet proposed dedicated approaches to *detect and explain* such operations, allowing attackers to achieve malicious behaviors while bypassing certain security vetting mechanisms.

We fill this research gap in this work by proposing a new prototype tool, HiSenDroid, which deploys an automated static app analyzer tailored for detecting HSOs. HiSenDroid performs a sequence of static analyses, including call graph analysis, forward data-flow analysis, interprocedural backward data-flow analysis, and so on. For exposed HSOs, HiSenDroid further goes one step deeper to record detailed information for explaining why these HSOs should be flagged as such.

To summarize, key contributions of our work include:

- We propose using a static analysis approach to discover HSOs that are not exposed to the state-of-the-art static and dynamic analysis tools in Android apps. To this end, we leverage control flow and data flow analyses to identify the unique code-level characteristics of HSOs.
- We designed and implemented a prototype tool HiSenDroid for analyzing HSOs. We release HiSenDroid as an open-source project [5] for supporting security analysts in their analysis needs and fostering further research in this direction.
- We evaluated HiSenDroid on a large-scale dataset that contains 10,000 benign and 10,000 malware samples and discovered emerging anti-analysis techniques employed by malware samples, such as fulfilling certain restrictions related to *time*, *location*, *SMS message*, *system properties*, *package manager*, and other logics.
- With the help of FlowDroid [12], a static taint analyzer, we further experimentally show that HSOs have been recurrently leveraged by attackers to leak sensitive user information.

The rest of the article is organized as follows: Section 2 defines HSO and presents the motivation of our research, i.e., why there is a strong need to demystify HSO. Section 3 depicts the design and implementation of the proposed approach. Sections 4 and 5, respectively, describe the characteristics of common and susipicious HSOs detected by our approach from a large-scale dataset. Section 6 presents a practical implication of our approach by characterizing sensitive data leaks triggered by HSOs. Section 7 discusses the limitations of the tool. Section 8 reviews the related works, and finally Section 9 concludes this article.

## 2 HSO DEFINITION AND MOTIVATION

We conducted an exploratory study to understand the characteristics of HSOs in Android apps. We first dumped operations in a set of real-world Android malware. Then, we manually examined those operations to observe the characteristics of such operations that could be considered as hidden-triggered operations. Based on our manual summarization, we found that (1) *if statement* and the notion of *branch* are key in the definition of HSO; (2) the *if statement* contains a specific *operation* that triggers the hidden sensitive flows, and this trigger condition is related to Android API.

Let $B$ denote one of the two branches of an *if-then-else statement*, or the branch of an *if statement* where the *else* branch is considered empty.

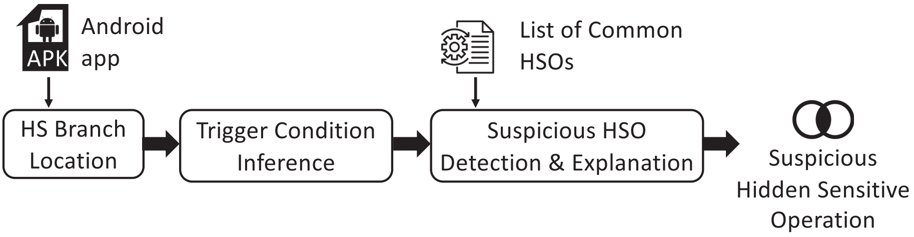

APK
Android app
HS Branch Location
Trigger Condition Inference
List of Common HSOs
Suspicious HSO Detection & Explanation
Suspicious Hidden Sensitive Operation

Fig. 1. The working process of HiSenDroid.

*Definition 1 (**Hidden Sensitive Branch (HSB)**).* *B* is an HSB if it fulfills the following **rules**:

(1) *B* contains sensitive Android APIs, and these APIs are different from those contained in the other branch involved in the *if-then-else statement*. The rationale behind this condition is that a hidden branch is supposed to achieve some sensitive behaviors that are different from those of the "normal" branch (i.e., non-HSB), which per se might also access sensitive APIs as part of the app's expected behaviors.

(2) *B* does not involve any of the variables appearing in the *condition expression* of the *if-then-else statement*. The rationale behind this is that the branch is triggered by conditions that are also different from its (sensitive) behaviors.

Less formally, an HSB could be defined as an "if branch" which accesses sensitive APIs, and which is fully "independent" of the *if condition* and the other branch of the *if statement*.

Let *C* denote the *condition* of an *if statement*.

*Definition 2 (HSO).* An HSO is an HSB that is triggered by a condition *C* containing values obtained via (or directly impacted by) Android system APIs or system properties (i.e., attributes of system classes). This may return different values when being executed under different circumstances, so as to trigger HSOs.

Listing 1 exemplifies a simplified code snippet illustrating these definitions in practice. Note that Listing 1 presents the typical characteristics of an HSO in many real-world apps that we have manually analyzed. At line 7, the app firstly checks if it is running on one of the popular Android emulators (i.e., *genymotion, bluestacks, and bignox*). If not, the app reads the device information and sends it to a hard-coded phone number through an SMS. Otherwise, if an emulator environment is detected, it will only perform some unharmful string operations (ignored). In this example, three private data—namely the device's IMEI, IMSI, and phone number—are retrieved in lines 9–11 and sent to a hard-coded phone number via SMS (line 15). All of these three leaks are hidden behind the trigger condition *ed.checkPackageName()* (line 7). The trigger condition checks the return value of a self-defined method *checkPackageName()* (line 30), which is determined by several other *if-conditions* defined in the invoking method (lines 31,37, and 39). Finally, the trigger condition in the HSO is traced back to a system API *PackageManager.queryIntentActivities()* (line 38) (cf. Definition 2). This trigger condition examines whether popular Android emulator packages (lines 26–28) are available in the device, i.e., checking if the app is running on these emulators. If the running environment is not one of the hard-coded emulators, the HSO will be performed. Otherwise, benign string operations are executed (lines 17–19) (cf. Definition 1).

## 3 OUR APPROACH

To better help security analysts understand HSOs placed in Android apps, we designed and implemented a prototype tool, named HiSenDroid, to automatically locate such operations in Android apps. HiSenDroid takes as input an Android app and outputs a set of HSOs. Figure 1 illustrates

```
public class MainActivity extends AppCompatActivity {
 protected void onCreate(Bundle savedInstanceState) {
  SmsManager smsManager = SmsManager.getDefault();
  ED ed = new ED(this);
  StringBuilder message = new StringBuilder();

  if(ed.checkPackageName()) {
   TelephonyManager tm = (TelephonyManager)     getSystemService(Context.TELEPHONY_SERVICE);
   String imei = tm.getDeviceId();
   String phoneNumber = tm.getLine1Number();
   String subscriberId = tm.getSubscriberId();
   message.append(imei);
   message.append(phoneNumber);
   message.append(subscriberId);
   smsManager.sendDataMessage("+115800763861", null, (short)1001, message.toString().getBytes(), null, null);
  } else {
   //benign string operations
}}


public class ED {
 public ED(Context pContext) {
  mContext = pContext;
  mListPackageName.add("com.google.android...genymotion");
  mListPackageName.add("com.bluestacks");
  mListPackageName.add("com.bignox.app");
 }
 public boolean checkPackageName() {
  if (!isCheckPackage || mListPackageName.isEmpty()) {
   return false;
  }
  final PackageManager packageManager = mContext.getPackageManager();
  for (final String pkgName : mListPackageName) {
   final Intent tryIntent = packageManager.getLaunchIntentForPackage(pkgName);
   if (tryIntent != null) {
    final List<ResolveInfo> resolveInfos = packageManager.queryIntentActivities(tryIntent,
      PackageManager.MATCH_DEFAULT_ONLY);
    if (!resolveInfos.isEmpty()) {
     return true;
    }
   }
  }
  return false;
}
```

Listing 1. An example of a real-world hidden sensitive data flow.

the working process of HiSenDroid. It achieves the aforementioned goal through three main modules, namely: (1) HSB Location; (2) Trigger Condition Inference; (3) Suspicious HSO Detection and Explanation. We now respectively detail these three modules.

### 3.1 Hidden Sensitive Branch Location

The first module of HiSenDroid is responsible for locating HSBs in Android apps (i.e., fulfilling the rules in Definition 1). Towards locating HSBs, this module first statically goes through all the methods that appeared in the DEX file of the input APK. For each method, this module then constructs an intra-procedural **control-flow graph** (**CFG**) and traverses the graph to locate *if-then-else statements*. Once an *if-then-else statement* is located, it further extracts the sensitive APIs accessed by the two branches (hereinafter referred to as *if-branch* and *else-branch*). Sensitive APIs are such methods that are protected by Android permissions, which are classified following the latest Android API-permission mappings PSCout [13], Axplorer [14], Arcade [7], and NatiDroid [36]. Any of the two branches will be considered a potential HSO if it has indeed accessed sensitive APIs that are different from the APIs accessed by the other branch.

When extracting sensitive APIs, in order to obtain a *soundy* result [51] (e.g., including all the sensitive APIs accessed by a potential HSB), this module traverses not only the methods directly presented in the HSB but also all the methods that could be reached from the branch. This process is made possible by first constructing a **call graph** (**CG**) for the input APK. Unfortunately, as discussed by many existing works, Android apps do not have a single entry point (e.g., *main()*)

that connects other parts of the application code, making static analyses challenging to cover all the app code. Fortunately, this challenge has been well addressed by the state-of-the-art by artificially creating a so-called dummy main method, connecting together all the separated code parts, including system-driven lifecycle methods and event-driven callback methods [12].

Based on our observation and the findings of previous work [57], the connection between trigger conditions and the operations along its paths is often weak. Indeed, the variables appearing in triggers typically do not propagate data flow to its following paths. Take Listing 1 as an example, the app checks if it is running inside Android emulators at line 7, where the trigger condition code itself is not supposed to steal private data and is only meant to determine the right situation for running hidden code. To leverage this property, we attempt to check whether variables appearing in the HSB have data dependency with any variable within the condition expression. Thus, for a given potential HSB, this module goes one step further to check if any of the variables appeared in the HSB's *condition expression* has been leveraged by the HSB code. If so, this HSB will not be considered as a true HSB and thereby will be excluded from further analyses. This module achieves this by conducting a simple intra-procedural control-flow analysis. In the case of true HSB, there should not be intersections between the set of variables that appeared in its *condition expression* and those within the branch.

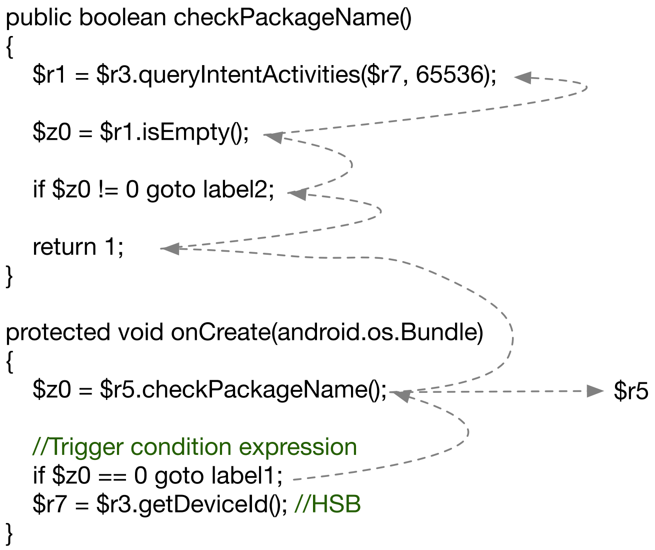
### 3.2 Trigger Condition Inference

After locating HSBs, the second module goes one step deeper to infer HSOs so as to fulfill Definition 2. Given a true HSB, the idea of detecting HSOs is to infer the detailed trigger conditions that lead to the execution of the HSB.

