Hindawi
Wireless Communications and Mobile Computing
Volume 2018, Article ID 2769417, 19 pages
https://doi.org/10.1155/2018/2769417

Research Article

Hydra-Bite: Static Taint Immunity, Split, and Complot Based Information Capture Method for Android Device

Ziru Peng[1,2], **Xiangyang Luo**[1,2], **Fan Zhao**[1,2], **Qingfeng Cheng**[1,2], **and Fenlin Liu**[1,2]

[1] *State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China*
[2] *Zhengzhou Science and Technology Institute, Zhengzhou 450001, China*

Correspondence should be addressed to Xiangyang Luo; luoxy_ieu@sina.com

Received 8 March 2018; Revised 17 April 2018; Accepted 23 May 2018; Published 17 July 2018

Academic Editor: Kim-Kwang Raymond Choo

In order to attract attention to the malicious use of large-scale operation of applications, Hydra-Bite, an Android device privacy leak path implemented by splitting traditional malicious application and restructuring to a collaborative application group, is proposed in this paper. For Hydra-Bite, firstly, traditional privacy stealing Trojan is analyzed to obtain the permission set. And the permission set redundancy elimination splitting algorithm is subsequently adopted to extract the simplest key permission set and split the set by functions so as to form the collaborative application group. Then, a covert channel is adopted for the intergroup Apps to remove the information's taint tagged by security methods. Meanwhile, a communication medium selection algorithm and an information normalization coding method are proposed to improve the efficiency and the concealing property for taints removal. Finally, collaborative external transmission of information is realized on the basis of intragroup Apps' communication. The experimental results show that Hydra-Bite could resist the detecting and killing of about 60 security engines such as Kaspersky, McAfee, and Qihoo-360 in VirusTotal platform and capture the privacy information of the devices of different versions from Android 4.0 to Android 7.0. Hydra-Bite can resist the killing of the following two methods, the typical detection tool Androguard based on "permission-API" and the typical static taint tracking tool FlowDroid. Compared with traditional privacy stealing Trojan, Hydra-Bite has higher information capture rate and stronger antikilling performance.

1. Introduction

Android operating system is widely applied in ILDs (Intelligent Devices), covering home furnishing, communication, business and vehicle-mounted terminals, etc. As reported, Android operating system has a global occupancy of 86.2% in the ILD market. ILD can store massive key information of users, e.g., location, communication records, accounts, and movement tracks. Along with the large-scale operation of application programs, that is, the same operational entity operates multiple Apps, such operation mode may be utilized by information selling organization and the key information of the users may be stolen by interapplication collaboration. We explore and report such stealing mode. The first purpose is, at the research level, attracting the attention of relevant security researchers. The second purpose is, at the application level, promoting the research on the App security audit mechanism in the platform. Based on the above purposes, this paper wants to prevent potential large-scale user information collection behavior in ILDs.

Traditional information stealing Trojan is mainly implemented by a single App. Those Trojans can be divided into two categories. The first category is Root permission applying (Root for short) and the second category is non-Root permission applying (non-Root for short). Specifically, typical Root approaches include Rootcager [1], Hellfire, Jmedia, and Bgserv [2]. Once "Hellfire" succeeds in promoting to Root permission through in-packet nesting, cloud matching, etc., it has extremely strong antikilling ability. In this condition, common antivirus software cannot completely remove it. However, the Root category is fragile. For example, before installation, Root method greatly depends on user state and system environment; after installation, it may trigger antivirus software's alarm immediately before promoting to the permission, so Root category has poor operability. Typical non-Root category includes methods which are Zsone [3],

GPSSPY and Nickyspy [4], SMS Tracker, Spitmo, and Zitmo, and the information is stolen usually through the application for excessive permission. Typical non-Root Trojan is the Soundcomber [5] which steals the information through call recording, speech recognition, and other technologies. Compared with the method of capturing key information by applying Root permission, other methods that do not apply for Root permissions have low recognition degree but the methods that do not apply for Root permission are easily blocked or detected by the users through searching and uploading information behaviors, especially after dynamic permission mechanism is introduced into Android. These Trojans can steal a lot of information, thus causing harm to equipment privacy. For example, leakage of Wi-Fi information and social information may cause devices' location information to be tracked [6–8].

