Dark Hazard: Learning-based, Large-scale Discovery of Hidden Sensitive Operations in Android Apps

Xiaorui Pan*, Xueqiang Wang*, Yue Duan†, XiaoFeng Wang*, and Heng Yin†
*Indiana University Bloomington
Email: {xiaopan, xw48, xw7}@indiana.edu
†University of California, Riverside
Email: yduan005@ucr.edu, heng@cs.ucr.edu

***Abstract*—Hidden sensitive operations (HSO) such as stealing privacy user data upon receiving an SMS message are increasingly utilized by mobile malware and other potentially-harmful apps (PHAs) to evade detection. Identification of such behaviors is hard, due to the challenge in triggering them during an app's runtime. Current static approaches rely on the trigger conditions or hidden behaviors known beforehand and therefore cannot capture previously unknown HSO activities. Also these techniques tend to be computationally intensive and therefore less suitable for analyzing a large number of apps. As a result, our understanding of real-world HSO today is still limited, not to mention effective means to mitigate this threat.**

In this paper, we present HSOMINER, an innovative machine-learning based program analysis technique that enables a large-scale discovery of unknown HSO activities. Our approach leverages a set of program features that characterize an HSO branch[1] and can be relatively easy to extract from an app. These features summarize a set of unique observations about an HSO condition, its paths and the relations between them, and are designed to be general for finding hidden suspicious behaviors. Particularly, we found that a trigger condition is less likely to relate to the path of its branch through data flows or shared resources, compared with a legitimate branch. Also, the behaviors exhibited by the two paths of an HSO branch tend to be conspicuously different (innocent on one side and sinister on the other). Most importantly, even though these individual features are not sufficiently accurate for capturing HSO on their own, collectively they are shown to be highly effective in identifying such behaviors. This differentiating power is harnessed by HSOMINER to classify Android apps, which achieves a high precision (>98%) and coverage (>94%), and is also efficient as discovered in our experiments. The new tool was further used in a measurement study involving 338,354 real-world apps, the largest one ever conducted on suspicious hidden operations. Our research brought to light the pervasiveness of HSO activities, which are present in 18.7% of the apps we analyzed, surprising trigger conditions (e.g., click on a certain region of a view) and behaviors (e.g., hiding operations in a dynamically generated receiver), which help better understand the problem and contribute to more effective defense against this new threat to the mobile platform.

[1] A **branch**, unless otherwise specified, refers to a branching structure, which contains a **condition** and multiple **paths**.

NDSS '17, 26 February - 1 March 2017, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2017.23265

I. INTRODUCTION

The permeation of mobile technologies today also exposes their users to new kinds of security and privacy risks. Notably in the past years, mobile threats continues to be on the rise [12], not only from malware and other potentially harmful apps (PHA) such as back-doors, fraud apps, ransomware, spyware, etc., as reported by Google [23], but sometime even from large organizations' products that also exhibit unexpected behaviors like collecting private information (e.g., precise locations) without consent, installing unwanted programs, aggressively advertising, etc. To counteract these threats, major app stores today have beefed up their security vetting, putting various malware scans in place. Particularly, Google Play and other big stores run submitted apps for a short period of time to catch their suspicious behaviors [43]. While such protection works on less sophisticated PHAs, it leads to further evolution of attack techniques, bringing in a new set of apps that *deliberately hide their sensitive behaviors behind the events triggered only in their target situations*. For example, a PHA only pops up advertisements and collects a user's contacts when it runs on a physical device (not an emulator) and interacts with a human (Section II). Understanding and effective detection of these *hidden sensitive operations* (HSO, including both hidden behaviors and their triggering conditions) is becoming the new frontier in the fight against mobile malice.

HSO in mobile apps. Actually, HSO has been extensively studied in desktop malware, for example, attempts to identify the presence of a virtual machine (VM) and change behaviors to evade detection [19] [9]. Similar tricks are also played by mobile PHAs and in some cases, legitimate apps. An example is gaming apps, which stop providing services when they run in an emulator. On the other hand, the unique software and hardware resources on mobile devices enable apps to cover their behaviors with a wider spectrum of *triggers*, that is, conditions under which the hidden operations will be performed. Examples include locations or SMS messages (e.g., collecting personal information only at a specific location), as reported by the prior studies [34], and user input patterns, system servers and other system events, as newly discovered in our study (Section IV). In general, although it is perceived that HSO does exist in mobile ecosystems, little has been done so far to gain an in-depth understanding of its security impact and technical trend, as well as unexpected tricks already out there.

This lack of understanding is mainly attributed to the challenges in finding the hidden operations, particularly previously unknown ones, on a large scale. Most desktop HSO malware today has been captured using some levels of dynamic analysis, the only viable solution for finding evading behaviors from binary code. The analysis, however, is limited by its coverage and scalability. For Android apps, their byte-code is more accessible and can therefore be inspected using static analysis. The trouble is that determining the presence of triggers is notoriously hard: essentially, a trigger is just a branch condition (also called a *check* in our study), one of the most common program structures; it is hard to link such a condition to the intention for hiding suspicious activities. Current solutions rely on either carefully specified security-sensitive behaviors (e.g., permission-protected methods, read from sensitive Content Providers [51]) or well defined trigger conditions [34] to avoid false positives. A prominent example is TriggerScope [34], which utilizes a set of *narrow* conditions, as supported by symbolic execution, to identify suspicious triggers. The problem is that the condition here needs to be *precise* and therefore restrictive (e.g., comparison between a time value and a constant), which limits the approach only to known types of triggers (time, location and SMS in the paper). Likewise, the specific behaviors used for detection only work on a known set of suspicious activities covered by triggers. As a result, *none of the existing techniques are capable of finding unknown HSO*. Also, the use of heavyweight techniques (e.g., symbolic executions) renders the approaches like TriggerScope less suitable for a large-scale study.

Our approach. In this paper, we present a new technique that makes possible large-scale discovery and analysis of previously-unknown HSO within Android apps. Our approach, called HSOMINER, is built upon a set of unique features shared across different types of HSO. More specifically, an HSO tends to check only system inputs (e.g., time, SMS, keys entered by the user, etc.), rather than its hosting app's internal inputs, for triggering hidden behaviors. Further, such behaviors, performed by the hidden path of the trigger condition, are very different from the operations on the other path that serve to cover the sensitive activities when they are not triggered (e.g., sending SMS vs. simply exiting the current method). Most interestingly, except their control dependency, the trigger and the hidden behaviors tend to be unrelated: e.g., consider a time bomb that uses time to determine when to perform malicious activities (e.g., stealing private data); rarely does the time also serve as an input to the activities. Fundamentally, a trigger is meant to check whether an app is running in the target situation (right moment, location or device), which is often orthogonal to what the app intends to do in that situation (data stealing, SMS sending, etc.). This is different from a normal branch, in which the operations on either path are often connected to the condition through data flows or shared resources (e.g., if the camera is ready, take a picture through the camera).

None of these features is dependent on specific trigger conditions or sensitive behaviors, which potentially allows them to be used to find previously unknown HSO. Also importantly, they are relatively easy to extract from a program and *even though each individual of them may not be accurate enough for detection (incurring false positives), collectively they provide a more precise description of suspicious hidden activities*. In our research, we utilize lightweight program analysis techniques to recover these features from the branching structures within an Android app, and run a machine learning algorithm to identify those involving HSO. Our evaluation shows that HSOMINER achieved a precision over 98% and a recall above 94%, at a speed of 13 minutes per app over the apps with a much larger size (typically around 8.43 MB) than those studied in the prior research [34], [51]. This level of performance enabled us to conduct a measurement study on HSO at an unprecedented scale: we scanned over 338,354 Android apps (including 124,207 from Google Play and 214,147 from VirusTotal), and discovered 63,372 containing HSO; among them, 1773 involve the HSO techniques *never reported before* (Section IV).

Findings. Our measurement of HSO, the largest of its kind, brings to light the pervasiveness of hidden activities across the Android ecosystem: about 18.7% of the apps were found to involve some suspicious behaviors they attempt to cover. In addition to known triggers, such as time, location and SMS, we found that suspicious behaviors (e.g., sending SMS) are protected by monitoring various system events, including incoming Intent, new package added, screen locked etc. and by detecting the presence of human users: e.g., setting a threshold for the interval between two consecutive clicks on the screen. Even for the old tricks, such as identifying an emulator, new techniques have been employed, such as checking certain bits of /system/bin/linker to find out whether the app is running on an X86 system. Further, HSO techniques seem to evolve with Android, leveraging new functionalities added to the OS and new services it supports: an example is a PHA that uses the `device manager` to hide the behaviors of stealing user data. Further our study shows that HSO code has been disseminated through the libraries shared across technical forums/repositories such as GitHub and pudn [14]. Also new techniques proposed in the academia have been quickly picked up by PHA authors. For example, Anti-emulation techniques proposed on HITCON 2013 [10] were found to be used in real-world PHAs.

Contributions. The contributions of the paper are summarized as follows:

• *New technique*. We developed HSOMINER, a novel machine-learning based program analysis technique for automated detection of hidden sensitive operations, including those previously unknown, on a large scale. HSOMINER is built upon a set of simple yet salient program features, leveraging their collective differentiating power to identify HSO. In this way, we keep the features general, thereby allowing the technique to work on new types of triggering conditions and suspicious behaviors. Also, the high-level idea of using lightweight program features and machine learning to avoid complicated code analysis could find its application in other security domains.