We began with a preliminary study to understand what kinds of trigger conditions have been used to hide suspicious APIs, as identified from the literature [1, 2, 4, 17, 18, 21, 31, 54, 57, 59, 68] on trigger conditions. For example, Petsas et al. [59] investigated anti-analysis techniques that can be employed by Android apps to evade detection, including pre-initialized static information(e.g., IMEI value), dynamic information that does not change (e.g., Sensors data) and VM instruction emulation (e.g., hardware variable). In their article, they demonstrated how dynamic analysis could be evaded by the aforementioned trigger conditions in an emulated environment. Pan et al. [57] further summarize that almost all the trigger conditions of HSOs can be characterized by **System Properties** (e.g., OS or hardware traces of a mobile device) or **Environment Parameters** (time, locations, SMS, etc.). To the best of our knowledge, the values in both types can be obtained through Android system APIs. In other words, an HSO trigger condition is expected to involve, directly or indirectly, one or more system API calls for interacting with the Android operation system. Therefore, since it is very important to identify all possible trigger conditions, we propose considering all the condition checks to infer HSO's trigger conditions[3] as long as they involve system properties, environment parameters, and any other values yielded by system APIs.

In this work, we follow the same criteria to infer HSOs (i.e., the trigger conditions involve values obtained through Android system APIs). Specifically, to infer the trigger conditions, for each of the variables that appeared in the HSB's *condition expression*, there is a need to conduct backward data-flow analyses to locate its definition statement. The code block between the definition statement and the *if-then-else statement* is then referred to as a **Condition Triggering Block** (**CTB**). Then,

[3] We remind the readers that state-of-the-art studies (e.g., by Moser et al. [53] and Zeng et al. [73]) have further revealed that obfuscation (via reflective calls or opaque predicates) could be leveraged to complicate the inference of trigger conditions (e.g., changing the way how a system property is obtained from the system). We do not take obfuscation as a type of trigger condition but will only consider it as a technique that complicates the process of identifying trigger conditions. We will discuss the impact of obfuscation on our approach at the end of Section 5.



```
public boolean checkPackageName()
{
    $r1 = $r3.queryIntentActivities($r7, 65536);

    $z0 = $r1.isEmpty();

    if $z0 != 0 goto label2;

    return 1;
}

protected void onCreate(android.os.Bundle)
{
    $z0 = $r5.checkPackageName();                $r5

    //Trigger condition expression
    if $z0 == 0 goto label1;
    $r7 = $r3.getDeviceId(); //HSB
}
```

Fig. 2. The simplified working process of the trigger condition inference module. The code is presented in simplified Jimple, which is an intermediate representation of Soot [35]. Soot is the underline static analysis framework leveraged by HiSenDroid to achieve the backward data-flow analysis.

given a potential HSO, we check whether a system API is involved in the *definition statement* of the HSO's CTB. If not, we will regard this HSO as a false result and consequently will not consider it for further analysis.

When inferring the *definition statement*, inter-procedural analysis needs to be taken into account because the trigger conditions can be defined in other methods and transferred to the HSB via callee's returned values or caller's parameter values. Indeed, take the code snippet shown in Listing 1 as an example, the trigger condition is actually defined in method *checkPackageName()* despite the HSB is seated in the *onCreate()* method. Figure 2 illustrates the backward tracking flow showing how our approach identifies the trigger condition. When there is a method involved in the backward tracking flow, our data-flow analysis will keep tracking the method's caller object as it may be relevant to the definition of the trigger condition. For example, our analysis will keep tracking $r1 when statement *$r1.isEmpty()* is reached. If the method is a user-defined function, our data-flow analysis will further jump into the method and keep tracking its returned variables (all variables will be tracked if there are several return statements). The backward data-flow analysis will terminate if System APIs are identified, or Android's entry-point methods (such as UI callback methods or components' lifecycle methods) are reached. The analysis will also stop if the condition is linked to a constant value that is further not originated by *if-statements*.

### 3.3 Suspicious HSO Detection

The last module takes the outputs of the previous module to detect HSOs, following the rules presented in Definitions 1 and 2 (cf. Section 2). Unfortunately, these rules are not perfect and may introduce false-positive results that have similar characteristics of HSOs but are actually user intended behaviors. Indeed, for the same operations, under different circumstances, they could be flagged as conventional usages or suspicious operations and could lead to benign or user intended malicious behaviors. These false results include common programming patterns used in legitimate *if-then-else statements* (hereinafter referred to as *conventional usages*), which should be excluded by HiSenDroid. Therefore, we resort to building a list of conventional usages (or whitelist) and based on it, in the last module of HiSenDroid, we filter out non-malicious HSOs and only keep suspicious HSOs.

Nevertheless, we argue that it is non-trivial to understand the developer's intention behind the operations. Therefore, in this last module, in addition to automatically detecting suspicious HSOs, HiSenDroid goes one step deeper to also provide adequate details to explain why a suspicious HSO is flagged as such, i.e., what is the trigger condition, what is the logic of the *if condition*, and what are the sensitive behaviors triggered if the logic is fulfilled. This function is provided for helping security analysts understand whether the flagged HSOs should be regarded as malicious or not.

By leveraging HiSenDroid, in Section 4, we study and collect *conventional usages* in large sets of Android apps, whereas in Section 5, we put the emphasis on *suspicious* HSOs.

## 4 CONVENTIONAL USAGE ANALYSIS

The overall goal of this work is to detect HSOs so as to unveil the evasive technologies that are frequently leveraged to hide malicious behaviors. In this section, we evaluate our approach based on a large set of Android apps towards checking if our approach HiSenDroid is capable of fulfilling this goal. Specifically, in this section, we conduct an exploratory study of recent HSOs aiming to understand the current status quo of conventional usages and build a comprehensive list of conventional usages (to be used by HiSenDroid to discriminate suspicious HSOs from conventional usages).

Recall that our approach, in its last working step, takes as input a customizable list of conventional usages to filter out non-suspicious HSOs, which subsequently helps in saving significant security analysts' efforts as they now only need to scrutinize the retained small number of likely suspicious HSOs. Towards identifying such conventional usages, we apply a semi-automatic process to summarize based on their frequency of occurrence. The conventional usage whitelist is built based on reasonable assumptions that legitimate HSOs frequently appear in Android apps, including both malware[4] and goodware. We manually inspected the trigger APIs that have appeared more than 50 times in our dataset and determined if they should be categorized as a conventional usage. By doing so, we defined seven major categories of conventional usages. Also, the results of our manual analysis are cross-validated by two authors. The two authors first independently conduct the manual analysis (to discover knowledge with support evidence from various software artifacts). They then had physical meetings to discuss, merge, and finalize the results.

***Experimental Setup.*** We applied HiSenDroid (with the list of conventional usages set to be empty[5]) on a dataset that contains 10,000 malware samples (referred to as *malware set*) and 10,000 benign apps (referred to as *benign set*). The *malware set* was collected from VirusShare [6] from 2012 to 2020. To better reflect recent trends on the deceptive techniques used in malware samples, we only include the samples whose first seen date was on or after 2016. The malware samples were submitted to VirusTotal[6] for screening, and only the ones that have been labeled by more than five anti-virus engines (VirusTotal has hosted over 70 anti-virus scanners) were selected.

The *benign set* was randomly selected from a pool of more than 100,000 apps crawled from Google Play in 2019, which are further scanned to ensure non of them are tagged by VirusTotal.

Our tool has identified 45,342 HSOs (35,974 in the *malware set*, and 9,368 in the *benign set*) triggered by 54,152 conditions. Note that some HSOs may be triggered by more than one condition

[4] Malware is included because often not all of its code is malicious. It might contain a malicious payload but the other code could still remain benign.

[5] The experimental results should contain both conventional usages and suspicious HSOs.

[6] https://www.virustotal.com.

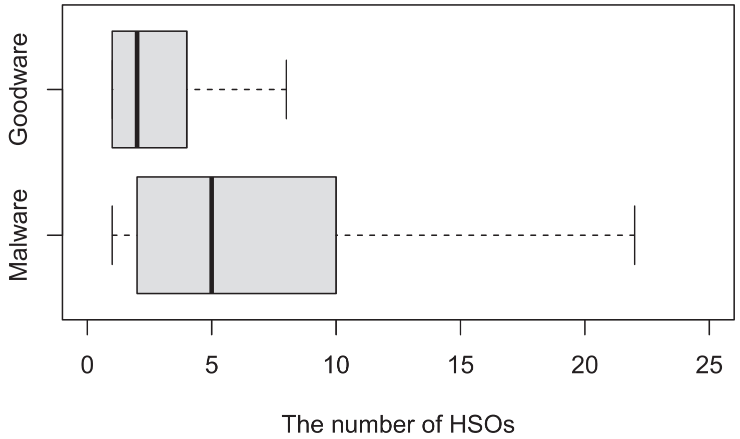


Fig. 3. Distribution of the number of HSOs in *benign set* and *malware set*.

(e.g., multiple conditions in a CTB that are connected by *AND* or *OR* operators). Towards evaluating the precision of HiSenDroid, i.e., the identified HSOs meet our previous rule definitions, we manually examine 20 randomly selected APKs from the total 8,107 apps that have been identified to contain at least one HSO. From these apps, our approach identified 157 (with a confidence level of 95% and a confidence interval of 7.81%) HSOs in total, among which 155 of them are eventually confirmed to be true HSOs, giving a precision of 98.7%. This result suggests that HiSenDroid is capable of identifying HSOs in Android APIs.

Figure 3 further presents the distribution of the number of HSOs detected in the apps from *benign set* and *malware set*. Expectedly, malware samples involve significantly more HSOs than that of benign apps, as confirmed by the *p-value* a **Mann–Whitney–Wilcoxon** (**MWW**) test at a significance level[7] at 0.001 [26]. This result suggests that HSOs are more favored by malware than benign apps. Hence, our community should pay more attention to the appearance of HSOs to help security analysts better dissect malicious apps.

Based on the previous experimental results, we manually analyzed the trigger conditions and the corresponding hidden operations to identify *conventional usages*, i.e., HSOs (at least based on our definition) that are actually legitimate and occur relatively often in Android apps. We first inspected the trigger APIs that have appeared more than 50 times in our dataset and determined if it is a *conventional usage*. By doing so, we identified seven major categories of *conventional usages*. Then we reviewed each of the rest of the cases to further filter out the other *conventional usages*. Finally, 43,141 *conventional usages* have been identified, out of which 40,412 cases belong to the seven major categories. As the whitelist is generated by manual analysis, it cannot cover all possible *conventional usages*. However, we believe that the majority of the *conventional usages* have been identified (i.e., from the seven categories), new special cases can always be added to the list and incorporated into HiSenDroid in the future. We now elaborate on the seven major categories, each with an example code snippet presented in Listing 2.