For the above information stealing Trojan, researchers have proposed multiple information protection strategies. These strategies can be divided into "permission-API" detection and "taints tracking." The "permission-API" detection strategy includes the typical selectable authorization tool Kirin [10] and its improved version Apex [11], excessive permission detection tool Stowaway [12] and PScout [13], etc. They judge whether App is malicious through calling sensitive API and detecting risk permission combination, but such mode has high false alarm rate and it is difficult to cope with the privacy stealing method based on "collaboration." Therefore, the "static taint tracking" of key information has been researched more and more since 2013. The representative tools are as follows: static analysis tool ScanDroid [14], DroidChecker [15], Chex [16], COVERT [17], etc. These tools carry out static analysis of App through detection of permission, sensitive data leak path and data-flow analysis, etc. Additionally, App data-flow analysis tool FlowDroid [9] establishes the propagation path of the key information from "sensitive source" (e.g., IMEI number, longitude, and latitude) to "receiving node, sink" (e.g., sending short message, uploading in Internet) through static taint tracking, thus tracking the taint information flow. At present, the method has been taken by several security engines as the analysis kernel of Android application, but it is difficult to implement "static taint tracking" on the system bottom frame. The literatures [18–20] are other covert communication detection methods.

In order to attract attention to the large-scale operation of Apps used maliciously, this paper researches the Android system's application layer and proposes Hydra-Bite. Specifically, firstly, Hydra-Bite uses the permission split and reconstruction module to split traditional privacy stealing Trojan, and collaborative App group is constructed. In the first step, the problem is the coarse permission particle size of the collaborative application group. To solve the above problem, the permission set redundancy elimination splitting algorithm is proposed to extract the key permission set and split it by functions. Secondly, the taint cleaning module and taint tagged by the static taint tracking method on the key information are cleaned through Android covert channel. To solve the problem of wide varieties of communication medium and bandwidth [23], information normalization coding method is proposed. In the second step, to solve the problem that communication medium is easily occupied by irregular user operations, the communication medium selection algorithm is proposed. The experimental results show that, compared with traditional privacy stealing Trojan, Hydra-Bite has higher information capture rate and stronger antisearching and antikilling rate.

This paper's major contributions are as follows.

(1) Static taint immunity, split, and complotted based information capture method: Hydra-Bite is an information capture method for Android devices. This method can get information and avoid killing by collaborative Apps. At the same time, Hydra-Bite cleans the mark tagged by security methods through covert channel. This paper sends a security alert about Hydra-Bite.

(2) Proper split method for permission sets: Hydra-Bite proposes permission set split after removal of redundancy algorithm. This algorithm can split the permission set of traditional malicious applications and reconstruct the permission sets after the split to be a cooperative application group.

(3) Information and communication media adaptation: Hydra-Bite proposes a normalized coding method. This method solves the problem of the difficulty of covert communication media and information adaptation before information enters the covert channel.

(4) Dynamic selection of communication media: Hydra-Bite proposes a dynamic selection algorithm for communication media. The algorithm solves the problem of how to dynamically select the largest communication bandwidth medium when some media are occupied.

(5) Taint cleaning: Hydra-Bite proves the shortage of existing static taint tracking methods. Hydra-Bite can bring coded information with taint through covert channel, so that the taint cannot be tracked.

The remaining content of this paper is arranged as follows: in Section 2, "Hellfire," "Soundcomber," and "FlowDroid" are taken as examples to introduce relevant work from the two aspects of "Trojan" and "Protection"; in Section 3, method's principles and steps are explained in detail; in Section 4, the proposed method is evaluated from the two aspects of "information capture" and "antikilling performance"; Section 5 is the conclusion.