• *New discoveries*. Using HSOMINER, we carried out so far the largest study on HSO. In the study, over 330K recent apps were scanned, which led to the discovery of over 60K apps with hidden behaviors. Analysis of these apps further reveals new HSO techniques and their evolving trends. This is invaluable for better understanding of the mobile HSO risks and the enhancement of our defense against this emerging threat.

Roadmap. The rest of the paper is organized as follows: Section II presents the background of our research; Sec-

tion III elaborates our design, implementation and evaluation of HSOMINER; Section IV describes our large-scale measurement study and findings; Section V surveys the related prior research; Section VI discusses the limitations of our techniques and potential future research, and Section VII concludes the paper.

II. BACKGROUND

In this section, we explicate the HSO activities studied in our research and existing HSO techniques, particularly those used in Android apps. We also present the assumptions made in our study.

HSO and Android. As mentioned earlier, we use the term "hidden sensitive operations" (HSO) to describe the branch condition (the trigger) that hides security-sensitive behaviors and the behaviors along one path of the trigger that can only be invoked when the condition is satisfied. Such HSO tricks have long been utilized by malware authors to evade dynamic analysis. Here, the trigger condition serves to determine whether the malware is running in its target environment, which is typically found through a set of heuristics: for example, looking for differences in the outputs of certain APIs, the presence of virtual machine (VM) related system files or observable performance degradation related to virtualization, etc. [18]. Further, detection of human interactions and the use of various analysis engines (e.g., FireEye Multi-Vector Virtual Execution Engine [11]) also becomes popular in desktop HSO. In the case that such behaviors are not triggered, sophisticated malware can even display a message to disguise the disruption of its service as configuration troubles [20].

Similarly, tricks for detecting emulator form the mainstay of the HSO techniques employed by Android apps. These approaches tend to exploit information leaks on QEMU and VirtualBox, including their unique system files and constant values in the results of certain API calls like `getDeviceId`, `IMEI`, `Build.FINGERPRIN`, `getLine1Number` [48]. Moreover, a mobile device today is characterized by its abundant built-in sensors and strong support for user interactions, which also opens new avenues for hiding sensitive code: for example, checking the presence of camera to find out whether an app indeed runs on a phone or the patterns of clicks on the screen to determine whether indeed a human (instead of an automatic tester) is using the device (Section IV-C).

To defeat the HSO for evading emulators and other analysis environments, techniques have been proposed to cover the traces of these environments [45]. As an example, without protection, a call to `android.os.Build.MODEl` immediately returns `google_sdk` in the Android emulator; to avoid this exposure, one can enhance the emulator by hooking such APIs and forcing them to output fake values that mimic those of real-world devices: e.g., `SM-G920F`. Alternatively, popular apps can be installed on real phones and analyzed there. A problem is that such anti-HSO measures entirely rely on the knowledge about specific HSO behaviors, and therefore tend to be less effective on unknown HSO. So discovery and understanding of new HSO behaviors becomes important to mitigate this potential security threat.

Assumptions. We consider a situation where the app developer utilizes all kinds of system events to determine when to trigger hidden sensitive operations. Note that we do not assume any specific forms of trigger conditions (e.g., comparison between a time value and a constant, as did in the prior research [34]), as long as the triggers involve system (device build info, state of a sensor, etc.) or user inputs (Section III-B); nor do we limit the hidden operations to a set of manually selected security-sensitive behaviors, as long as such operations indeed involve at least one API that once abused, can cause harm to the app user (Section III-C). Examples of such sensitive APIs are provided by the Android official security documentation [7][2]. Further, just like all prior effort on static analysis of HSO [37], [33], we do not consider the apps whose branch conditions have been deeply obfuscated. Finally, as also found in the prior research [34], legitimate apps can also exhibit some evasive behaviors. A prominent example is gaming apps, many of which stop running within the emulator. So it is important to note that this study aims at understanding such behaviors, not directly detecting PHAs, though discovery of HSO often indicates the presence of potentially-harmful behaviors.

III. FINDING HSO

Here we elaborate the design, implementation and evaluation of our techniques.

A. Overview

The design of HSOMINER focuses on the structure of a branch, which exhibits unique features when it involves hidden behaviors, regardless of the details of the operations. Such features can be observed from the *relations* between the branch's individual components, including its condition and paths, and the *relation* between the components and system events. More specifically, a trigger condition is always related to system inputs (time, location, screen touches, etc.), whereas a non-HSO branch may rely on an app's local data alone (e.g., a comparison between a loop variable with a constant). Further comparing the paths controlled by the trigger, one with sensitive operations and the other not (serving as a cover for the former), we can see a significant discrepancy between their behaviors (e.g., retrieving accurate location data on one path and displaying UI elements on the other), with the latter acting less alarmingly to avoid any suspicion about the whole branch. Also interestingly, a trigger is much less likely than a legitimate condition to share resources with the sensitive operations: as an example, for a legitimate Intent handler, after the content of an incoming Intent is checked, the follow-up operations will also happen on the Intent; in the case that the branch just serves as a disguise for an HSO, however, the operations can be completely unrelated to the Intent (data stealing, sending out SMS and others). From these three categories of relations, our approach extracts 7 features (see Section III-B) that describe the connections between a condition and system events and between the condition and its security-sensitive path, and also the behavior distance between two paths. These salient features, as demonstrated by their differentiating powers (Section III-B), are then used collectively to capture an HSO branch.

Architecture. Figure 1 illustrates the architecture of HSOMINER, which includes a pre-processor, a feature extractor and a classifier. The pre-processor unpacks an app's APK and then decompiles and converts its code into intermediate

[2]The list contains only a subset of APIs we consider sensitive (Section III-C)

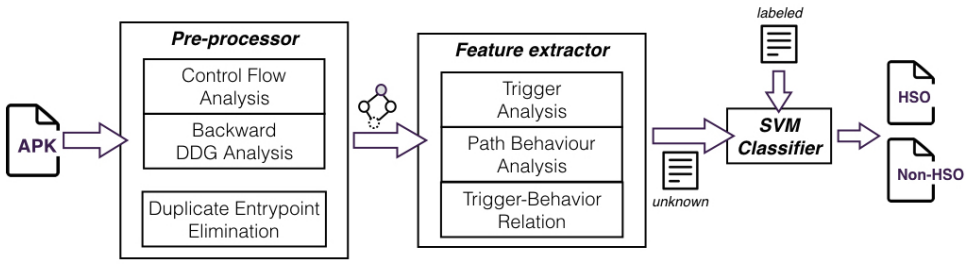

Fig. 1: Overview of components in HSOMINER.

```
1 public void hso(){
2     boolean containsAV = false;
3     List<PackageInfo> packagelist =
4                 getPackageManager().getInstalledPackages(0);
5     for(PackageInfo package:packagelist){
6         if(package.packagename.contains("com.antivirus")){
7             containAV = true;
8         }
9     }
10
11    StringBuilder params = new StringBuilder();
12    //1) retrieve common info like imei, imsi into params
13
14    if(!containsAV){
15        //2) retrieve security sensitive info into params
16        params.append("&no=").append(utils.getPhoneNumber());
17        params.append("&lat=").append(utils.getLatitude());
18    }else{
19        params.append("&no=").append("00000000000");
20        params.append("&lat=").append("0.00");
21    }
22
23    //3) upload params to remote server
24 }
```

Fig. 2: A possible implementation of HSO.

language, which is ultimately transformed into a global control-flow graph (CFG). The CFG, built on top of Soot [15], comprises a set of subgraphs according to the app's entry points. The subgraphs are further linked together with asynchronous tasks, lifecycle of Android components and inter-procedural invocations. HSOMINER automatically processes these sub-graphs, identifies potential conditional branches, and outputs them for feature extraction. To fully understand the triggers involved in conditional statements, a backward data dependency graph (DDG) is constructed for the variables appearing in the statements, while the behaviors of paths are directly identified during construction of the CFG. The pre-processor also clusters packages in apps (Section III-C), which helps to minimize the set of entry points to be analyzed.

The feature extractor takes trigger conditions and their corresponding paths as its inputs and outputs a set of features collected from them (Section III-B). These features, together with a set of confirmed HSO or Non-HSO training instances, are utilized to learn a classification model, which serve to detect other HSO apps.

Example. Here we use an example in Figure 2 to explain how HSOMINER works. The highlighted branch condition `!containsAV` in the figure checks whether any Anti-Virus scanner exists on the current device. If not, the app collects the user's phone number and precise location (which are sent out later); otherwise, it just attaches blank content to the message.

When analyzing the branch, HSOMINER traces back to the program location where the boolean variable `containsAV` is defined and finds that it has been determined by a system input returned by `getInstalledPackage`, which includes the list of all installed packages. Also looking at the difference between the two paths, our approach discovers that sensitive APIs `getPhoneNumber` and `getLatitude` are present on one path and neither show up on the other. Finally, the condition, which is related to the package list, does not have any data dependency or other resource sharing (e.g., through a common object) with the statements on the security-sensitive path. As a result, our classifier flags the branch as suspicious. Following we describe the details of the features selected and individual system components.