***SDK Version.*** With Android system update, new APIs are defined to replace old ones. To maintain the compatibility of apps across different Android versions, it is a common practice to check the SDK version before deciding the right API to use. Lines 2–7 show a simplified code snippet of a legitimate conventional usage that fulfills all the rules we defined for an HSO. The code checks

[7] Given a significance level $\alpha = 0.001$, if p-value $< \alpha$, there is one chance in a thousand that the difference between the two datasets is sue to a coincidence.

```
 1 // conventional usages - SDK version check
 2 public void addJavascriptInterface(Object var1, String var2) {
 3  if (VERSION.SDK_INT < 17) {
 4   TaoLog.e("HybridWebView", "addJavascriptInterface is disabled before API level 17 for security.");
 5  } else {
 6   super.addJavascriptInterface(var1, var2);
 7 }}
 8
 9 // conventional usages - User Interface
10 class ClickEvent implements View.OnClickListener {
11     public void onClick(View view) {
12         if(view.getId() == backButton.getId()){
13             webView.goBack()
14         }
15         else if (view.getId() == reloadButton.getId()){
16             webView.reload();
17 }}}
18
19 // conventional usages - File Handling
20 public static File inputstreamtofile(InputStream ins) {
21  File SDFile = Environment.getExternalStorageDirectory();
22  File desDir=new File(SDFile.getAbsolutePath());
23  File newFile=new File(desDir.getAbsolutePath() + File.separatorChar+"myPaint.png") ;
24  if(desDir.exists()){
25   OutputStream os = new FileOutputStream(newFile);
26   while ((bytesRead = ins.read(buffer, 0, 8192)) != -1) {
27    os.write(buffer, 0, bytesRead);
28 }}}
29
30 // conventional usages - Permission Check
31 public static String getDeviceInfo(Context context) {
32  if (checkPermission(context, Manifest.permission.READ_PHONE_STATE)) {
33   String device_id = tm.getDeviceId();
34  } else {
35   requestPermissions(context, new String[] {Manifest.permission.READ_PHONE_STATE}, REQUEST_CODE)
36 }}
37
38 // conventional usages - Network
39 public String g() {
40 var1 = ((ConnectivityManager)a.getSystemService("connectivity")) .getActiveNetworkInfo();
41 var9 = var1.getType();
42 if(var9 == 1){
43  var10 = ((WifiManager)a.getSystemService("wifi")).getDhcpInfo();
44 }}
45
46 // conventional usages - Intent Management
47 public void onReceive(final Context context, Intent intent) {
48  String action = intent.getAction();
49  if (action.equalsIgnoreCase ("android.net.conn.CONNECTIVITY_CHANGE") {
50   connectivityManager.getActiveNetworkInfo();
51 }}
52
53 // conventional usages - SharedPreferences
54 public class VpnAddressIp{
55  public SharedPreferences sp;
56  public String VPNAddress() {
57   sp = context.getSharedPreferences("SP", Context.MODE_PRIVATE);
58   VPNflag = sp.getInt("VPNFlag", 1);
59   VPNAddress vpnaddress = new VPNAddress(context);
60   if (VPNflag == 1) {
61    VPNAddressBean bean = vpnaddress.queryVPN(1);
62    networkaddress = bean.getNetwork();
63   }
64   return networkaddress;
65 }}
```

Listing 2. Examples of conventional usages.

whether the Android version is newer than *Android level 17* (line 3), if so, the app leverages the *addJavascriptInterface()* API to inject Javascript into the *WebView* (line 6), otherwise, it logs an error message (line 4) as the API is not available in the Android version lower than 17. While SDK version check commonly exists in both malware and benign apps (with 10,303 and 3,223 cases, respectively), this check does not intend to hide the behaviors within the *if-then-else* statement and hence should be excluded from the HSO results.

***User Interface.*** When the user interacts with UI widgets (e.g., press a button), it retrieves and compares the UI widget's *id* (i.e., a system API) to determine which widget has been fired. If there happened to be a sensitive API invoked in one of the branches' statements, this code block will be misidentified as HSO. Lines 10–17 show an example of a button's callback method, which checks the ID of the buttons (lines 12 and 15) and either goes back to the previous webpage (line 13) or reloads the current page (line 16). User Interface has 8,052 instances in the *malware set* and 2,426 instances in the *benign set*.

***File.*** The existence of a file or a directory is usually checked before file operations, such as reading and writing files. The code for checking file existence typically put the subsequent actions in one branch (where the file does exist), and shows an error message in the other branch (where the file does not exist). In some cases, it even has only the *if-branch*. Therefore, it satisfies the rules mentioned above and will be mistakenly identified as an HSO. Our results have observed 7,217 and 1,701 cases in our *malware set* and *benign set*, respectively. A file checking example can be found in lines 20–28, where the code checks the existence of an external storage (line 24), and copy an image there (lines 26 and 27).

***Permission.*** Since Android 6.0, the dangerous-level permissions need to be explicitly checked and requested before accessing the APIs protected by these permissions. The example code for checking permission can be found in lines 31–36. It first checks whether the app has been granted *READ_PHONE_STATE* permission (line 32). Then, the app either invoke the permission protected API (line 33) or request the missing permission (line 35) based on the check result. Even though a sensitive API *getDeviceId()* is called in one branch, which behaves quite differently than the other branch, it does not mean to hide this behavior. Therefore, it is regarded as a *conventional usage*. Permission check has appeared 6,727 and 936 times in the HSOs identified in the *malware set* and *benign set*, respectively.

***Network.*** Network information (e.g., network type, connection status) is always checked before performing network-related behaviors, ensuring that the network status is suitable for accomplishing the subsequent tasks. For example, the network type (e.g., WiFi, cellular) is checked before downloading large files, and if it is on the cellular network, the download will be suspended. Another example demonstrated in lines 39–44 examines the type of connected network (line 41) and get its DHCP information if the phone is connected to WiFi (1 is the value of ConnectivityManager#TYPE_WIFI). There are 5,224 and 744 identified HSO cases that are related to the *Network* in the *malware set* and the *benign set*, respectively.

***Intent.*** *Intent* is a crucial mechanism to assist the communication between different components in the Android system. *Intent* has various legitimate usages, including starting activities and services, passing data and properties, and so on. Lines 47–51 demonstrate a legitimate example of handling the callback method of receiving an *Intent*. In this example, it checks the *action* defined in the received *Intent*, and calls *getActiveNetworkInfo()* method (i.e., a sensitive API) if the *action* is *CONNECTIVITY_CHANGE*. There are 1,911 an 1,011 identified HSO cases that are related to the *Intent* in the *malware set* and the *benign set*, respectively.

***SharedPreferences.*** In Android system, data can be saved as <key, value> pairs and stores as a *SharedPreferences* object in a file that can be accessed by *getSharedPreferences()* interface. It provides a lightweight and easy-access data store mechanism, which is widely used in storing small collection of data, such as configurations of the app. Reading the values from the *SharedPreferences* and action accordingly is considered a legitimate behavior. Lines 54–65 illustrate an example of using *SharedPreferences*, where the code retrieves the value of a configuration item "VPNFlag" (line 58) from a *SharedPreferences* object named "SP" (line 57), and query the corresponding VPN services accordingly (lines 60–62). There are 878 and 358 cases involving the usage of *SharedPreferences* in our *malware set* and *benign set*, respectively.

**Completeness of conventional usages.** Since the conventional usage categories are summarized with manual efforts on a given set of apps, they may not be representative and thereby may not cover all possible cases. Therefore, in this work, we go one step deeper to further investigate the completeness of all the seven categories of conventional usages by applying our approach to another set of randomly selected 10,000 malware and 10,000 benign apps from AndroZoo [45]. We remind the readers that AndroZoo includes over 10 million Android apps that were collected from both the official Google Play store and several third-party app markets. To avoid potential biases in our results, we made additional efforts to remove potentially duplicated apps (i.e., different versions of the same app), and only the latest version is retained. For the 20,000 apps, we apply HiSenDroid to analyze these apps and inspect the trigger conditions that have appeared more than 50 times. We then manually determine if they are conventional usage. To do this, two of the authors spent ten person-days manually summarizing conventional usages (e.g., API-API or Key-API pairs). After manually checking the experimental results and the bytecode of apps, we have totally picked up 41,035 conventional usages, among which 38,920 cases fall into the predefined whitelist (with a success rate of 94.8%). This result shows that, despite testing on different apps, our whitelist is still quite stable and effective in eliminating conventional usage.

Apart from the aforementioned commonly appeared conventional usages, we further look into some of the uncommon conventional usages. Our manual observation confirms that those uncommon conventional usages are indeed legitimate HSOs that do not appear frequently in Android apps. We present two concrete examples of uncommon conventional usages to illustrate this concept. One example is that an app first checks if the directory of downloads exists (i.e., a standard directory to place files that have been downloaded by the user), and then automatically starts the download using the *android.app.DownloadManager#enqueue* API once the download manager is ready and connectivity is available. We consider it a conventional usage because it is against the second principle of suspicious HSO's definition: the user does not intend to hide such behavior. In addition, given that there exist several substitute ways of downloading files (e.g., Http request, URLConnection, BufferedInputStream, FileOutputStream), the native APIs lie in *android.app.DownloadManager* are not that commonly used by app developers. Thus, we regarded it as an uncommon conventional usage. As another example, the sensitive behavior of vibration could be triggered only when a user clicks a certain button. We consider it also a conventional usage because it involves non-hidden behaviors. In fact, if app developers intend to hide sensitive behaviors, it would be obvious that they won't use vibration functionality to notify users. Moreover, the usage of vibration is less common because it would annoy Android users, leading to a poor user experience. Such cases appear less than 50 times in our dataset and thus we regard them as uncommon conventional usage as well.

## 5 SUSPICIOUS HSO ANALYSIS

After eliminating conventional usages, all the remaining ones will be reported as suspicious HSOs. Among the 20,000 apps considered in this work, 1,304 of them, including 982 malware and 322 benign samples, were retained. These apps have been reported to contain in total 2,201 suspicious HSOs, with 1,790 and 441 from malware and benign apps, respectively. These numbers are recapped in Table 1. This experimental result shows that suspicious HSOs are widely present in real-world Android apps. Figure 4 further illustrates the distribution of suspicious HSOs in our dataset. On average, there are 2.0 and 1.4 HSOs in each malware sample and benign app, respectively.

In this work, the elimination of conventional usages is based on a pre-defined whitelist, which only includes recurrently presented HSOs in benign apps. Some less frequent yet still legitimated HSOs could have been overlooked and hence result in suspicious ones. Indeed, the remaining suspicious HSOs may not always be true positives (i.e., may contain a small number of false

Table 1. Number of Suspicious HSOs

| Initial Dataset | # HSOs | # Suspicious HSOs |
|---|---|---|
| 10,000 benign apps | 9,368 (in 3,071 apps) | 441 (in 322 apps) |
| 10,000 malicious apps | 35,974 (in 5,036 apps) | 1,790 (in 982 apps) |
| Total | 45,342 (in 8,107 apps) | 2,201 (in 1,304 apps) |

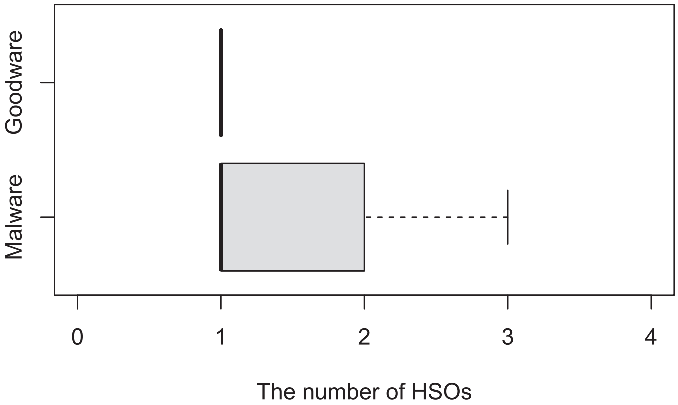


Fig. 4. Distribution of the number of *suspicious* HSOs in *benign set* and *malware set*.

positives). To this end, we go one step further to calculate the precision of our approach in pinpointing suspicious HSOs in Android apps. Unfortunately, there is no known ground truth available for evaluating HSO usage in Android apps. Thus, we resort to a manual process to calculate the precision. In this work, we manually inspected the bytecode of apps to see if HiSenDroid correctly and precisely identified the suspicious trigger rather than those commonly appeared code blocks for normal usage. Here, we identify truly suspicious behavior (i.e., confirmed to be true positive) only when the HSO is security-relevant and potentially brings harm to Android users. Specifically, we rely on two principles to identify truly suspicious HSOs: (1) the hidden behavior involves security-relevant APIs that are protected by Android permissions, classified following the latest Android API-permission mappings (cf. Section 3.1), and (2) the sensitive APIs are intentionally hidden under dedicated trigger conditions. As a result, we count those who meet the two aforementioned principles as true positives. For example, if an app first intends to retrieve Device ID, and when unsuccessful, tries to read the MAC address, we will consider it as a false positive because it is against the second principle: does not intend to hide such behavior. As another example, an app checks the build's fingerprint to see if it is running on popular emulators, and the sensitive behavior of retrieving subscriberId would be triggered only when it is not running in an emulator. We consider it as a true positive because it involves security-relevant APIs and there is sensitive behavior that is clearly hidden under trigger conditions. In our dataset, 1,304 apps have been reported to contain at least one HSO. Among the 1,304 apps, HiSenDroid has identified 14,394 HSOs, of which 2,231 of them are regarded as suspicious HSOs. By manually looking at each of those reported suspicious HSOs, we are able to confirm that 1,938 out of 2,231 of them are true positive results (or 293 of them cannot be confirmed without deeply examining the code), giving a precision of 86.8.