2. Related Work

This section firstly illustrates the function basis of Hydra-Bite by introducing traditional data capture Trojans "Hundreds," "Gypsymoth," "Hellfire," and "Soundcomber"; then, through the introduction of the peculiarity of "Permission-API" and "static taint tracking," it explains the weakness of traditional privacy-preserving methods when facing the Hydra-Bite privacy leak path.

2.1. Traditional Key Information Capture Method. Traditional key information capture methods are based on obtaining Root permission and applying for excessive permissions. The following part will explain both in detail.

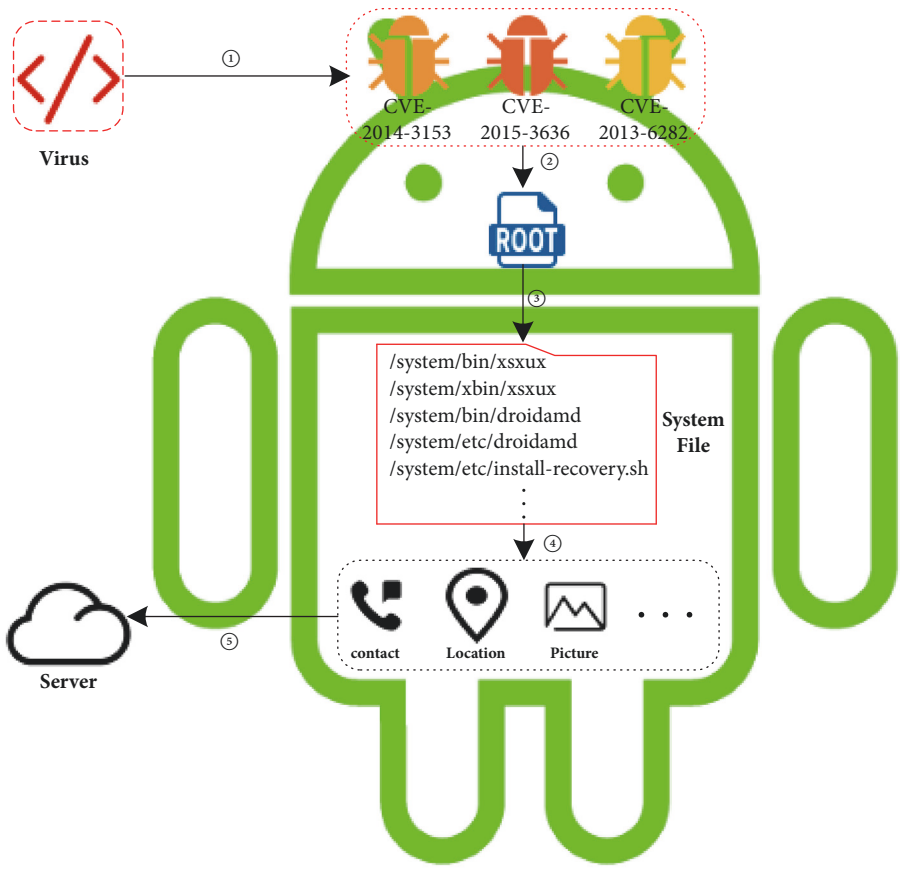

Figure 1: The process of "Hundreds" infects equipment.

(1) The Capturing Methods Based on Root Permission Acquisition. In June 2015, the virus named "Hundreds" is active. The principle of Hundreds to capture information is shown in Figure 1. Steps ①, ②, and ③: The App carrying virus enters the device and then releases the privilege promotion code and core module. Steps ④, ⑤, and ⑥: After the permission promotion code begins to work, the information of system version is uploaded to the server. Step ⑦: The server identifies the system and returns the Root scheme that is compatible with the current system version. Steps ⑧ and ⑨: The module uses the above Root scheme to invade the system folder for the convenience of fake itself as a system App, so that Hundreds can self-start and cannot be deleted. Steps ⑩ and ⑪: The Hundreds collects device information and uploads it to the server.