B. Features

As mentioned earlier, HSOMINER relies on a set of unique program features to identify HSO activities, including those characterizing the constraint on a trigger condition, behavior differences across different paths and trigger-behavior relations. Here we present such features, and demonstrate their differentiating powers using 213 VirusTotal [16] apps confirmed to contain HSO activities and 213 randomly chosen Google-Play apps considered to be legitimate. The Virustotal apps were those matching the signatures of known Android malware,e.g., Android.HeHe [5], RCSAndroid [22] and those reported by the prior studies [35], [48], and the Google-Play apps were cleared by VirusTotal and double-checked manually.

Trigger condition. The trigger of an HSO is meant to identify the *situation* for invoking hidden activities. For almost all the HSO instances discussed in the literature [5], [22], [45], [4], [2], such a situation is characterized by some system properties (e.g., OS or hardware traces of a mobile device) or environment parameters (time, locations, user inputs, etc.), which are only exposed to an app through system interfaces. As a result, an HSO condition is expected to involve, directly or indirectly, one or more API calls for interacting with the OS. For example, Android time bombs tend to directly compare the current time (returned by *java.util.Calendar*) with a constant [3], which is often directly embedded within a trigger condition. As another example, the boolean variable `containsAV`, as illustrated in Figure 2, is related to the call `getInstalledPackages`.

It is important to note that ordinary conditions unrelated to HSO may also involve system inputs. However, in most cases, they use local variables only, for example, comparing a loop variable during each iteration. Interestingly, we found that a simple binary feature on whether any system input is involved in a condition, denoted by SI (system input), gives a good indication for the presence of HSO. As shown in Table I, the F-score of SI is 0.85 (with precision at 0.812) when identifying

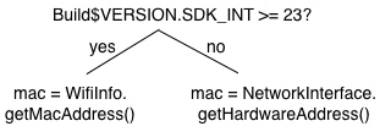

Fig. 3: Accessing MAC.

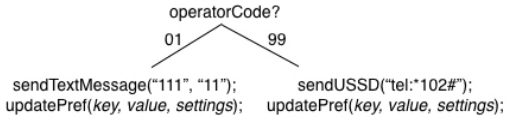

Fig. 4: Checking account balances.

HSO instances over our ground truth dataset. The details for extracting the feature from the apps' bytecode is provided in Section III-C.

Behavior differences. In the case that an HSO condition is not satisfied, an "exposed" path, without hidden activities, would be taken. Intuitively, the app behaviors on this path and those on the hidden path should be considerably different: as shown in Figure 2, the hidden path involves a set of sensitive API calls (`getPhoneNumber` and `getLatitude`), while the exposed path has none. Apparently in this example, a Jaccard distance: $D = 1 - \frac{O_l \cap O_r}{O_l \cup O_r}$, where O_l and O_r are the sets of sensitive operations on two paths of a branch statement, describes different behaviors between hidden and exposed paths. Such behaviors are specified by the APIs that are considered to have security and privacy implications, as provided by the Android official documentation [7] and a few other lists (PScout [26] and DroidSIFT [52]) and other system properties and Android settings. Note that the API set we choose is more general than that adopted in prior research. Also importantly, we place less restrictions on the relations (e.g., the order between a data retrieval API and sinks) of different types of APIs, which makes it more likely to capture unknown HSO activities.

In practice, however, the situation could be more complicated. Figure 3 illustrates a legitimate branch, whose paths involve different sensitive APIs, even though they all perform similar operations, that is, querying the MAC address of the device. Interestingly, the cross path behaviors look not so different if we focus on different *groups* of similar APIs and other system properties: e.g., `NetworkInterface` and `WifiInfo` can all be used to collect network information; other examples include system or Android properties such as `'https.proxyHost'` and `'android_id'`, whose content can also be acquired through the APIs like `Proxy.getDefaultHost()` and `TelehonyManager.getDeviceId()`. To address this issue, we group APIs or system keys based upon the similarity of their functionalities. Our distance D across different paths is calculated over the group identities of individual APIs or system properties, which we call AD (Activity distance). Table III shows the AD of the branch illustrated in the figure.

Further, even when both paths involve different groups of APIs, they might still perform similar operations. Figure 4 shows another example: API groups on both paths are quite different. The path under condition "*operatorCode* equals 01" includes SMS APIs, whereas the other path does not. This case could be flagged as HSO if AD is used alone for detection. However, it turns out to be completely legitimate: the branch is actually within a utility app for network providers in Russia, which gives options to their customers to check her account balance through SMS or Unstructured Supplementary Service Data (USSD). Identifying such a subtle relation across paths is clearly nontrivial. However, in our research, we observed that the presence of shared variables and constants across the paths often indicates the existence of such a relation. In the above example, content like the key of shared preference shows up across the paths, for the purpose of updating the value of account balance. Intuitively a legitimate branch is more likely to have data dependency across its paths than an HSO branch. Based upon this observation, we utilize another feature in our research, called DD (data distance), to complement AD. Specifically, $DD = 1 - \frac{1}{2}(\frac{V_l \cap V_r}{V_l \cup V_r} + \frac{F_l \cap F_r}{F_l \cap F_r})$, where V_l and V_r are the sets of variables (excluding those locally defined on current path), and F_l and F_r are the sets of referenced class fields on two paths of a branch statement. Table I shows the F-scores for both AD and DD over our ground-truth dataset.

Condition-path relation. Further, we observe that for a branch involving hidden behaviors, the link between its condition and the operations along its paths is often remarkably thin. Indeed the former sits right on the control flow of the latter. Except for this relation, however, a trigger typically does not propagate any data flow to its paths or share other resources with the operations there. Fundamentally, while an HSO condition is meant to determine the right situation for running its hidden code, the code itself is not meant to process any inputs provided by the condition. For example, when an HSO app reads from a register to find out whether it is running inside an emulator, the code for stealing private user data, as invoked by the condition, is not supposed to take the OS fingerprints as its input. To leverage this property, we come up with two unique features, as elaborated below.

Specifically, we attempt to find out whether any operation or variable on paths has data dependency with any variable within the condition. This is done through a define-use analysis performed on every single variable on each path (see Section III-C for details). Also analyzed in our research are implicit condition-path relations: we recover system-related object instances from variables on each path, in order to check whether these resource objects (e.g., `LocationManager`) are also related to the variables, APIs or system keys (e.g., '`location_providers_allowed`') within the condition. For example, an implicit relation can be a check on the key `location_providers_allowed` within the condition (whether location information can be accessed), and an access to `LocationManager` in the path (use of the location data). Some of the relations can be found through program analysis, in the case of shared object instances (e.g., check the state of default SIM card by `getSimState()` with a `TelephonyManager` instance and get phone number by `getLine1Number()` of the same instance in the path). In other cases, we need domain knowledge to determine whether a key relates to an API or whether two APIs are related. In our research, we collected such relations from legitimate apps, using the most pervasive API-API or key-API pairs observed from their branches. The details are presented in Section III-C.

Features describing such trigger-behavior relations include DF (data dependency), which is the ratio of the variables on

a path connected to the condition through data flows, and IR (implicit relation), which is the number of variables, keys and APIs implicitly related to the condition. The F-scores for both features, as measured on our ground-truth set, are illustrated in Table I.

TABLE I: F-score of features

	SI	AD	DD	DF^2	IR^2
HSO	0.85	0.579	0.67	0.766	0.774

[1] F-score is calculated based on classification with each single feature.
[2] Each of DF and IR is related with two paths.

C. Detection

All the features are chosen based not only upon their differentiating powers but also upon their relative convenience of collection from an app's code. Following we explicate how to run lightweight, localized program analyses to recover these features and how to use them to detect HSO activities.

Pre-processing. Given an app, the preprocessor first runs Apktool [6] to unpack it and extracts bytecode from its apk file. Then, our analysis tool, built on top of Soot [15], looks for entry points of different packages, including lifecycle callbacks (e.g., `onCreate`), user interactive callbacks (e.g., `onClick`) and broadcast receiver callback (i.e., `onReceive`), as prior studies do [41], [52]. From each of the entry points, HSOMINER explore all reachable code to create a control-flow graph (CFG), which we call *global subgraph with regard to the entry* or simply global subgraph. To improve the performance of this analysis, we use a technique to automatically identify the packages analyzed before [49], based upon a set of features (e.g., call frequency of a set of Android APIs, total number of API calls, and other statistical features in meta-data) that fingerprint them. For those packages, which almost always are shared libraries, our approach skips their entry points, without building their global subgraphs.

A global subgraph models the Android Activity/Service life-cycles and handles inter-component communications (ICC) and asynchronous tasks: ICC is analyzed using Epicc, an open-source ICC mapping system [44], in our implementation and asynchronous classes within the Android framework are described using the calling convention table provided by DroidSIFT [52]. Our approach also supports a context-sensitive, flow-sensitive and inter-procedural backward data-flow analysis on selected variables to construct their data-dependency graphs (DDG). This technique serves the purpose of discovering the origins of selected variables within the conditions leading to sensitive APIs (Section III-B), which is important to the missions such as establishment of the link between a branch condition and system inputs.