Recall that the conventional usages are excluded in this work through a whitelist built through empirical evidence, and the whitelist is only considered as a configuration option to our approach. We believe that the performance of detecting suspicious HSOs could be further improved if we are

Table 2. Categories of Trigger Conditions in HSO

| Category | Most Frequent Trigger API Examples |
|---|---|
| Time | util.Calendar#get<br>util.Date#getTime<br>util.Calendar#getTimeInMillis |
| System Properties | os.Build#MODEL<br>telephony.TelephonyManager#getSubscriberId<br>telephony.TelephonyManager#getDeviceId |
| Location | telephony.TelephonyManager#getSimCountryIso<br>telephony.TelephonyManager#getCellLocation<br>location.LocationManager#getLastKnownLocation |
| SMS Message | telephony.SmsManager#divideMessage<br>telephony.SmsManager#getDefault<br>telephony.SmsManager#getData |
| Package Manager | content.Context#getPackageManager<br>content.pm.PackageManager#getPackageInfo<br>content.pm.PackageManager#getApplicationInfo |
| Miscellaneous | android.widget.CheckBox#isChecked<br>android.app.KeyguardManager#isKeyguardLocked<br>java.net.NetworkInterface#getHardwareAddress |

able to construct a better whitelist of legitimate HSOs. This is nevertheless outside the scope of this work. We hence consider it as our future work.

## 5.1 Trigger Conditions

Known trigger types such as time-bomb and anti-emulator techniques have been broadly studied, specific algorithms for detecting such known trigger types have been developed [15, 27, 32]. Nevertheless, the community still lacks the understanding of unknown trigger types. We, therefore, investigate the most frequent triggering conditions in the suspicious HSOs detected by HiSenDroid.

HiSenDroid has identified 168 unique APIs that have been traced as the source of the triggers in detected suspicious HSOs. To make it much clearer, we present all these system properties trigger conditions and environment parameters trigger conditions in the artefact package.[8] We then manually categorize them according to the types of objects they accessed. Table 2 illustrates the trigger condition categories and examples of system APIs that are frequently leveraged to fulfill the trigger conditions discovered in HSOs. The top trigger condition categories include time (e.g., at a certain time of a day), SMS (e.g., when receives SMS of certain formats), location (e.g., if the device is in certain countries), system property (e.g., checks the device's manufacturer), and package manager (e.g., if specific apps are installed).

Here we elaborate on each trigger condition category with real-world suspicious HSO cases identified in our dataset.

***Time Triggers*** compare time-related properties (such as current system time, time zone, etc.) with hard-coded values to determine whether or not to execute the hidden sensitive behaviors. Listing 3 demonstrates a code snippet from app *com.wukongtv.wukongtv*,[9] which leverages time-related triggers to hide suspicious behaviors. When the first time the app launches, it writes the

[8] https://bitbucket.org/se_anonymous/hisendroid/src/master/experiments_results/.

[9] SHA-256:3397079daa388bdbcdcc42b6834d3c792bf5c80ad24491e3893de7cfc2b11db7.

timestamp into the SharedPreferences (i.e., *var0*). It then compares the current system time with the first launch time (line 6); if the time interval is greater than one day, it triggers the sensitive method *bq.f()* (line 7) that retrieves the information (e.g., package name and process name) of running tasks (lines 10–15). Doing so conceals the suspicious behaviors from automatic dynamic detection, which usually starts testing immediately after the app is installed.

```
 1 static void c(Context var0) {
 2  Calendar var10000 = Calendar.getInstance();
 3  int var2 = var10000.get(6) * 100;  //day_of_year
 4  var2 += var10000.get(11);          //hours_of_day
 5  // var0 is retrieved from SharedPreferences
 6  if (Math.abs(var0/100L - (long) (var2/100)) >= 1L) {
 7   ab.h = bq.f(var0);
 8 }}}
 9
10 public static Long[][] f(Context var0) {
11  var31 = var3.getRecentTasks(10, 1);
12  while(var31.iterator().hasNext()){
13   // get package name and process name of recent tasks
14   ...
15 }}
```

Listing 3. Code Example of Time Trigger.

***System Property Triggers*** leverage system properties, such as the phone model, the phone number, and hardware information, to limit the sensitive behaviors within specific device brands (e.g., Samsung) or types (e.g., real device). These triggers are also commonly adopted by anti-emulator techniques to detect the presence of emulators. Listing 4 demonstrates an anti-emulator example extracted from app *com.gwsoft.imusic.controller*,[10] which checks if the build's fingerprint contains specific strings that indicate popular emulators (line 2). The sensitive behavior of retrieving *subscriberId* (line 5) is only executed if it does not run in an emulator.

```
1 private static boolean a(Context var0) {
2  if (Build.FINGERPRINT.contains("vbox86p/vbox86p")&& !Build.FINGERPRINT.contains("ttVM_Hdragon/ttVM_Hdragon")&&
      !Build.FINGERPRINT.contains("generic/sdk/generic")&& !Build.FINGERPRINT.contains("generic_x86/sdk_x86/generic_x86")
3  // omit other strings that fingerprints popular Android emulators
4  ){
5   var2 = ((TelephonyManager)var0.getSystemService("phone")) .getSubscriberId();
6   var11.put("imsi", var2);}}
```

Listing 4. Code Example of System Property Trigger.

***SMS Triggers*** Utilize the content, type, and phone number of received SMS messages to hide sensitive behaviors. An example derived from app *com.fingersoft.hillmotor*[11] is shown in Listing 5. When an SMS message is received, it checks the originating address of the message (line 4). If it matches a pre-defined value (e.g., 10 or 11 etc in this example), the behavior that repeatedly sends a message (line 6) to the same number via a text message service.

```
1 //var2 is the originating address retrieved from SMS
2 //var3 is the message body
3 public boolean repeat(Context var1, String var2, String var3) {
4 if ((var2.startsWith("10") || var2.startsWith("11") || var2.startsWith("12")) && !var2.equals("114") &&
      !var2.equals("12306") && !var2.equals("116114") && !var2.equals("12580")) {
5 SmsManager var13 = SmsManager.getDefault();
6 var25.sendTextMessage(var2, (String)null, "Y", var16, var12);
7 }}
```

Listing 5. Code Example of SMS Trigger.

[10] SHA-256:8c679a7c57a7fbb355fb363d3784cc8380655701d482837869edd95f3a3ea470.

[11] SHA-256:95e1cf498dec79351a9d104f5e9fb0110c267e9eff0099ada475d8832a2afb7302521.

***Location Triggers*** obscure sensitive behaviors with fine grained (e.g., latitude and longitude) and coarse grained (e.g., country) location information. Listing 6 shows an example derived from *com.inter.apps.patqut.apk*[12] which queries the country code of the device (saved as *var1*), and checks if it is in Malaysia (line 4). If so, the app then triggers the *postLoginData2()* method (line 5), which retrieves the device's id (line 9) and hands it over to another activity for further malicious behaviors.

```
 1 TelephonyManager var3 = (TelephonyManager)this.getSystemService("phone");
 2 String var1 = var3.getSimCountryIso().toUpperCase();
 3 public void getin(String var1) {
 4  if (var1.equals("MY")) {
 5   this.postLoginData2();
 6 }}
 7
 8 public void postLoginData2() {
 9  String var2 = ((TelephonyManager)this.getSystemService("phone")) .getDeviceId();
10  // hand over the obtained DeviceId to a new activity
11  ...
12 }
```

Listing 6. Code Example of Location Trigger.

***Package Manager Triggers*** scan the list of installed apps and inspect if specific apps (usually anti-virus tools) are installed before conducting any sensitive behaviors. Listing 7 shows a code snippet taken from *flash15.1.apk*[13] which searches for *AhnLab V3 Mobile Plus 2.0* (i.e., an anti-virus tool) in the list of installed apps (line 1–7). If the anti-virus tool is not installed (line 10), it then starts its malicious behaviors. Specifically, it gets the package name of the current active activity (lines 11 and 12), and puts it into sleep if it is a bank app. After that, it launches a new activity that contains a phishing web page to steal user's bank credentials.

```
 1 private boolean judgeAV() {
 2  this.pm = this.getPackageManager();
 3  this.listAppcations = this.pm.getInstalledApplications(8192);
 4  for(int v = 0; v < listAppcations.size(); ++v) {
 5   if(listAppcations(v).name.equalsIgnoreCase("AhnLab V3 Mobile Plus 2.0")){
 6    return true;}
 7   return false;}
 8
 9 public void run() {
10 if (!AutBan.this.judgeAV()) {
11  List var2 = ((ActivityManager)AutBan.this.getSystemService ("activity")).getRunningTasks(1);
12  String var1 = ((RunningTaskInfo)var2.get(0)).topActivity .getPackageName();
13  // if the top activity is a bank app, it puts the activity into sleep and start a phising page
14  ...
15 }};
```

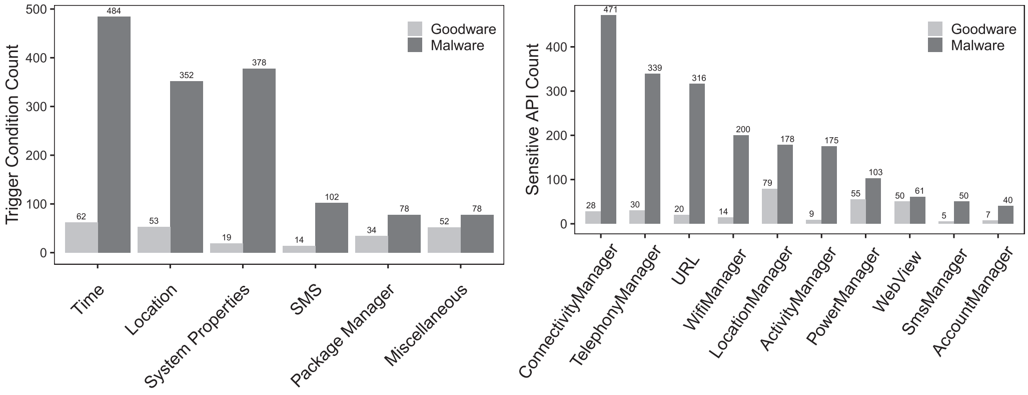
Listing 7. Code Example of Package Manager Trigger.