In December 2015, the virus named Gypsymoth is active. The principle of Gypsymoth to capture information is shown in Figure 2. Steps ① and ②: The virus has been promoted to Root permissions, through system's vulnerabilities. Step ③: Some system files are replaced by Gypsymoth after Root permission was promoted so that Gypsymoth can reside in the system by monitoring system running environment and sniffing file change. Steps ④ and ⑤: When the basic survival requirements are satisfied, Gypsymoth starts capturing information and uploads it to the server.

In June 2016, the virus named Hellfire is active. The principle of Hellfire to capture information is shown in Figure 3. Step ①: The virus carrying App enters the device. Step ②: The virus carrying App releases its subpackage which is used to obtain Root permission. Steps ③ and ④: The subpackage gets the SDK version of current system and sends it to server. Steps ⑤ and ⑥: Hellfire receives and executes Root scheme returned from server. After Root permission promotion, Hellfire can parasite to the underlying module of the system and reside in equipment to collect device information continuously.

Methods to capture key information have a significant effect when Root permission is promoted. However, these kinds of method have strong user and system environment dependence. These kinds of method do not work for systems that disable Root permission or users who pay attention to App's permissions.

(2) The Capturing Methods by Obtaining Excessive Permissions. Excessive permissions are the permissions beyond those which are necessary to meet the App's function. Method is based on excess permissions, usually applied for a large number of permissions and disguised as a normal application, by application repackaging.

In 2011, the Soundcomber [5] method was proposed by Schlegel et al. applied for excessive permission, such as sound

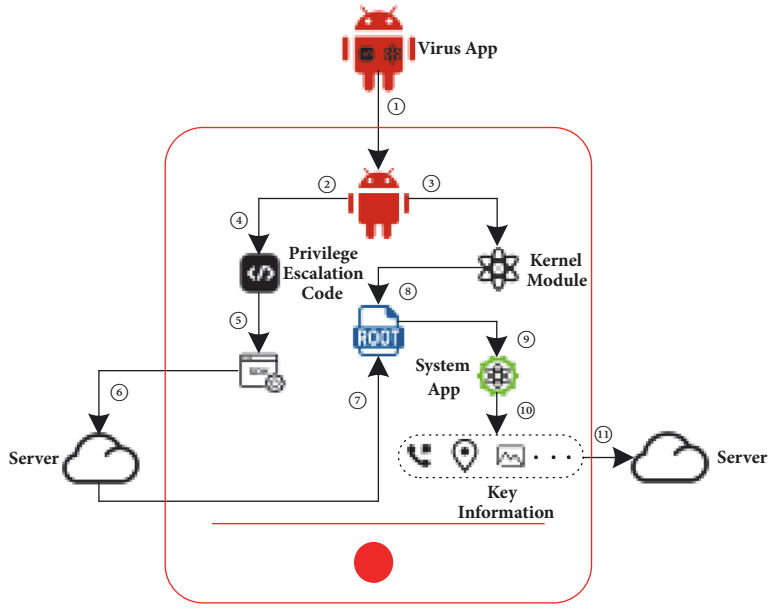

Figure 2: The process of "Gypsymoth" infects equipment.