Over each subgraph, HSOMINER first locates all basic blocks (a block of statements not involving any branch) containing sensitive activities, which are represented by a broad set of APIs as defined by the Android documentation [7]. Note that unlike other prior work with description of an app's behaviors, here we intend to adopt a much more general list, with almost all prominent APIs onboard except for those of fundamental classes in Java/Android (e.g., `java.lang.*`), utility classes (e.g., `android.util.Log`) and other classes related to data format (e.g., `JSONObject`), since they can hardly be linked to any damaging operations. Once such a *sensitive block* is discovered, the preprocessor goes backward over the CFG to find all the branch statements (e.g., if-then, switch-case, etc.) predicate the block. Once found, the scope of such a branch is determined on the subgraph through exploring different paths of the branch until the program location where the paths converge. Further, a backward data-flow analysis is performed for each branch, generating the DDG for the variables its condition carries. The DDG and the scope of the branch forms a new graph, called *condition-path graph* (CPG), which is used for the follow-up feature extraction. An example CPG is shown in Figure 5. As we can see from the figure, like the subgraph, the CPG is inter-procedural, fully preserving the information for a precise branch analysis.

Feature extraction. Over the CPG, the feature extractor automatically identifies the branch features for HSO finding, as elaborated below.

• *SI*. As mentioned in Section III-B, SI is used to determine whether a branch condition is related to system inputs, which is more pervasive among trigger conditions than innocent conditional branches (those not involving any HSO). To extract this feature from a given branch, HSOMINER inspects all variables in the branch condition to find out whether they are affected (via accessibility analysis on backward DDG) by any APIs that receive inputs either from system resources (such as geo-locations, IMEI, etc.) or from user interfaces. Examples of such APIs include `<Date getTime()>`, `<Locale getCountry()>` and `<Settings$Secure getInt(···)>`. In our research, the list of such APIs are obtained using SuSi [25], which automatically classifies Android API into sources and sinks. Our implementation includes the sources identified by SuSi as system inputs, with 18,076 APIs for Android 4.2.[3] Note that system inputs not always come from APIs. Actually an app can directly read from system properties such as `android.os.Build` and receive data from Intents or other system events. Therefore, we also add these properties and events to the list (so a condition related to these properties and events is considered to receive system inputs).

Given the list of system inputs, our feature extractor automatically traverses the CPG of a branch to look for the presence of the inputs on the DDGs of the branch's conditional variables. The SI for the branch linked to any of such inputs is set to one. Otherwise, it becomes zero. It is important to note that this constraint on the condition (relation with system inputs) is much looser than that of Triggerscope [34], which utilizes specific constraints like whether the current time is after '2016-12-22'. In this way, we expect the feature, together with others, can help find unknown types of HSO.

• *AD and DD*. AD and DD are used to measure the behavior difference between two paths of the same branch. As mentioned earlier, our study shows that paths associated with a true HSO tend to act differently than those attached to a normal branch (Section III-B) due to the fact that the hidden activities (such as accessing location data, sending SMS and modifying system settings) are clearly dissimilar to, and often completely independent of those exposed. To extract AD, our approach simply goes through the paths attached to a branch on its CPG to identify all the sensitive APIs and the operations involving

[3] Note that the set of Android APIs is relatively stable.

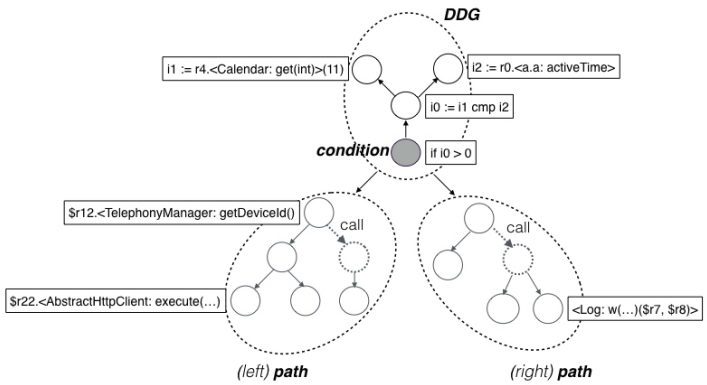

Fig. 5: The condition-path graph (CPG) of a branch.

sensitive system properties (Section III-B) along the paths, maps them to their corresponding behavior groups before calculating the Jaccard distance between these two sets of behavior groups (on two paths respectively).

To discover the data dependency between paths, a generic approach is to backtrack the data flow for each variable on a path to find out whether indeed it is related to some variable(s) on the other path: that is, they are all traced back to the same assignment or definition statement or all referring to the same object. This analysis is clearly heavyweight and does not seem to be necessary: for all the legitimate branches with such relations found in our study, we observed that related variables are all associated with a common statement (assignment or definition) within the CPG of the branch or an object passed to the method hosting the branch through an inter-procedural call. Therefore, we can establish the data dependecies between the variables across paths by simply analyzing their data flows over the CPG of the branch and the current method. This enables us to conveniently calculate DD, as described in Section III-B.

• *DF and IR*. Like other features, both DF and IR, which describe the explicit data-flow and implicit relations between a branch condition and its path, are also extracted from a branch's CPG. Specifically, DF measures how variables on paths are affected by the condition through data flows. This feature is extracted from a branch through a define-use analysis: that is, whether any variable used on a path has actually been defined at the condition. Here "define" means that the variable is the target of the operation performed by either a definition or an assignment statement (that is, where the variable gets a new value). Given a set of variables on a path, $v_1, \cdots, v_n$, our approach simply inspects each variable on the CPG to find out whether it has been defined in the condition. DF is calculated as $\frac{k}{n}$, with k being the number of those related to the condition.

Also as mentioned in Section III-B, a branch condition can be linked to a path implicitly, through shared resources, particularly, common object instances (e.g., `java.net.Socket`, whose `isConnected()` property is checked in the condition and `getInputStream()` method is used in the path), related APIs (e.g., `<Environment getExternalStorageState()>` and `<OutputStream write(···)>`), related system key-API pairs (e.g. "`ACCESS_FINE_LOCATION`" and `<LocationManager requestLocationUpdates(···)>`). In our research, we systematically collected a set of such related resources through inspecting 11,463 branches within 1500 popular legitimate apps and picked up top 50 pervasive API-API and key-API pairs from the branches (see Table II for the examples). During its operation, HSOMINER automatically inspect the CPG of a branch, looking for the variables associated with the same resource object and related APIs or key-API pairs between the condition and the paths. A path's IR is then calculated as the number of its related APIs, variables and keys.

It is important to note that all these features, SI, AD, DD, DF and IR, are identified through a *localized* analysis, which focuses on a branch's CPG. In this way, we make feature extraction lightweight, which is critical for achieving scalability for an HSO analysis.

Classification. As illustrated in Table I, those lightweight features do contribute to the detection of HSO activities *individually*. However, such a contribution is limited, coming with significant false positives and negatives: the precisions of these features range from 56.4% to 84.6% and their coverage from 51.4% to 84.3%. Clearly none of them is perfect and none of them can work alone. This is mainly caused by the generality preserved by these features, which is important for finding unknown HSO: for example, instead of precisely specifying how a trigger condition should look like (e.g., a time variable compared with a constant [34]), we only expect the condition to involve system inputs. The key idea of HSOMINER is to use these imperfect yet lightweight features *collectively* to enhance their effectiveness in finding hidden activities. For this purpose, we resort to machine learning techniques, using a classification model to predict whether a branch is indeed an HSO. Our study shows that this approach significantly elevates the quality of HSO detection, raising precision to 98% and coverage to 94.4% (Section III-D).

Specifically, we construct a feature vector $v = (SI, AD, DD, DF_l, DF_r, IR_l, IR_r)$, where l and r represent the left and right paths of a branch, respectively. For simplicity, the vector here only describes the branch with two paths. When it comes to those involving multiple paths, we can use AD and DD to represent the largest distance across all pairs of paths, and add DF and IR for each path to v. Table III shows a few examples of the feature vectors, including a time bomb that triggers hidden behaviors on a certain day (the trigger continuously checks the current date using a calendar), a benign instance that checks `INTERNET` permission before initiating network procedure, and two other instances (accessing MAC and checking account balance) described in Section III-B.

TABLE II: Examples of APIs and key/API pairs used in IR

Item in condition	Item in path
⟨android.location.LocationManager: isProviderEnabled(· · ·)⟩	⟨android.location.LocationManager: requestLocationUpdates(· · ·)⟩
⟨android.webkit.WebViewClient: ⟨init⟩()⟩	⟨android.webkit.WebView: loadUrl(· · ·)⟩
⟨android.net.NetworkInfo: getState()⟩	⟨android.net.ConnectivityManager: getNetworkInfo(· · ·)⟩
⟨android.os.Environment: getExternalStorageState()⟩	⟨java.io.File: mkdir()⟩
'location_providers_allowed'	⟨android.location.LocationManager: getLastKnownLocation(· · ·)⟩
'PACKAGE_CHANGED'	⟨android.content.pm.PackageManager: java.util.List getInstalledPackages(· · ·)⟩
'GET_ACCOUNTS'	⟨android.accounts.AccountManager: getAccountsByType(· · ·)⟩

Using the vector, a classification model is then trained over a ground-truth dataset. The dataset in our research includes confirmed HSO apps like Android.HeHe [5] and legitimate ones (Section III-D). The machine learning algorithm implemented in HSOMINER is support vector machine (SVM), which was chosen in our research due to its over-fitting resistant feature. Like other learning-based approach, we trained and then evaluated the SVM model using cross-validation, before running it against 330K real-world apps. Section III-D provides the details of this study.