***Other Triggers.*** Besides the most frequent trigger categories, we also observed some sophisticated triggers specifically designed to counter automated dynamic testing approaches. Listing 8 shows an example taken from a music player app *com.gwsoft.imusic.controller.*[14] The app hides sensitive behaviors that retrieve the device's information (lines 8–12) behind a trigger that will only be fired when an item on the song list (i.e., *mCatalogSongsList*) is clicked (line 2). The trick here is that automated dynamic testing tools running on an emulator are likely not to have any music files and, therefore, will have no items on the list to click. Hence, only legitimate users who intend to use it to play music will have the chance to trigger the sensitive behavior.

[12] SHA-256:22c9d7738073a7ac8f9b58029057c2741e89faac76b623837db2f3a8bb2d93c5.

[13] SHA-256:fdaba7f032ee7ff9adf799713b25d4c2fef86ddbbe8709bf6ec021505b8f1d0d.

[14] SHA-256:8c679a7c57a7fbb355fb363d3784cc8380655701d482837869edd95f3a3ea470.



(a) Categories of Trigger Conditions in HSO.

(b) Top 10 Categories of Sensitive APIs in Malware.

Fig. 5. Categories of trigger conditions in HSO and sensitive APIs in malware.

```java
//contains at least one song in the list
public void onItemClick(AdapterView<?> var1, View var2, int var3, long var4) {
 if (mCatalogSongsList != null && var3 + -1 >= 0 && var3 + -1 < mCatalogSongsList.size()){
  CountlyAgent.onEvent(CuttingActivity.this,  "activity_diy_do_re", String.valueOf(var3));
}}

public static void onEvent(Context var0, String var1, String var2) {
 HashMap var3 = new HashMap;
 var3.put("phone", getIMSI());
 var3.put("ip", getLocalIpAddress());
 var3.put("app_version", versionName);
 var3.put("imei",getDeviceId());
}
```

Listing 8. Code Example of Other Trigger.

### 5.2 Sensitive APIs Involved in Suspicious HSOs

While the invocation of sensitive APIs does not necessarily mean it is malicious, sensitive APIs deliberately hidden in an HSB do raise suspicion. HiSenDroid has identified 134 unique hidden sensitive APIs that appeared 3,195 times in our dataset. Figure 5(b) presents the top ten classes of the most frequently invoked sensitive APIs in *malware set* and *benign set*. The most involved APIs are network related, including the ones in *URL*, *ConnectivityManager*, and *WifiManager* classes. The *WebView* ((displays web pages)) and *SmsManager* (manages SMS operations such as sending text messages) are also prevalently used in HSOs. Other commonly involved API classes include *PowerManager* (controls the power state of the device such as keeping the screen stay on), *LocationManager* (provides access to the system location services such as getting last known location), *ActivityManager* (gives information about activities and services such as getting running tasks on the phone), *TelephonyManager* (provides access to information of the telephony services such as phone number), and *AccountManager* (manages user's online accounts). The detailed most common APIs in HSOs can be found in Table 3.

Interestingly, while most of the API classes have significantly more instances in malware samples than benign apps, *WebView* is an exception. We, therefore, took an in-depth look into benign apps with WebView APIs in their HSOs and observed that 34 out of 50 cases are free apps that display advertisement web pages for revenue.

Table 3. Details of The Top 10 Classes of Hidden Sensitive APIs in HSO

| Class | Most Frequent Sensitive API Examples |
|---|---|
| ConnectivityManager | net.ConnectivityManager#getActiveNetworkInfo<br>net.ConnectivityManager#getNetworkInfo<br>net.ConnectivityManager#getAllNetworkInfo |
| TelephonyManager | telephony.TelephonyManager#getDeviceId<br>telephony.TelephonyManager#getSubscriberId<br>telephony.TelephonyManager#getCellLocation |
| URL | net.URL#openConnection<br>net.URL#getContent<br>net.URL#openStream |
| WifiManager | net.wifi.WifiManager#getScanResults<br>net.wifi.WifiManager#getConnectionInfo<br>net.wifi.WifiManager#getWifiState |
| LocationManager | location.LocationManager#getLastKnownLocation<br>location.LocationManager#requestLocationUpdates<br>location.LocationManager#getBestProvider |
| ActivityManager | app.ActivityManager#getRunningTasks<br>app.ActivityManager#getRecentTasks<br>app.ActivityManager#moveTaskToFront |
| PowerManager | os.PowerManager.WakeLock#release<br>os.PowerManager.WakeLock#acquire()<br>os.PowerManager.WakeLock#acquire(long) |
| WebView | webkit.WebView#setBackgroundColor<br>webkit.WebView#addJavascriptInterface<br>webkit.WebView#loadDataWithBaseURL |
| SmsManager | telephony.SmsManager#sendTextMessage<br>telephony.SmsManager#sendMultipartTextMessage<br>telephony.SmsManager#sendDataMessage |
| AccountManager | accounts.AccountManager#getAccountsByType<br>accounts.AccountManager#getAccounts<br>accounts.AccountManager#getUserData |

### 5.3 Trigger Condition to Hidden Sensitive API Pairs

We now investigate the relationships between trigger conditions and the hidden sensitive APIs accessed in their corresponding HSOs so as to identify common patterns leveraged by attackers to achieve malicious purposes. Figure 6 graphically summarizes such relationships, i.e., trigger-to-hidden-sensitive-API pairs, where each node represents an API in either the trigger conditions or the HSBs, while each edge denotes the connections between them. HiSenDroid has identified 404 nodes within which 346 are APIs in trigger conditions, 134 are APIs in HSBs, and 15 APIs exist in both triggers and HSBs. There are 2,847 edges found between them, which are illustrated in different colors according to their trigger conditions' categories.

Table 4 further details the top 10 pairs found in HSOs with their categories and counts. The most frequent HSO patterns are to hide network-related activities behind retrieving the phone's location. For instance, the top pattern that appeared 135 times in our dataset requests the SIM provider's country code. Based on the user's location, it then determines whether or not to open a web page, and what web pages (e.g., advertisement pages) to display to the user. Time-related HSO

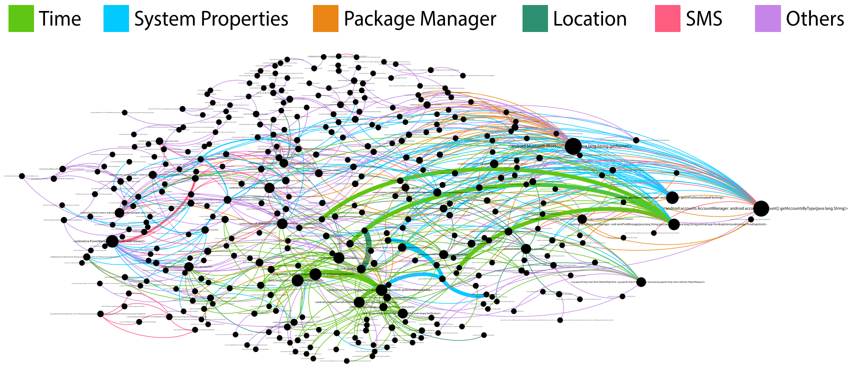


Fig. 6. The HSO trigger-sensitive API pairs graph.

Table 4. Top 10 Trigger Condition to Hidden Sensitive API Pairs

| Category | Trigger Condition APIs | Hidden Sensitive APIs | Counts |
|---|---|---|---|
| Location | TelephonyManager#getSimCountryIso | java.net.URL#openConnection | 135 |
| Location | TelephonyManager#getCellLocation | ConnectivityManager#getActiveNetworkInfo | 131 |
| Time | java.lang.System#currentTimeMillis | ConnectivityManager#getNetworkInfo | 80 |
| Time | java.lang.System#currentTimeMillis | ConnectivityManager#getAllNetworkInfo | 66 |
| Time | java.lang.System#currentTimeMillis | ConnectivityManager#getActiveNetworkInfo | 57 |
| System Properties | android.os.Build#MODEL | TelephonyManager#getSubscriberId | 54 |
| System Properties | android.os.Build#MODEL | Settings$System#putInt | 54 |
| SMS | android.os.Message#obj | PowerManager$WakeLock#release | 52 |
| Package Manager | PackageManager#getInstalledPackages | ActivityManager#getRunningTasks | 51 |
| Time | java.lang.System#currentTimeMillis | DefaultHttpClient#execute | 49 |

patterns are also widely found in the detected HSOs. They first compare the current system time with preset values. If the condition fulfills (e.g., the app is running for more than ten minutes), they try to initialize a network connection and send out the user's private information such as IMEI, phone number, and so on. More than 250 instances in our dataset leverage this pattern to steal users' private information stealthily. Other frequent HSO patterns on the top list are involved in anti-emulator tricks include checking the phone's model name and checking if specific apps are installed (which could indicate if it is an emulator) before acquiring sensitive information.

### 5.4 Suspicious HSOs in Third-party Code

Finally, we further look into the identified HSOs to check if they are introduced by app developers or reused by third-party libraries. For the sake of simplicity, we consider the code only located in the unique app package (also known as the app id) as developers newly implemented code while all the other code (i.e., in packages not connected with the app's domain name) as third-party code (e.g., third-party libraries). Among the 2,201 HSOs, surprisingly, over half of them (i.e., 1,342) are contributed by third-party code (i.e., 1,173 HSOs in malware and 169 in benign apps), among which malware tends to be more favored to introduce HSOs through third-party code than benign apps. This experimental evidence suggests that attackers have more incentives to achieve malicious behaviors through third-party code as it allows easy code reuse that makes it much easier to implement new malware.

### 5.5 Comparison with State-of-the-art

We now compare our approach with state-of-the-art works targeting the problem of detecting HSOs. To the best of our knowledge, there are two closely related approaches: HSOMiner [57] and

Table 5. The Comparison Results between HiSenDroid and TSOpen

| Tool Name | # Analyzed Goodware | # Analyzed Malware | # HSOs in Goodware | # HSOs in Malware | # Time HSOs | # Location HSOs | # SMS HSOs |
|---|---|---|---|---|---|---|---|
| HiSenDroid | 10,000 | 10,000 | 441 (in 322 apps) | 1,790 (in 982 apps) | 546 | 405 | 116 |
| TSOpen | 10,000 | 10,000 | 110 (in 51 apps) | 237 (in 123 apps) | 229 | 49 | 69 |
| Common | 10,000 | 10,000 | 71 | 194 | 186 | 48 | 31 |

TriggerScope [27]. Unfortunately, the source code of HSOMiner is not publicly available, and it is infeasible to compare against it because they trained data on the authors' labeled dataset, which has also not been publicly released. We have contacted the authors about launching their approach to analyze Android apps. Unfortunately, we have not yet received any response from them. Similarly, the authors of TriggerScope have also not made it publicly available. As a result, we cannot compare it with TriggerScope as well. Fortunately, Jordan Samhi has provided a re-implemented version[15] of TriggerScope based on the details given in its research article. The re-implemented version is named as TSOpen (referring to as the open implementation of TriggerScope) and has already been leveraged by previous studies [63]. In this work, we resort to comparing our approach with TriggerScope by actually comparing it with TSOpen.

To set up the experiments for a fair comparison, we run TSOpen on the same 10,000 malware and 10,000 benign apps selected in evaluating HiSenDroid in Section 4 (as indicated in the second and third columns in Table 5). The experiments are executed under the same environment, i.e., the same server, and the same timeout threshold (i.e., 20 minutes).