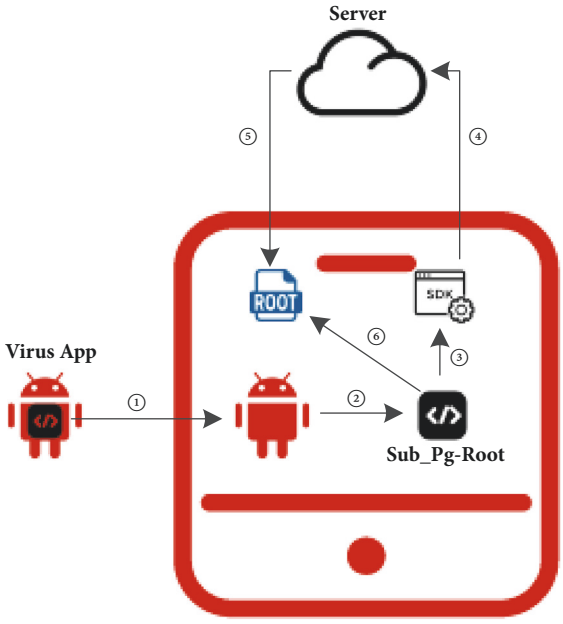

Figure 3: The process of "Hellfire" infects equipment.

recording, disk access, and Internet access. When monitoring the device calls, these permissions of Soundcomber are used to turn on the recording function. Then Soundcomber will store the audio files recorded before. Finally, when the condition is suitable, Soundcomber recognizes key words from audio files and uploads them to the network.

In 2012, the Tapprints [24] method proposed by Emiliano et al. applied permissions for accelerometer, gyroscope, and IMEI number. Tapprints identifies the device model with the IMEI number and then queries screen size by the device model. After that, Tapprints silently listens to users' clicking coordinates and clicking objects on screen. Tapprints combines machine learning method to infer users' input in devices, with the previous coordinates and objects.

In 2017, with the popularity of smart wearable equipment, Maiti et al. proposed a method of obtaining information user inputted through the smart wearers' sensors [25]. Their method applies for permissions to access sensors that belong to mobile communication equipment and wearable equipment. Finally, there is observation of hand movements by permissions applied before to improving the accuracy of information capturing.

The above device information stealing method based on excessive permission avoids the system environment dependence. However, this method still has strong user dependence. Meanwhile this kind of method can also be blocked by the existing security means [26, 27].

2.2. Methods for Preventing Key Information Capturing. To detect and block App's key information obtaining, there are mainly two methods: one based on detecting the mapping relation of "permission-API" and the other conducting static taint analysis on the App. The following part will introduce both in detail.

(1) The Protecting Methods Based on Detecting "Permission-API" Mapping. This detecting method is based on "permission-API" mapping which can estimate whether the Apps conduct key information capturing by analyzing if the App applies for high-risk permissions such as Root, or whether Apps call high-risk API combination such as recording and uploading. The representative within this type of protecting methods is PScout [13] which analyzes the Android system Source code and obtains the mapping relationship between API set and permission set, and it