TABLE III: Feature vector for HSOMINER

	SI	AD	DD	DF_l	IR_l	DF_r	IR_r
time bomb (Calendar)	True	1.0	1.0	0.0	0	0.0	0
permission check	False	1.0	1.0	0.31	2	0.0	1
access MAC	True	0.0	1.0	0.0	0	0.0	0
account balance	True	1.0	0.5	0.0	1	0.0	0

D. Evaluation

In this section, we report our evaluation of HSOMINER, in terms of its capability to identify HSO activities and its performance.

Settings. Our implementation was evaluated over a labeled bad set, a labeled good set and an unknown set. To avoid overfitting caused by unbalanced data, we made the good set and bad set of same size. The bad set contains confirmed 213 PHAs, each with one HSO branch. The good set involves 213 Google-Play apps that have never been flagged by any Anti-Virus (AV) service hosted on VirusTotal [16]. To overcome the potential overfitting caused by small training set and make our classifier more generic, we did two rounds of retraining by using manually confirmed classified samples as training samples. The unknown set has 338,354 apps, with 124,207 of them collected from Google Play, and the rest 214,147 randomly downloaded from VirusTotal. The details of our data sources are described in Table IV. Note here that although a large number of apps were flagged by at least one VirusTotal scanner, many of them can actually be innocent, due to the false positives introduced by these 57 scanners, which are particularly sensitive to the apps with rich functionalities such as collection of location information. Also many apps hosted by VirusTotal are also legitimate.

TABLE IV: Details of the unknown set (with the number of the apps flagged by at least one VirusTotal scanner)

	Not flagged	flagged	Total
Google Play	52,319	71,888	124,207
VirusTotal	47,937	166,210	214,147
Total	100,256	238,098	338,354

For the performance evaluation, we randomly selected 3000 popular apps from Google Play, with their sizes ranging from 36KB to 90MB. Further, 35 relatively small apps used by the prior research [34] were also tested, to give us some idea about how our system performs compared with the prior one. These experiments were conducted on a Dell desktop with 3.3GHz Intel Core i5 processor and 16GB RAM. The timeout of the analysis was set to 60 minutes, in line with the prior study [51].

Effectiveness. To understand the effectiveness of HSOMINER, we first trained the classifier over the labeled sets, using a polynomial-kernel based SVM. Our 4-fold cross validation shows that HSOMINER achieved a precision of 98% for HSO detection and a recall of 94.4%. The detailed results are presented in Table V.

TABLE V: Detailed accuracy of SVM classifier

	Precision	Recall	F-score
HSO	0.98	0.944	0.962
Non-HSO	0.946	0.981	0.963
Weighted Avg.	0.963	0.962	0.962

We then ran the trained model to predict unknown instances across all 338,354 apps. Altogether, 63,372 apps with 70,660 branches were flagged as suspicious HSO. Among them, we randomly sampled 125 apps (with one flagged branch each) for a manual validation. All except two instances were considered to be true positives, with a precision of 98.4%, in line with what we found from the labeled set. In the two likely false positives, the apps first attempt to retrieve Device ID and if unsuccessful, try to read the current device's MAC address. We did not see concrete evidence that the apps intend to hide such activities through some narrow trigger conditions. Nor did we observe that sensitive user data was leaked. As a result, we did not count them as true positives. In all other cases, clearly we found HSO behaviors such as triggering hidden activities based upon time, UI inputs, environment types (emulator or not), etc. Table VI presents the types of trigger conditions discovered from these instances. Top on the list are "Time", "Device Info" and "System Event". Our approach also revealed some less known triggers such as "progress bar", "account manager", as well as unique hidden behaviors, like `broadcast relay` and `system version inference`. Details of our findings from the unknown set are presented in Section IV.

Performance. To understand the performance of HSOMINER, we ran our prototype on 3000 randomly-selected apps from Google Play, with an average size of 8.43 MB. On average, HSOMINER spent 765.3s on each app (except 8.4% apps that were partially analyzed before timing out). Further, we attempted to compare our approach with TriggerScope [34], a logic-bomb detector. Without access to its code, the only

TABLE VI: Trigger condition of detected HSO

	Cases	Count
System Event	screen on, boot completed	19
Time	calendar, current time, date	43
Device Info	device id, build info	41
Device Settings	system settings	6
Location/Environment	latitude, country, network operator	8
UI	key repeat count, progress bar	2
Miscellaneous	uid, account manager	4

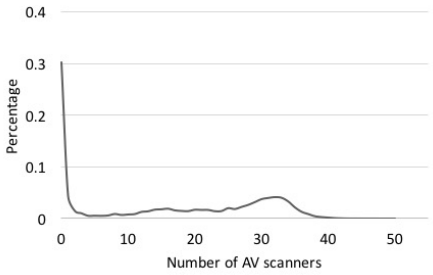

Fig. 6: Distribution of apps by number of AV scanners.

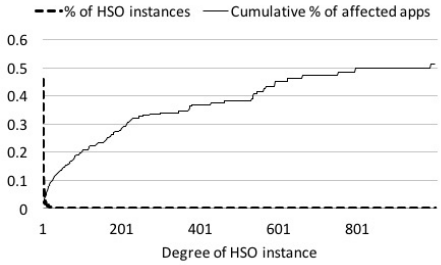

Fig. 7: How HSO instances are distributed across apps. Here "Degree of HSO instance" means the number of apps affected by one HSO instance. As we can see, a large proportion of HSO instances are used only once (see solid line); However, many apps include shared libraries with HSO activities (indicated by dotted line).

thing we could do is to test HSOMINER on the apps also used to evaluate TriggerScope. Altogether, we got 35 of such apps, whose sizes range from 14KB to 461KB. All other apps apparently are too obsolete to be found online. With the analysis timeout set to 60 minutes, HSOMINER successfully processed each of these apps within 42 seconds on average, with 255 seconds in the worst case and less than 60 seconds for 79.2% of the apps. Note that the prior research reports an average performance of 219.21 seconds per app, around 5.2 times slower compared with our approach, though this comparison may not be entirely fair, due to the lack of the code of Triggerscope and uncertainty about its evaluation environment. At the very least, the study does show that HSOMINER indeed works efficiently.

IV. MEASUREMENT AND DISCOVERIES

The efficiency of HSOMINER enables us to study HSO on a scale that has never been achieved before. Through a systematic analysis of over 330K apps, including popular apps from Google Play, our research reveals the pervasiveness of the HSO activities, which were found in 18.7% of the apps, and the diversity of triggers (based on patterns of UI events, states of system servers, etc.) and hidden activities (dynamic code loading, video recording, etc.). Also importantly, HSOMINER discovered new HSO activities never reported before. Examples include user data collection happens only after 10 ms of video playing and sensitive activities invoked only when a specific area on the screen is clicked upon. Further our study sheds light on how HSO activities evolve and HSO techniques are disseminated. Following we elaborate these findings.

A. Landscape

Our measurement study was conducted on a total of 338,354 Android apps, of which 214,147 were randomly selected from those cached by VirusTotal and 124,207 from Google Play. Duplicate apps were removed according to their SHA256. Distribution of the apps' release time shows that: although there are apps published years ago, most (about 70.9%) of them are new and submitted after 2015. Further we illustrate how those cached by VirusTotal were selected in Figure 6. The apps downloaded from there were randomly selected across the number of Anti-Virus (AV) scanners that flag them (with about a third of them considered to be legitimate). On average, the size of these apps is 5.98 MB and among all 2,245,235 packages extracted from them, 94.2% were successfully processed by HSOMINER without causing any time-out.

Pervasiveness. From the 338,354 apps, HSOMINER reported that 63,372 of them (18.7%) contain HSO branches. For most of these apps, however, their hidden behaviors come from shared libraries. Altogether, we found 3,491 unique HSO instances from these apps. The distribution of these instances across all flagged apps is illustrated in Figure 7. Specifically, 60.9% of these instances are present in no more than two apps (thus apparently unrelated to libraries), while remaining 39.1% are inside the libraries extensively shared by tens to thousands of apps. Also note that 12.6% of VirusTotal apps contain such non-library HSO branches while only 8.0% of Google Play apps contain this kind of branches. As an example, a time bomb within the library com.baidu.kirin was shared by 9,710 apps; the HSO involves accessing device information and initiating a network updating procedure only during a time interval specified by the constant `kirin_open_period`.

Further we show in Figure 8 the distribution of these HSO apps across different countries (recovered from the "C(country)" attribute of developer certificate). As we can see from the figure, which includes the 15 countries hosting most HSO apps, Russia, Israel and China are top on the list. Also interestingly, apparently there is a correlation between the number of HSO instances in a country and the Cost Per Install (CPI) there. Using the CPI data provided by AppBrain [8], we found that the higher the CPI, less likely an app is involved in HSO activities.

HSO and PHA. Also we found that HSO indeed relates to potentially harmful apps. Figure 9 shows a comparison between the ratios of the non-HSO apps (those not reported by HSOMINER) flagged by VirusTotal and the HSO apps. Our study shows that 69.71% of the former were detected by at least one AV scanner while 93.08% of the latter triggered alarm. The gap between these two sets of apps becomes even wider when we set the threshold of alarm to at least 9 scanners. In this case, 59.01% of the non-HSO apps were reported and 88.22% of the HSO ones found to be PHAs. This finding shows that HSO can serve as an indicator for detecting PHAs.