The experimental results are summarized in Table 5. Overall, the number of HSOs found by HiSenDroid in goodware and malware (i.e., 441 and 1,790, respectively) is larger than those found by the TSOpen (i.e., 110 and 237, respectively). Recall that when evaluating the performance of HiSenDroid at the beginning of Section 5, we have manually validated the 2,231 suspicious HSOs yielded by HiSenDroid, for which 1,938 are confirmed to be true positives, giving a precision of 86.8%. In this work, we further conduct the same manual validation for the results of TSOpen. Our manual validation confirms that TSOpen has at least correctly detected 90.2% of logic bombs. This result is expected as TSOpen only detects three types of HSOs (i.e., time, location, and SMS) while HiSenDroid aims at detecting a broader scope of HSOs. To enable a fair comparison,[16] in this work, we will only consider HiSenDroid's results falling in these three categories.

As highlighted in Table 5 (cf. Columns 6–8), HiSenDroid detects more HSOs in all of the three categories. Among the detected HSOs, we find that 265 HSOs (186, 48, and 31 in time, location, and SMS, respectively) were detected by both tools (as summarized in the fourth row in Table 5). Besides that, there are 802 HSOs (360, 357, and 85 in time, location, and SMS, respectively) exclusively detected by HiSenDroid, while still 82 HSOs (38, 1, and 43 in time, location, and SMS, respectively) identified by TSOpen are not flagged by HiSenDroid.

On further investigation, we found the reason why HiSenDroid failed in detecting the 82 HSOs is that HiSenDroid's definition of potentially-sensitive APIs is different from the definition in TSOpen. In this work, we consider all the APIs that are protected by Android permissions as potentially sensitive, while TSOpen takes a different approach to pre-select such a set of sensitive APIs.[17] Their set of sensitive APIs includes both permission-protected and permission-free APIs. For example, TSOpen treats the following two APIs, <android.content.BroadcastReceiver:

[15] https://github.com/JordanSamhi/TSOpen.

[16] We consider the original outputs of HiSenDroid and TSOpen for comparison since only a small number of their results could be false positive.

[17] The sensitive APIs are a part of internal implementation of TSOpen, which is not configurable.

void abortBroadcast()> and <android.os.Handler: boolean sendEmptyMessage(int)>, as sensitive APIs. However, HiSenDroid does not consider them as sensitive because they are not protected by permissions. Furthermore, considering that TriggerScope was published in 2016 and the Android API rapidly evolves, it is understandable that certain APIs (especially the latest ones) are not included, resulting in possibly less suspicious HSOs. Moreover, as claimed in their article, TriggerScope only focused on characterizing logic bombs on some given behaviors, while HiSenDroid treated each sensitive API in state-of-the-art Android API-permission mappings [7, 13, 14, 36] as a target API, leading to better performance in terms of both quantity and variety in detected HSOs, compared with TriggerScope.

### 5.6 Impact of Code Obfuscation

As experimentally revealed by Zeng [73] and Moser et al. [53], trigger conditions of HSOs could be obfuscated in order to evade the detection of advanced semantics-based malware analyzers. Therefore, we are interested in checking to what extent our approach is impacted by obfuscation, especially when applied to pinpoint HSOs in real-world Android apps. Since there is no existing dataset that is suitable for our experiment, we resort to preparing such a dataset from scratch, i.e., to form a set of obfuscated app pairs for which each pair contains a non-obfuscated app and its obfuscated counterpart. We start by randomly selecting 1,000 malware from our dataset and then apply Obfuscapk [10] on them to generate their obfuscated counterparts. Obfuscapk is a modular Python tool designed to directly obfuscate closed-source Android apps. Obfuscapk supports six types of obfuscation operations, which could be configured to achieve different granularities when obfuscating Android apps.

The six types of operations are summarized as follows:

(1) Nop: Insert junk code. Nop, short for no-operation, is a dedicated instruction that does nothing. This technique just inserts random nop instructions within every method implementation.
(2) Rename: operations that change the names of the used identifiers (classes, fields, and methods).
(3) Reorder: This technique consists of changing the order of basic blocks in the code. When a branch instruction is found, the condition is inverted (e.g., branch if lower than, becomes branch if greater or equal than) and the target basic blocks are reordered accordingly. Furthermore, it also randomly rearranges the code abusing goto instructions.
(4) Reflection: This technique analyzes the existing code looking for method invocations of the app, ignoring the calls to the Android framework (see AdvancedReflection). If it finds an instruction with a suitable method invocation (i.e., no constructor methods, public visibility, enough free registers ) such invocation is redirected to a custom method that will invoke the original method using the Reflection APIs.
(5) Advanced Reflection: Uses reflection to invoke dangerous APIs of the Android Framework. To find out if a method belongs to the Android Framework, Obfuscapk refers to the mapping discovered by Backes et al. [14]
(6) Encryption: packaging encrypted code/resources and decrypting them during the app execution. When Obfuscapk starts, it automatically generates a random secret key (32 characters long, using ASCII letters and digits) that will be used for encryption.

In this work, we are interested in checking the impact of all of these six types of operations on our approach. Hence, for each of the selected apps and each obfuscation type, we launch Obfuscapk to generate an obfuscated app. For the 1,000 selected apps, we expect to generate 6,000 obfuscated apps and eventually form 6,000 obfuscated app pairs. We then launch HiSenDroid to analyze those

Table 6. The Comparison Results of HiSenDroid before and after Obfuscation Techniques in Malware

| **Obfuscator** | **Nop** | **Rename** | **Reorder** | **Reflection** | **Advanced Reflection** | **Encryption** |
|---|---|---|---|---|---|---|
| # HSOs Before Obfuscation | 144 (in 821 apps) | 18 (in 332 apps) | 146 (in 792 apps) | 13 (in 345 apps) | 145 (in 821 apps) | 141 (in 811 apps) |
| # HSOs After Obfuscation | 144 (in 821 apps) | 18 (in 332 apps) | 146 (in 792 apps) | 4 (in 345 apps) | 64 (in 821 apps) | 141 (in 811 apps) |
| Common | 144 | 18 | 146 | 4 | 64 | 141 |

apps and compare the number of detected HSOs obtained on apps with and without obfuscation. Table 6 summarizes our experimental results.

Expectedly, except for reflection, our approach is resilient to all the other four obfuscation types. Our deep analysis reveals that the reason why HiSenDroid is unaffected by Nop obfuscator is that Nop obfuscator will only insert junk code, which is a dedicated instruction that does nothing. In terms of Rename and Reorder obfuscator, their code transformations will retain the functionality as the original APK and thus will not impact our approach. Also, the reason why the Encryption obfuscator has no effect on HiSenDroid is that it will only encrypt constant strings in code, which will not impact the data flow analysis of our approach. In terms of reflection obfuscator and advanced Reflection obfuscator, both trigger conditions and sensitive API invocations can be redirected to other code entities by reflection calls, while those entities cannot be always resolved statically since the reflection call targets may not be statically resolved, which would lead to false negatives of HiSenDroid. The remaining two types that have an impact on our approach are all related to reflection, which performs complicated code changes that will likely break the data flow processes. Nevertheless, even for reflection, our approach can still detect around one-third of HSOs.

To better mitigate the impact of reflection-based obfuscation on our approach, we further propose to strengthen the capability of HiSenDroid by integrating the state-of-the-art reflection analysis tool DroidRA to handle reflection usages [67]. After statically locating the reflective calls, DroidRA can transform a reflection-included Android app to a reflection-free version, where the located reflective calls will be represented by standard java calls. The newly generated reflection-free app would allow HiSenDroid to yield reflection-aware analysis results. Specifically, considering the 345 apps and 821 apps that are obfuscated by reflection calls and advanced reflection calls, respectively, we first apply DroidRA to convert them into 1,166 reflection-free apps. After that, we execute HiSenDroid to perform HSO analysis on these new apps and compare the number of detected HSOs obtained based on the original apps. As a result, HiSenDroid is able to detect all 13 reflection-relevant HSOs which are obfuscated with reflection obfuscation, while detecting 124 (with a success rate of 85.5%) reflection-relevant HSOs that are obfuscated with advanced reflection obfuscation. The reason why HiSenDroid fails on detecting a small portion of reflection-relevant HSOs is that DroidRA may not resolve all the advanced reflective calls. For example, DroidRA relies on COAL [56] solver to infer reflective calls, which might introduce false negatives, leading to reflection calls unresolved and thus can not be successfully detected by HiSenDroid. Nevertheless, our experimental result shows the capability of HiSenDroid in achieving most of the reflection-aware HSO detections.

## 6 IMPLICATION: DETECTION OF HIDDEN SENSITIVE DATA FLOWS

After being able to automatically detect suspicious HSOs, we now go one step further to investigate how such HSOs can bring security harms to users. There might be different security implications, in this work, we only focus on sensitive data leaks, which is also part of our initial attempts towards demonstrating the usefulness of identifying suspicious HSOs. Specifically, we are interested in detecting **hidden sensitive data flows (HSDFs)**, i.e., leaking sensitive data collected through HSOs. To the best of our knowledge, hidden sensitive data flow has not yet been explored by our community. Unfortunately, it has not even been clearly defined. To this end, we first define HSDF

Table 7. The List of Selected Source Methods that, by Default, are not Included by FlowDroid

| API Signature |
|---|
| android.net.wifi.WifiManager#getConnectionInfo() |
| android.app.ActivityManager#getRunningTasks |
| android.app.ActivityManager#getRecentTasks |
| android.accounts.AccountManager#getUserData |
| android.net.ConnectivityManager#getNetworkInfo |
| android.provider.Settings$System#getUriFor |
| android.telephony.TelephonyManager#getNeighboringCellInfo |
| android.telephony.TelephonyManager#getCellLocation |
| android.accounts.AccountManager#getAccountsByType |
| android.net.wifi.WifiManager#getScanResults |
| android.net.wifi.WifiManager#getConfiguredNetworks |
| java.net.URL#openConnection |
| android.net.ConnectivityManager#getAllNetworkInfo |
| android.net.VpnService#prepare |
| android.hardware.Camera#open |
| android.net.ConnectivityManager#getActiveNetworkInfo |

following the previous rules leveraged to define HSOs (cf. Section 2). Let $S$ denote a sensitive data flow (also known as a private data leak as mentioned in the FlowDroid work [12]), we consider that a sensitive data flow happens when a sensitive "tainted" information goes from a source (e.g., the API method *getDeviceId*) to a given sink (e.g., the API method *sendTextMessage*).

*Definition 3 (HSDF).* A sensitive data flow $S$ is an HSDF if the source of $S$ appears in the HSB of an HSO.

Although HSDFs have not yet been specifically exploited by the state-of-the-art, our community has proposed various approaches to detect general sensitive data-flows. One of the most famous approaches is FlowDroid [12], a state-of-the-art static analyzer that performs taint analysis to pinpoint sensitive data leaks flowing from a pre-defined set of *source* methods to *sink* methods. These *source* and *sink* methods can be easily customized. In this work, we leverage FlowDroid to detect sensitive data flows related to HSOs. If a sensitive data flow reported by FlowDroid has its source method invoked in an HSO, we regard it as an HSDF.

By applying FlowDroid[18] to 1,304 apps (982 malware and 322 goodware) involving suspicious HSOs, we find that 67 apps further involve HSDFs, accounting to in total 401 HSDFs. While manually checking the experimental results of FlowDroid and HiSenDroid, we find that 16 sensitive APIs, which are frequently invoked within HSOs to collect system information, are not taken into account by the source set of FlowDroid by default. These APIs (listed in Table 7), after manual confirmation, should still be considered as source methods by FlowDroid as they are responsible for retrieving sensitive data that should not be exposed to other parties. Here, to clarify, when doing the experiment, we include both of the default source and sink methods of FlowDroid and the additional sensitive APIs involved in HSOs in the SourceAndSink.txt file of FlowDroid. During the manual process, we have not found any sensitive API (i.e., involving dangerous operations) that should be additionally considered as a *sink* method by FlowDroid. Hence, we add the 16 APIs to

[18] In this work, the latest *development* branch of FlowDroid [3] is leveraged for the experiments. It should be roughly equivalent to the FlowDroid 2.8 release.

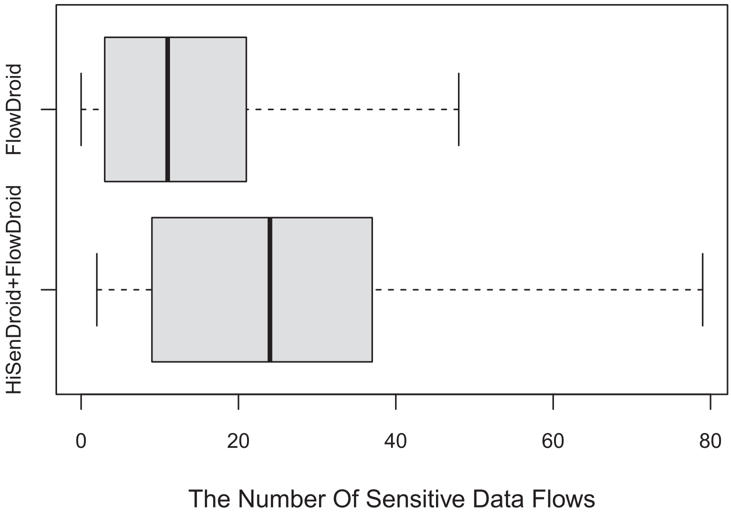


Fig. 7. Results of sensitive data flows in Android apps.

the source set of Flowdroid and keep its sink set unchanged (hereinafter referred to this version as FlowDroid + HiSenDroid) and relaunch it on the same set of apps. This time, we are able to disclose 1,110 HSDFs from 1,304 apps. This result shows that suspicious HSOs could be leveraged to leak users' sensitive information outside of their devices. As an example shown in Listing 5, the sensitive data *device id* and *subscriber id*, which are unique to the device and hence can be leveraged to uniquely track the phone, are eventually sent outside the device through a text message.

Considering general sensitive data-flows (SDF), we compare FlowDroid with HiSenDroid on the same dataset. In general, among the 1,304 apps, HisenDroid+FlowDroid detect 31,215 SDF, which is significantly larger than that of the original FlowDroid (which is 16,946). This result, as expected,[19] does experimentally demonstrate the effectiveness of our approach towards revealing more data flows in Android Apps. Our experimental results are illustrated in Figure 7, which indicates the distribution of the number of sensitive data flows in each app yielded by HisenDroid + FlowDroid and HiSenDroid. This result shows that FlowDroid + HiSenDroid has significantly improved the original results of FlowDroid, which shows the usefulness of our identified HSOs and suggests that there is a strong need to characterize HSOs.

## 7 LIMITATIONS

The main limitation of our approach lies in the backward data-flow analysis, which applies only context-insensitive analysis and thereby may lead to imprecise results. Furthermore, at the moment, our approach is not aware of dynamically loaded code, reflectively accessed methods, and native code. Subsequently, HiSenDroid may overlook certain app features and hence result in false-negative results.

Second, HiSenDroid data-flow analysis may be susceptible to obfuscation techniques. According to some former research works [29, 61, 64], obfuscation (especially those involving complicated changes in the program code) may cause false negatives of the static analysis approach. Indeed, as demonstrated by Moser [53], obfuscation is actually a challenge for almost all static program

[19] We remind the readers that, in this work, we did not improve FlowDroid by itself but only enlarged its *source* set as some of the sensitive APIs, which are favored by HSOs, are overlooked by FlowDroid.

analyzers. Just like all prior efforts on static analysis of HSOs [27, 57], we do not consider the apps whose branch conditions have been deeply obfuscated. Fortunately, the majority of obfuscations applied to Android apps only involve basic transformations (such as renaming [24]) that do not involve complicated code changes (e.g., structural or logic changes, or invoke sensitive code through reflections), which will not impact the analysis of our approach. This has also been confirmed by our exploratory study towards understanding the impact of obfuscation on our approach, as discussed in Section 5.6. Considering reflection obfuscation, integrating DroidRA with HiSenDroid as a pipeline is demonstrated to be effective in eliminating the impact of reflection calls. Therefore, we believe that the technical capabilities and our results would not be significantly impacted by code obfuscation. Nevertheless, as part of our future work, we plan to integrate other approaches developed by our fellow researchers to mitigate these long-standing challenges, e.g., by applying DroidRA [40, 67] to mitigate the impact of reflection-enhanced code obfuscations.

Although summarized from many sensitive operations, the definition of HSO rules may not be perfect. Indeed, on the one hand, the set of sensitive operations considered for summarization may not be representative, and the set of apps leveraged to obtain such sensitive operations may not be represented as well. On the other hand, the manual analysis leveraged to summarize the rules may contain errors since it is known that human efforts are prone to errors. Apart from that, the definition of HSO is based on empirical evidence that might not be perfect. There might be complicated cases that do not follow the definition but still manifest themselves as hidden sensitive behaviors in practice, leading to false negatives. This limitation can also apply to the conventional usage analyses since the list of conventional usages is manually summarized based on a given set of apps. The subsequent outputs (i.e., whitelist) may not be representative. Nonetheless, our follow-up study using a set of 20,000 new apps has shown that this impact is negligible. Furthermore, in this work, we have attempted to provide detailed insights to explain why HSOs are reported as such. This knowledge is expected to be useful for practitioners and researchers to characterize conventional usages and for security analysts to understand suspicious HSOs.

Moreover, since the original implementation of TriggerScope is not publicly available, we have resorted to an open re-implementation version of TriggerScope to compare our approach against it. This alternative decision may result in possible biases as the re-implementation may not really represent the original version. Unfortunately, the re-implemented version is the only source we can publicly locate to fulfill the comparison. As of our future work, we plan to also evaluate the reliability of the re-implementation of TriggerScope so as to mitigate potential biases, if any.

Last but not the least, the performance of the hidden sensitive data flow analysis may be impacted by the collection of sensitive APIs (i.e., sources). On one hand, some sensitive APIs, especially the latest ones, might be overlooked by FlowDroid and hence cannot be considered for pinpointing potential leaks, leading to false-negative results. In this work, our experimental results have confirmed this. On the other hand, some historically sensitive APIs included in FlowDroid's source list might be deprecated and subsequently removed from a certain Android API version [42]. There is hence no need to include them when analyzing apps targeting higher API versions, as these APIs will not be used anymore, not even mentioning causing sensitive data leaks. To overcome these impacts, we believe there is a need to keep updating FlowDroid's list of sensitive APIs, in order to achieve a more effective and sound sensitive data flow analysis for Android apps. Furthermore, ideally, FlowDroid should also not be expected to include APIs that are released after itself.

## 8 RELATED WORK

HSOs have long existed in Android malware as evasive technologies have widely been used by attackers to hide their malicious behaviors. Our research community has hence proposed various approaches to tackle these issues. We now discuss some of the representative works from two angles, including the evasive techniques that have been proposed to hide malicious code from being identified, and the detection methods proposed to pinpoint such evasive techniques.

**Evasive Techniques.** There has been a number of research works on hiding malicious behavior from detection, most of which focus on evading the dynamic test platforms such as virtual machines and emulators. Early works target the Windows platform [17], while recently the trend has been moved to Android [18, 21, 31, 48, 50, 59, 68]. These evasive techniques detect the presence of a simulated environment by either looking into the system properties of the testing platform (e.g., system fingerprints, hardware capabilities) [18, 59, 68], or leveraging a reverse Turing test that examines if the app interacts with a human user [21]. For instance, Diao et al. [21] observed that programmed interaction has specific patterns of input and interaction frequency, which is different from real users. Overall, the evasive techniques usually hide malicious activities in an *if-then-else statement*. The hidden malicious behavior will only be set off when certain conditions are fulfilled (e.g., not in an emulator); otherwise dummy benign operations are triggered. The prevalence of such evasive techniques motivated us to investigate the HSOs in Android apps and propose HiSenDroid to detect them.

**Detection of Evasive Techniques.** The pervasive evasive techniques (e.g., anti-emulator techniques) have motivated the research community to take countermeasures. Great effort has been spent on detecting known types of hidden behaviors that hampers the dynamic analysis process. These works include detecting anti-emulator techniques [15, 32, 33, 44] and generic logic-bombs [16, 20, 27, 58, 75]. The approaches of detecting anti-emulator techniques compare the behavioral deviation of the tested apps on the various environments when feeding them the same input. The fundamental idea is that if the app behaves differently in different environments, it is likely trying to evade one or more analysis platforms (usually referred to as bare-metal analysis in the literature) [15, 32, 33, 44]. While these early works investigate a critical category of hidden operations (i.e., anti-emulator), the proposed methods lack generalization that cannot be applied to detect other types of hidden operations emerging recently.

Besides the detection of anti-emulator techniques, several works are focusing on uncovering other trigger-based behaviors. These approaches leverage symbolic execution or static code analysis and instrumentation to expose the hidden branches in an *if-then-else statement* [16, 20, 27, 58, 75]. As examples, Zheng et al. [75] proposed to leverage a static analysis approach to retrieve all UI related events, and use dynamic testing to trigger them and log the invocation of sensitive APIs. Unlike HiSenDroid that leverages static analysis, the dynamic analysis-based approach introduces significant system- and time-overhead. The coverage of the dynamic analysis is also in question. Fratantonio et al. [27] proposed TriggerScope to detect hidden triggered behaviors based on the observation that certain triggers (i.e., time, location, and SMS related triggers) always involve the comparison of specific types of input (i.e., system time, system location, and received SMS). Symbolic execution is then leveraged to detect such narrow conditions. While TriggerScope is effective in detecting the above-mentioned three types of logic bombs, it cannot be generalized to detect hidden operations triggered by other types of conditions, such as system property, which has been found pervasive in Android apps.

Similar to HiSenDroid, another line of work attempts to detect unknown types of trigger-based behaviors [57, 69]. A prominent example is HSOMiner [57], which extracts static characteristics of hidden behaviors as features and trains a machine learning model to identify the code blocks that observe similar patterns. The major differences between our work and HSOMiner are twofold. First, HSOMiner requires a large number of manually labeled training samples, which involves extensive human experts' effort. Its performance also heavily relies on the manually labeled training data, which is prone to errors. Our method, on the other hand, is an automatic process without human intervention. Second, HSOMiner, as a machine learning-based approach, lacks explanations of the decisions. In contrast, our static code analysis-based approach outputs the full call traces of detected HSOs, and provides more detailed information for further analysis.

## 9 CONCLUSION

In this work, we present to the community a prototype tool called HiSenDroid, which performs a static code analysis to uncover HSOs that will only be triggered under special circumstances such as at a specific location or in a certain time period. Additionally, HiSenDroid goes one step deeper to provide details aiming at helping security analysts understand why a given HSO is flagged as such. Experimental results over 20,000 apps, including both malicious and benign apps, show that HSOs are indeed quite frequently presented in Android apps and HiSenDroid is effective in automatically discovering them. Moreover, with the help of FlowDroid, a state-of-the-art static taint analyzer, we further experimentally find that HSOs could eventually lead to privacy leaks.

## ACKNOWLEDGMENTS

The authors would like to thank the anonymous TOSEM reviewers who have provided insightful and constructive comments, which have been extremely useful for helping in improving this manuscript.