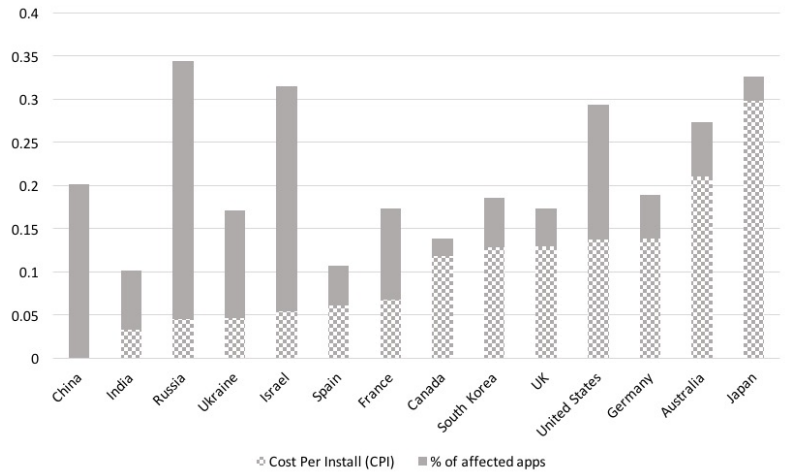

Fig. 8: Geographic distribution of HSO apps. Note here CPI is normalized and data for China is not available.

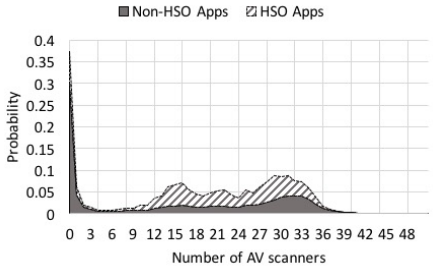

Fig. 9: VirusTotal result of Non-HSO and HSO apps

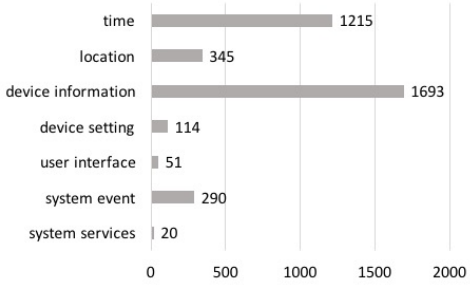

Fig. 10: Categories of HSO trigger conditions.

Triggers. We summarize the HSO trigger conditions observed from detected apps into 7 categories, as illustrated in Figure 10, including time, location, device information, device setting, user interface, system event and system services. Specifically, sensitive behaviors are invoked at a certain time, within a certain longitude and latitude range, or when the device is served by a certain network operator. Also, device specific information is found to be extensively used to detect emulators: for example, the presence of the device ID "9774d56d682e549c" may indicate that the app is being analyzed by Bouncer [43]. Further we observed that the setting of a mobile device is used to monitor the state of the device and determine the situation for running sensitive code. Examples include packers that stop decrypting executables or loading hidden code from resource files if `isDebuggerConnected` is True, and PHAs that install/uninstall apps silently when `adb_enabled` is set.

In addition to the aforementioned triggers, which have more or less been mentioned by the prior research [34], [51], [30]. Our approach also identified a set of conditions never reported before. Particularly, UI widgets were found to guard sensitive operations in clever ways. System events like reboot or date change are also leveraged to activate hidden behaviors. Even Android system services are used to trigger HSO operations: for example, when the `telegram` account becomes available in `AccountManager`, a PHA steals device owner's privacy like phone number, network operator and locale. Of particular interest is the finding that multiple trigger conditions are combined together to cover HSO activities. As an example, the package com.feicong sends SMS in background only when it receives a SMS on a certain day.

Among all the trigger conditions discovered, more conventional ones like time, location, device information dominate the pack (with nearly 3,253 out of 3,491 instances). The remaining 236 instances, however, demonstrate that sensitive activities could be hidden and invoked in more surprising ways. The details of these cases are presented in Section IV-B.

Hidden behaviors. Further we looked into the HSO behaviors hidden by the triggers. Table VII presents the 10 categories of activities discovered in our study. As we see from the table, network operations are among the most prevalent ones, showing up in nearly 70% of all the HSO instances studied in our research. Other common activities include accessing device information (e.g., phone number) (15.27%), sending SMS (12.32%) and reading location data (7.02%), which are all considered security-sensitive. In addition, HSOMINER identified a dozen of apps that dynamically load code, modify system settings, record audio/video, take pictures and access user accounts once triggered. These behaviors, obviously, are potentially harmful to the device users.

B. *Understanding HSO*

In this section, we focus on the most interesting findings made in our research, including new types of HSO, evolution of such hidden behaviors and the channels that propagate HSO techniques across app developers.

TABLE VII: Hidden behaviors of HSO.

behavior category	number of related HSO	percentage
get network state	1708	48.93%
access network	709	20.31%
access device info	533	15.27%
SMS	430	12.32%
read location data	245	7.01%
dynamic loading	10	0.29%
modify system setting	22	0.63%
record audio/video	5	0.14%
access accounts	12	0.34%
take pictures	17	0.49%

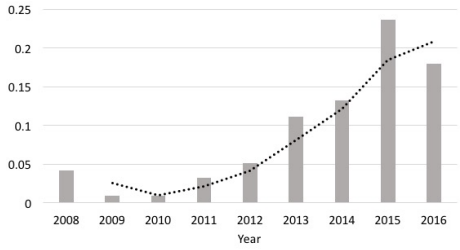

Fig. 11: Evolution of HSO.

New HSO. As mentioned in Section IV-A, not only has HSOMINER brought to light the pervasiveness of known HSO triggers, like time and location, but it also revealed several types of less known triggering conditions, including UIs, system events and system services. Specifically, we found 51 apps whose sensitive behaviors can only be invoked by some unique UI events. Some of these HSO are surprisingly complicated, leveraging the states of various UI widgets. For instance, the hidden behaviors of the package com.jackeey cannot be triggered by simply clicking on its widget. Instead, the HSO condition needs to be satisfied by the right distance and velocity of a fling event: network operations are activated only when the velocity of the wipe on the screen goes above 20.0 in pixels. As another example, the package com.FREE_APPS_237 does not exhibit any sensitive behaviors before a predefined area of a specific view is clicked. All such HSO can easily evade a dynamic analysis, even the one using the state-of-the-art UI automation tools [13], [42]. Also found in our study are the combinations of multiple more conventional triggers. For example, the HSO within jp.co.benesse.maitama can only be activated by clicking a view before a given date. Other intriguing cases include hiding behaviours in an app's progress bars or video views, which we elaborate in Section IV-C.

We also discovered that Android system services are utilized in HSO activities. Such system services (also known as system servers) include a large number of interfaces for device users to access and manage system configurations. For example, `AccountManager` serves as a centralized registry for managing the user's accounts. In our research, HSOMINER detected 20 apps that leverage these services to cover their sensitive operations. As a prominent example, an app with a package com.nrs first checks whether `org.telegram.account` exists in `AccountManager` and if so, it moves on to disable the Wi-Fi state of the current device and updating user's information (like IMEI, mobile country code) to a server though mobile data connection. Another example involves `DeviceManager`: when the service is not activated for the current user, the app package com.example.comandroid does not exhibit any suspicious behaviors (just displaying its main activity); however, once the service is launched, the app immediately hides itself by disabling its main activity and starts a service to steal incoming SMS messages in background.

Evolution. Our study also reveals the trend of HSO activities. Specifically, we collected meta-data from each app's APK files, including its `ZipModifyDate` (when the file has been changed) and certificate information, and analyzed the popularity of such techniques over time. Figure 11 shows the evolution of HSO apps from 2008 to 2016, based upon the timestamps of their hosting apps. We found that the percentage of the apps including HSO goes up most of the time. The finding indicates that HSO is gaining popularity in recent years and becomes increasingly prominent in countering program analysis.

Further, we identified variants of packages within apps based upon their common package names. From these packages, we found that 355 of them actually include HSO versions. For example, the package net.daum.adam within an app stamped 2014-05-13 has not been found to have any HSO behaviors, while the variant of the same package within a different app with a time stamp of 2015-06-26 uses multiple factors (e.g., device ID, Build model, build platform) to determine whether it is running inside an emulator, and when it is not, the package uploads user information and request an advertisement from the server.

Propagation. Our study also provides new clues about how HSO techniques are spread. We found that HSO instances are apparently disseminated through online forums. As an example, the package com.feicong, which was designed to hide its behaviors from 360 Security [1], was posted online (i.e., pudn) and later showed up within four apps in our samples. Interesting, these apps were first scanned by VirusTotal in June, 2015 though it was released as early as October, 2012. Another observation is that some new HSO techniques proposed in academia are quickly adopted by app developers. For example, some anti-emulator/taint/monkey techniques published on HITCON 2013 [10] were discovered in a library com.uc108, which was further integrated into 6 apps in our samples, all in year 2014.

C. Case Study

Here we elaborate a few real-world HSO cases found in our study.

Video trigger. Figure 12 shows an app discovered in our study that utilizes `VideoView` in its HSO trigger condition. Specifically, the app monitors the status of video playing through `getCurrentPosition`: if the video has been played for 100 ms, it starts collecting device ID and location data. While 100 ms is a very short time frame, the trick turns out to be quite effective against UI exploration tools like `monkeyrunner`, since such tools typically do not wait until the video is fully loaded to generate another user event. Our study found that similar techniques are used in 27 HSO instances in our dataset.

Trapdoor on view. The UI of a normal app is designed in a way that specific functionality is associated with a single widget/subview. However, we observed that some HSO apps hide their operations behind a specific region on a view. For

```
public void a(h arg1){
    int v6 = 100;

    //arg1.d is a VideoView
    if(arg1.d.getCurrentPosition() > v6){
        //leak sensitive info to server
        new a.b.d(this, this.b().toString()).start()
    }
}

private JSONObject b(){
    ...
    JSONObject v1;
    //collect as much sensitive info as possible
    v1.put("android_id", this.getId());
    ...
    v1.put('latitude', v2.getLatitude());
    v1.put('longitude', v2.getLongitude());
    v1.put('accuracy', (double)v2.getAccuracy());
    ...
    return v1;
}
```

Fig. 12: Video trigger.

```
public void a(MotionEvent me){
    ...
    Rect rect1 = new Rect(me.getX(), me.getY(), 1, 1);
    int width = this.display.getWidth();
    int height = this.display.getHeight();
    Rect rect2 = new Rect(0, height>>1, width>>1, height>>1+50)

    //check if certain area is clicked
    if(this.isHit(rect1, rect2)){
        //send SMS in background
        this.sendsms(...)
    }
}
```

Fig. 13: Trapdoor on view.

the app shown in Figure 13, each time when its view is clicked upon, the app checks whether the coordinates of the touch lie within a rectangle at the middle of the view with a height of 50 pixels and the width half of the whole view; only when this "trapdoor" is hit, will the app invoke its hidden activities, including sending SMS. Although it is not entirely clear whether these apps are actually malicious, this trick indeed makes it harder for the UI automation tools to trigger sensitive HSO operations.

Click interval. Another UI-based trigger technique found in our study is for detecting the presence of a human user. The approach leverages the observation that these automatic tools are designed to aggressively explore user interfaces, while a human always has a delay between her consecutive interactions with the app. Therefore, the app (com.unjiaoyou.mm) monitors the time interval between two clicks, as illustrated in Figure 14, under the function `unFastDoubleClick()`: it starts collecting user information (phone number, serial number of SIM card and IMSI) when the interval is greater than 0.5 seconds and simply exits `OnClickListener` otherwise.

Platform-specific attribute. Most known HSO techniques for detecting emulators rely on device ID or Build information. However, HSOMINER found a new approach that differentiates mobile and desktop systems from an executable's platform-specific content. Specifically, an app playing this trick reads from `/system/bin/linker` to inspect its `e_machine` field (which is 19th and 20th byte) in the ELF format. This field is 0 for ARM and 3 for X86, which discloses whether the system is running on a real mobile device or in an emulator. Also discovered in the app is a class that reads memory map to

```
private static boolean unFastDoubleClick(){
    long l1 = System.currentTimeMillis();
    long l2 = l1-a.e;
    if ((0L < l2) && (l2 < 500L)) {
        return false
    }
    a.e = l1;
    return true;
}

public final void onClick(View paramView){
    if(a.unFastDoubleClick()){
        //collect user information
    }
}
```

Fig. 14: Click interval.

determine the existence of `/system/bin/linker64`. These conditions (whether running on emulator or current system is 64bit) serve as a trigger for hidden behaviors, which includes unpacking and dynamically loading hidden code.

Broadcast relay. In addition to the novel trigger conditions, we also observed some interesting hidden behaviors that provide an additional layer of protection for suspicious activities. Specifically, we found that an app continuously monitors phone state change by registering a `BroadcastReceiver receiver1`. Interestingly, this receiver further defines another `BroadcastReceiver receiver2`, loads its code from the app's resource file and invokes it through reflection. Also, `receiver1` did not cause any alarm on VirusTotal while `receiver2` was flagged. This approach could make the hidden activities even more difficult to detect.

Version inference. Our study also shows that the behaviors under cover could be subtler than they appear to be. As a prominent example, we discovered an app that once its hidden path is triggered, invokes `setMobileDataEnabled`, compares the mobile state (acquired using `getMobileDataEnabled`) before and after the invocation, and sends the comparison result to its server. A close look at the app's code reveals that this unusual operation actually serves the purpose of *inferring* the version of the current OS: the API `setMobileDataEnabled` is available before Android L but no longer supported afterwards; this can be revealed from whether the attempt to call that API can go through. Indeed, we found that the app puts the inferred version ID in the message it sends out. Apparently, it does not want to directly call `Build.Version` to get the version number and instead uses the inference to hide its intention.

V. RELATED WORK

Trigger conditions. Hiding sensitive behaviors with various trigger conditions has been studied for a decade, starting from binary executables [29], [27], [40], [39], [28] and more recently moving onto Android apps. Most prior work is on detecting virtual machines (or the emulator for Android) and evading dynamic analysis. As a prominent example, RCSAndroid [20], a monitoring suite of HackingTeam, detects emulator by static properties like '`ro.kernel.qumu`'. In another example [45], the presence of sensors (based upon the features such as usage of virtual program counter and cache) are identified to differentiate a real mobile device from an emulator. Also leveraged as a trigger condition is the performance difference between platforms [48]. Such heuristics could even be discovered automatically, as demonstrated by prior work [35].

In addition to VM or emulator detection, other trigger conditions leverage time, location and SMS to identify the right situation on a mobile device for running sensitive code [3], [34]. A more recent study [31] expands the scope of HSO triggers by proposing to distinguish a human user from an automated UI exploration tool, so that security-sensitive operations only happen in the presence of humans.

A problem for these prior studies is that they work on a relatively small set of apps, a few thousand typically. As a result, little is known about what are happening in the wild. In our research, we performed a measurement study over more than 330K real-world apps and discovered a large number of HSO instances, including those never reported before, widening the eyes for the research on the subject. As an example, the UI-based approach, as proposed recently [31], turns out to already be in the wild, together with other complicated, surprising tricks (Section IV) like utilizing the status of video playing.

Detection. Also many new techniques have been proposed to automatically detect HSO activities, in binary executables or mobile code [29], [27], [40], [39], [38], [28], [34], [9], [53].

Several studies propose to capture the behavior deviation from the potentially harmful apps in different environments [29], [27], [40], [39], [38]. The idea is based upon the observation that anti-emulator HSOs are very likely to act differently when running in an analysis environment and when operating on uninstrumented devices. Since such a behavior comparison needs to be done using the same execution trace, these approaches usually record an app's interactions with the system in one environment and replay them to the same app in another. As we can see here, these approaches target on anti-emulator, while HSOMINER is designed for a more general purpose, aiming at different kinds of HSO activities. Also these dynamic analysis based techniques tend to be heavyweight and less comprehensive. By comparison, our approach uses static analysis and can therefore be scaled to the level of hundreds of thousands of apps.

Another line of research focuses on trigger analysis [28], [34], based on the observation that trigger conditions usually rely on some unique inputs of interest to the adversary (e.g., time, network), and sensitive operations are triggered only when a narrow requirement is met (e.g., current date equals a predefined trigger date; mobile device is located in a certain area). To automatically identify such trigger-based behaviors, these approaches often leverage techniques like symbolic execution, dynamic instrumentation and formal verification. Among them, most relevant to our work is TriggerScope [34], which aims to detect some types of logic bombs (time-, location- and SMS-triggered HSOs). Compared to HSOMINER, one main difference is that TriggerScope is designed to capture known HSO cases and its expected trigger conditions are very specific. In contrast, HSOMINER only assumes that a trigger condition is supposed to receive certain system inputs, which makes it possible to identify unknown HSO. Another important difference is that TriggerScope is built on heavy-weight techniques like symbolic execution, while HSOMINER utilizes efficient feature extraction, which makes it more suitable for a large scale analysis.

Also related to our research is AppContext [51], a PHA detection system based on supervised machine learning. It leverages the observation that benign and malicious behaviors could be differentiated from their context, e.g., UI events, system events and environment property methods. Although similar to HSOMINER in terms of using learning techniques, AppContext focuses more on PHA detection than on HSO behavior discovery.

Defense. In addition to the attempts to detect HSO, effort has been made to make an HSO technique less effective. As an example, Ether [32] leverages hardware virtualization extensions to stay transparent to malware. Another technique [36] dynamically modifies the execution of a whole-system emulator to mimic a real device in the face of anti-emulation malware. Similarly in Android, implementations have been proposed to make emulator transparent through runtime hooking and Android source modification [21], [24].

VI. DISCUSSION

With its significant step towards understanding and ultimately defeating hidden sensitive operations, the current design and implementation of HSOMINER are still preliminary, leaving more to be desired. Here we discuss a few limitations of our approach and potential future directions.

Accuracy and completeness. HSOMINER is built on top of existing static analysis techniques, which are known to be less accurate, particularly when it comes to handling the Android's Inter-Component Communications (ICC). It is possible, for example, that the define-use analysis we employed gives inaccurate results. Also more likely, some apps are simply too complicated to be analyzed, as observed in our study. Although improving the precision and completeness of the underlying static analysis techniques are out of the scope of this study, it is important to point out that HSOMINER is designed to be less dependent on accuracy of individual features. Instead, we attempt to leverage a collection of attributes extracted from a program to identify HSO behaviors, even when the individual attributes are contaminated with a certain level of noise. We strongly believe that leveraging the collective power of less accurate yet differentiating program features opens a promising new direction for a more cost-effective security analysis.