## REFERENCES

[1] 2022. Android Security: Adding Tampering Detection to Your App. Retrieved November 20, 2021 from https://www.airpair.com/android/posts/adding-tampering-detection-to-your-android-app#4-1-emulator. Accessed December 27, 2022.

[2] 2022. Android.hehe: Malware Now Disconnects Phone Calls. Retrieved November 20, 2021 from https://www.fireeye.com/blog/threat-research/2014/01/android-hehemalware-now-disconnects-phone-calls.html. Accessed December 27, 2022.

[3] 2022. FlowDroid Development Branch. Retrieved October 14, 2021 from https://github.com/secure-software-engineering/FlowDroid/tree/develop. Accessed December 27, 2022.

[4] 2022. Hacking Team RCSAndroid Spying Tool Listens to Calls; Roots Devices to Get in. Retrieved November 20, 2021 from http://en.hackdig.com/07/26261.htm. Accessed December 27, 2022.

[5] 2022. HiSenDroid. https://bitbucket.org/se_anonymous/workspace/projects/HIS. Accessed December 27, 2022.

[6] 2022. Virusshare. http://virusshare.com/. Accessed December 27, 2022.

[7] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. 2018. Precise android api protection mapping derivation and reasoning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1151–1164.

[8] 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 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories*. IEEE, 468–471.

[9] William Stofega Anthony Scarsella. 2022. Worldwide Smartphone Market Shares. https://www.idc.com/getdoc.jsp?containerId=US48435122&pageType=PRINTFRIENDLY. Accessed December 27, 2022.

[10] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. 2020. Obfuscapk: An open-source black-box obfuscation tool for Android apps. *SoftwareX* 11 (2020), 100403.

[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 *Proceedings of the Ndss*. 23–26.

[12] 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. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[13] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. 217–228.

[14] Michael Backes, Sven Bugiel, Erik Derr, Patrick McDaniel, Damien Octeau, and Sebastian Weisgerber. 2016. On demystifying the android application framework: {Re-Visiting} Android permission specification analysis. In *Proceedings of the 25th USENIX Security Symposium*. 1101–1118.

[15] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2010. Efficient detection of split personalities in malware. In *Proceedings of the NDSS*. Citeseer.

[16] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. 2008. Automatically identifying trigger-based behavior in malware. In *Proceedings of the Botnet Detection*. Springer, 65–88.

[17] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the 2008 IEEE International Conference on Dependable Systems and Networks with FTCS and DCC*. IEEE, 177–186.

[18] Valerio Costamagna, Cong Zheng, and Heqing Huang. 2018. Identifying and evading Android sandbox through usage-profile based fingerprints. In *Proceedings of the 1st Workshop on Radical and Experiential Security*. 17–23.

[19] Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2010. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the 19th International Conference on World Wide Web*. 281–290.

[20] Jedidiah R. Crandall, Gary Wassermann, Daniela A. S. de Oliveira, Zhendong Su, S. Felix Wu, and Frederic T. Chong. 2006. Temporal search: Detecting hidden malware timebombs with virtual machines. *ACM SIGOPS Operating Systems Review* 40, 5 (2006), 25–36.

[21] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. 2016. Evading android runtime analysis through detecting programmed interactions. In *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 159–164.

[22] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F. Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. 2018. Frauddroid: Automated ad fraud detection for android apps. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 257–268.

[23] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Guoai Xu, and Shaodong Zhang. 2018. How do mobile apps violate the behavioral policy of advertisement libraries?. In *Proceedings of the 19th International Workshop on Mobile Computing Systems and Applications*. 75–80.

[24] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. 2018. Understanding android obfuscation techniques: A large-scale investigation in the wild. In *Proceedings of the International Conference on Security and Privacy in Communication Systems*. Springer, 172–192.

[25] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2008. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys* 44, 2 (2008), 1–42.

[26] Michael P. Fay and Michael A. Proschan. 2010. Wilcoxon-mann-whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics Surveys* 4 (2010), 1.

[27] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. Triggerscope: Towards detecting logic bombs in android applications. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy*. IEEE, 377–396.

[28] Jun Gao, Li Li, Pingfan Kong, Tegawendé F. Bissyandé, and Jacques Klein. 2020. Borrowing your enemy's arrows: The case of code reuse in Android via direct inter-app code invocation. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[29] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini. 2020. Hidden in plain sight: Obfuscated strings threatening your privacy. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 694–707.

[30] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information flow analysis of android applications in droidsafe. In *Proceedings of the NDSS*. 110.

[31] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. 2014. Morpheus: Automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*. 216–225.

[32] Dhilung Kirat and Giovanni Vigna. 2015. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 769–780.

[33] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. Barecloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd {USENIX} Security Symposium*. 287–301.

[34] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. 2018. Automated testing of Android apps: A systematic literature review. *IEEE Transactions on Reliability* 68, 1 (2018), 45–66.

[35] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The soot framework for Java program analysis: A retrospective. In *Proceedings of the Cetus Users and Compiler Infastructure Workshop*. 35.

[36] Chaoran Li, Xiao Chen, Ruoxi Sun, Jason Xue, Sheng Wen, Muhammad Ejaz Ahmed, Seyit Camtepe, and Yang Xiang. 2022. Cross-language android permission specification. In *Proceedings of the 30th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[37] Li Li, Kevin Allix, Daoyuan Li, Alexandre Bartel, Tegawendé F. Bissyandé, and Jacques Klein. 2015. Potential component leaks in Android apps: An investigation into a new feature set for malware detection. In *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability, and Security*. IEEE, 195–200.

[38] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. 2015. IccTA: Detecting inter-component privacy leaks in Android apps. In *Proceedings of the 37th International Conference on Software Engineering*.

[39] Li Li, Tegawendé F. Bissyandé, and Jacques Klein. 2019. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering* 47, 4 (2019), 676–693.

[40] Li Li, Tegawendé F. Bissyandé, Damien Octeau, and Jacques Klein. 2016. Droidra: Taming reflection to support whole-program analysis of android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 318–329.

[41] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. 2017. Static analysis of Android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95.

[42] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. 2018. Cid: Automating the detection of api-related compatibility issues in android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–163.

[43] Li Li, Daoyuan Li, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. 2017. Understanding Android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security* 12, 6 (2017), 1269–1284.

[44] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. 2011. Detecting environment-sensitive malware. In *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*. Springer, 338–357.

[45] Pei Liu, Li Li, Yanjie Zhao, Xiaoyu Sun, and John Grundy. 2020. Androzooopen: Collecting large-scale open source android apps for the research community. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 548–552.

[46] Tianming Liu, Haoyu Wang, Li Li, Xiapu Luo, Feng Dong, Yao Guo, Liu Wang, Tegawendé F. Bissyandé, and Jacques Klein. 2020. MadDroid: Characterising and detecting devious Ad content for android apps. In *Proceedings of the Web Conference 2020*.

[47] Yonghui Liu, Li Li, Pingfan Kong, Xiaoyu Sun, and Tegawendé F. Bissyandé. 2021. A first look at security risks of Android TV apps. In *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops*. IEEE, 59–64.

[48] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep learning for android malware defenses: A systematic literature review. arXiv preprint arXiv:2103.05292. (2021).

[49] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2021. Deep learning for android malware defenses: A systematic literature review. arXiv preprint arXiv:2103.05292. (2021).

[50] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Explainable AI for android malware detection: Towards understanding why the models perform so well?. In *Proceedings of the 33rd IEEE International Symposium on Software Reliability Engineering*.

[51] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: A manifesto. *Communications of the ACM* 58, 2 (2015), 44–46.

[52] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2016. Mamadroid: Detecting android malware by building markov chains of behavioral models. arXiv preprint arXiv:1612.04433. (2016).

[53] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference*. IEEE, 421–430.

[54] Ravshanbek Norboev, Zakia Hossain, Lannan Luo, and Qiang Zeng. 2017. *On the Robustness of Stochastic Stealthy Network against Android App Repackaging*. Technical Report. Temple University.

[55] Jon Oberheide and Charlie Miller. 2012. Dissecting the android bouncer. *SummerCon2012, New York* 95 (2012), 110.

[56] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 77–88.

[57] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. 2017. Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in android apps. In *Proceedings of the NDSS*.

[58] Dorottya Papp, Thorsten Tarrach, and Levente Buttyán. 2019. Towards detecting trigger-based behavior in binaries: Uncovering the correct environment. In *Proceedings of the International Conference on Software Engineering and Formal Methods*. Springer, 491–509.

[59] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage against the virtual machine: Hindering dynamic analysis of android malware. In *Proceedings of the 7th European Workshop on System Security*. 1–6.

[60] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis.* Association for Computing Machinery, 176–186. DOI: https://doi.org/10.1145/3213846.3213873

[61] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting runtime values in android applications that feature anti-analysis techniques. In *Proceedings of the NDSS*.

[62] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering*. IEEE, 300–311.

[63] Jordan Samhi and Alexandre Bartel. 2021. On the (In) effectiveness of static logic bomb detector for android apps. arXiv preprint arXiv:2108.10381. (2021).

[64] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. 2022. JuCify: A step towards Android code unification for enhanced static analysis. In *Proceedings of the 44th International Conference on Software Engineering*. 1232–1244.

[65] Xiaoyu Sun, Xiao Chen, Kui Liu, Sheng Wen, Li Li, and John Grundy. 2021. Characterizing sensor leaks in android apps. In *Proceedings of the 2021 IEEE 32nd International Symposium on Software Reliability Engineering*. IEEE, 498–509.

[66] Xiaoyu Sun, Xiao Chen, Yanjie Zhao, Pei Liu, John Grundy, and Li Li. 2022. Mining Android API usage to generate unit test cases for pinpointing compatibility issues. arXiv preprint arXiv:2208.13417. (2022).

[67] Xiaoyu Sun, Li Li, Tegawendé F. Bissyandé, Jacques Klein, Damien Octeau, and John Grundy. 2021. Taming reflection: An essential step toward whole-program analysis of Android apps. *ACM Transactions on Software Engineering and Methodology* 30, 3 (2021), 1–36.

[68] Timothy Vidas and Nicolas Christin. 2014. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer, and Communications Security*. 447–458.

[69] Xiaolei Wang, Sencun Zhu, Dehua Zhou, and Yuexiang Yang. 2017. Droid-AntiRM: Taming control flow anti-analysis to support automated dynamic analysis of android malware. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. 350–361.

[70] Fengguo Wei, Sankardas Roy, and Xinming Ou. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1329–1341.

[71] Guosheng Xu, Yangyu Hu, Qian Guo, Ren He, Li Li, Guoai Xu, Zhihui Han, and Haoyu Wang. 2020. Dissecting mobile offerwall advertisements: An explorative study. In *Proceedings of the 2020 IEEE 20th International Conference on Software Quality, Reliability, and Security*. 518–526. DOI: https://doi.org/10.1109/QRS51102.2020.00072

[72] Guosheng Xu, Siyi Li, Hao Zhou, Shucen Liu, Yutian Tang, Li Li, Xiapu Luo, Xusheng Xiao, Guoai Xu, and Haoyu Wang. 2022. Lie to me: Abusing the mobile content sharing service for fun and profit. In *Proceedings of the ACM Web Conference 2022*. 3327–3335.

[73] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, and Zhoujun Li. 2018. Resilient decentralized android application repackaging detection using logic bombs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 50–61.

[74] Yanjie Zhao, Li Li, Haoyu Wang, Haipeng Cai, Tegawende Bissyande, Jacques Klein, and John Grundy. 2021. On the impact of sample duplication in machine learning based Android malware detection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–38.

[75] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. 93–104.

Received 19 April 2022; revised 5 October 2022; accepted 13 October 2022