Also, even though the design of HSOMINER is more generic than existing approaches, which enables it to catch new HSO instances, it is still based on a set of assumptions that fail to cover some HSO cases. For example, theoretically, a trigger condition does not need to involve any system inputs: hidden behaviors could be activated once an activity has been visited for a certain number of times. In practice, however, the trick of this type has never been observed before. Most importantly, HSOMINER classifies a branch structure based upon a set of features, which limits the impact of a single feature, and could still capture the case mentioned above. This needs to be further investigated in the future research.

Evasion. Just like all existing approaches [51], [34], HSOMINER could be evaded by carefully designed HSO techniques. An HSO branch can always be built in a way that mimics legitimate branches to make our technique less effective. On the other hand, we argue that the design philosophy of HSOMINER will make such attacks more difficult to succeed: the adversary may not be able to bypass our defense by simply avoiding one or two features; she needs to consider the identification power from the combination of multiple features

to carefully come up with a strategy to cheat our classifier. Even when the features selected for our implementation become less differentiating in the presence of a new attack, the framework of HSOMINER can easily accommodate other features, further raising the bar to the evasion attempts. Therefore, we have reason to believe that techniques like HSOMINER can be more robust than the approach based upon a single feature, like TriggerScope [34]. In the meantime, future research is certainly needed to better understand the cost of evasion under our approach and find new ways to enhance our technique against such a threat.

Also our current implementation cannot handle the HSO embedded in the native code. However, our design could be extended to help identify the hidden behaviors there. Further, HSO triggers can be deployed on the server side, which cannot be directly observed by our static analyzer. On the other hand, such a technique requires the server to send commands to the app, which will also need to be checked within the app. This just provides an opportunity for identifying hidden operations using our technique. Further, the adversary could obfuscate the code using reflection and other techniques to undermine the effectiveness of HSOMINER. Such attempts, however, could be detected by existing techniques [50], [47], making a PHA less stealthy. UI based HSO activities can also be hard to capture, due to the challenge in differentiating them from legitimate UI related operations. The problem could be addressed through leveraging the semantic relations between UI text and app behaviors, using AutoCog [46] and existing Natural Language Processing tools to extract this feature. However, for the UI related behaviors solely based on callbacks rather than the traditional *branch* structure, new techniques need to be developed to detect them.

Further optimization. As mentioned earlier, HSOMINER is meant to be efficient, performing only lightweight code analysis for feature extraction. Our current implementation, however, still involves some heavyweight techniques. Particularly, building global subgraph, which requires analyzing ICC, is expensive. In practice, however, we found that in the most cases, features identified locally (without ICC mapping) are sufficient for catching HSO branches. Future research may look into the potential to extract features from Intent construction and ICC components for behavior classification, to avoid the expensive mapping of ICC, and other alternatives to further improve the scalability of our approach.

VII. CONCLUSION

Hidden sensitive operations (HSO) have long been used by malware to evade detection. Today, the techniques of this kind gain new traction in the mobile PHA community. With the reports about various anti-emulation apps against the vetting process of app markets [17], [43] and other HSO strategies, little is known about the pervasiveness of the threat, their technical trends and impacts, due largely to the challenges in systematically discovering and analyzing real-world HSO instances. Although effort has been made to detect such activities statically, existing techniques are tuned toward specific trigger conditions or hidden behaviors, and also pretty heavyweight, rendering them less effective in detecting previously unknown HSO apps on a large scale.

In this paper, we report an innovative technique that makes it possible to analyze and discover unknown HSO instances across a large number of real-world Android apps. Our approach, HSOMINER, is built upon a set of unique observations about an HSO condition, its paths and the relations between them. Particularly, we found that such a condition tends to involve system inputs, but is less likely to link to its branch paths through data flows or shared resources. Further the behaviors between a hidden path and the counterpart that covers it are often very different. Features summarized over these observations were found to be easy to extract and highly effective when used collectively. In our research, we implemented a prototype system that runs a classifier to detect HSO instances with these features. It was designed to be general, well-equipped to catch previously unknown HSO instances. Our study shows that the new technique achieved over 98% precision and over 94% coverage. It is also efficient, enabling us to perform a large measurement study over 330K recent apps. This study sheds new light on Android HSO activities in the wild, and reveals the pervasiveness of HSO behaviors and new techniques deployed by the HSO authors, including the extensive use of UIs, system events and services as trigger conditions and surprising hidden behaviors. Our new technique, together with the new understanding, contributes to the better protection against this emerging threat to the mobile ecosystem.

ACKNOWLEDGMENT

We thank anonymous reviewers for their insightful comments. This work was supported in part by the National Science Foundation under grant 1223477, 1223495, 1527141, 1618493, 1664315, U.S. Army Research Office under grant W911NF1610127, Air Force Research Lab under grant FA8750 -15-2-0106, and DARPA under grant FA8750-16-C-0044. We would also like to thank VirusTotal for granting us the privilege for the large scale query and app downloading.

REFERENCES

[1] 360 security. http://www.360securityapps.com/en-us.

[2] Android malware evasion techniques-emulator detection. http://www.oguzhantopgul.com/2014/12/android-malware-evasion-techniques.html.

[3] Android malware set for july 4 carries political message. https://blogs.mcafee.com/consumer/android-malware-set-for-july-4-carries-political-message/.

[4] Android security: Adding tampering detection to your app. https://www.airpair.com/android/posts/adding-tampering-detection-to-your-android-app#4-1-emulator.

[5] Android.hehe: Malware now disconnects phone calls. https://www.fireeye.com/blog/threat-research/2014/01/android-hehe-malware-now-disconnects-phone-calls.html.

[6] Apktool-a tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/.

[7] Application security. https://source.android.com/security/overview/app-security.html.

[8] Average cpi per country. http://www.appbrain.com/stats/android-cpi-per-country.

[9] Detecting malware and sandbox evasion techniques. https://www.sans.org/reading-room/whitepapers/forensics/detecting-malware-sandbox-evasion-techniques-36667.

[10] Dex education 201: Anti-emulators. http://hitcon.org/2013/download/Tim%20Strazzere%20-%20DexEducation.pdf.

[11] Fireeye multi-vector virtual execution (mvx) engine. http://www.threatprotectworks.com/MVX-engine.asp.

[12] Mobile threat report-mcafee. http://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf.

[13] monkeyrunner. https://developer.android.com/studio/test/monkeyrunner/index.html.

[14] Programmers united develop net. http://en.pudn.com.

[15] Soot, a framework for analyzing and transforming java and android applications. https://sable.github.io/soot/.

[16] Virustotal-free online virus, malware and url scanner. https://www.virustotal.com.

[17] Blackhat usa 2012 - adventures in bouncer land. http://www.securitytube.net/video/8880, November 2013.

[18] Virtual machines and how malware authors know when they are being watched. https://securityintelligence.com/virtual-machines-malware-authors-being-watched/, October 2013.

[19] Does malware still detect virtual machines? http://www.symantec.com/connect/blogs/does-malware-still-detect-virtual-machines, August 2014.

[20] Hacking team rcs android source code. https://github.com/hackedteam/core-android/blob/master/RCSAndroid/src/com/android/dvci/Core.java, December 2014.

[21] Mindmac/hideandroidemulator. https://github.com/MindMac/HideAndroidEmulator, October 2014.

[22] Hacking team rcsandroid spying tool listens to calls; roots devices to get in. http://blog.trendmicro.com/trendlabs-security-intelligence/hacking-team-rcsandroid-spying-tool-listens-to-calls-roots-devices-to-get-in/, July 2015.

[23] The google android security team's classifications for potentially harmful applications. https://static.googleusercontent.com/media/source.android.com/en//security/reports/Google_Android_Security_PHA_classifications.pdf, April 2016.

[24] Xposed module repository. http://repo.xposed.info/, August 2016.

[25] S. Arzt, S. Rasthofer, and E. Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013.

[26] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[27] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient detection of split personalities in malware. In *NDSS*. Citeseer, 2010.

[28] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[29] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 177–186. IEEE, 2008.

[30] J. R. Crandall, G. Wassermann, D. A. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong. Temporal search: Detecting hidden malware timebombs with virtual machines. In *ACM Sigplan Notices*, volume 41, pages 25–36. ACM, 2006.

[31] W. Diao, X. Liu, Z. Li, and K. Zhang. Evading android runtime analysis through detecting programmed interactions. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, 2016.

[32] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[33] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM, 2014.

[34] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Triggerscope: Towards detecting logic bombs in android applications. 2016.

[35] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 216–225. ACM, 2014.

[36] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating emulation-resistant malware. In *Proceedings of the 1st ACM workshop on Virtual machine security*, pages 11–22. ACM, 2009.

[37] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 637–652, 2013.

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

[39] D. Kirat, G. Vigna, and C. Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 287–301, 2014.

[40] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti. Detecting environment-sensitive malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 338–357. Springer, 2011.

[41] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[42] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234. ACM, 2013.

[43] J. Oberheide and C. Miller. Dissecting the android bouncer. *Summer-Con2012, New York*, 2012.

[44] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, 2013.

[45] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[46] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1354–1365. ACM, 2014.

[47] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.

[48] T. Vidas and N. Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 447–458. ACM, 2014.

[49] H. Wang, Y. Guo, Z. Ma, and X. Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 71–82. ACM, 2015.

[50] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TRISECLAB-0414*, 1:5, 2014.

[51] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 303–313. IEEE, 2015.

[52] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

[53] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM, 2012.