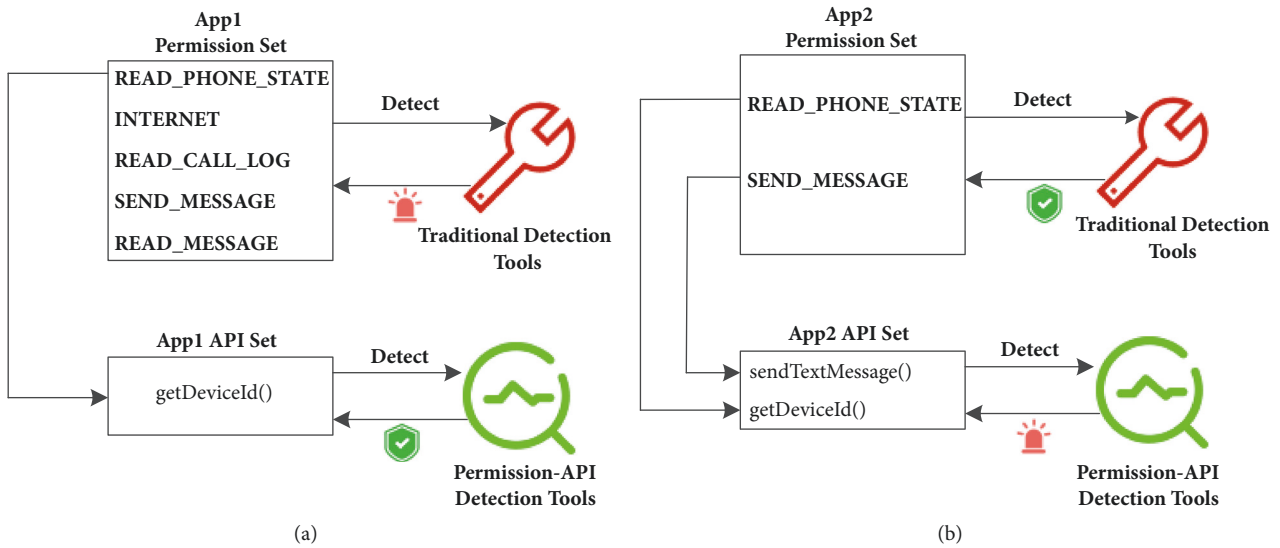

Figure 4: The different detection results between App1 and App2. (a) The detection results of App1. (b) The detection results of App2.

can conduct the API-Level analysis of the App through the particular mapping relationship.

As illustrated in Figure 4, App1 in Figure 4(a) is a benign application as it applies for unused sensitive permissions but it does not call any API to leak key information; App2 on Figure 4(b) only applies for "reading IMEI number" and "sending text messages" two permissions, which contains key information capturing potential. If we only analyze App1 and App2 from the permission aspect, App1 would trigger the alert and be wrongly diagnosed as it applies for lots of key information reading and uploading permissions and generates high-risk combination. However, if we analyze App1 and App2 from the more detailed API calling aspect through "permission-API" detecting method, App2 would trigger the alert as it calls for more dangerous API combination. Therefore, it improves the detecting accuracy through the API-Level scanning of the App.

(2) The Protecting Methods Based on Static Taint Tracking of the App. Static taint tracking method detects the complete path from where the key information firstly gets captured at the Source, to the sink point where it has leaked out of the App by analyzing the App on the Source level. This analysis method can monitor the data-flow path of Apps' key information reading and block the capture method through Apps applying for excess permissions. The representative work is FlowDroid [9], an Android App's high-accuracy static taint tracking method invented by Arzt et al. By imitating the complete Android life cycle, FlowDroid uses analysis method according to different demands, which is highly accurate and highly efficient comparing to other methods of this type.

Figure 5 is an illustration of taint analysis in real circumstances. This illustration contains two parts; the first one is forward taint analysis (①, ②, and ⑦), which tracks the flow of taint variables. The second one is backwards on-demand analysis, which detects the aliases before they get tagged by the taints. FlowDroid generates complete taints

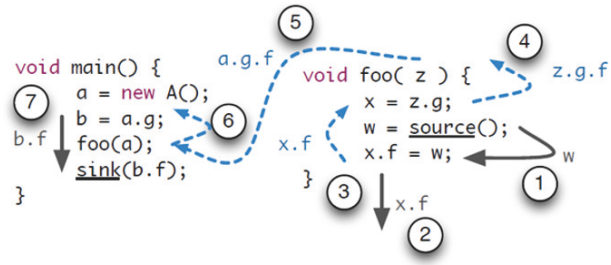

Figure 5: FlowDroid taint analysis [9].

path by using the above-mentioned forward taint analysis and backwards on-demand analysis. However, it faces APK files, and FlowDroid method can hardly track the taints hidden deep down in the bottom of the system.

3. Proposed Method

This section will outline the architecture of Hydra-Bite, according to Figure 6. Firstly, some terms and concepts which will appear in the later description are defined. Then, according to Figure 6, the process of Hydra-Bite will be outlined.

3.1. Terms and Definitions. Hydra-Bite will be introduced systematically in Section 3.1. The principle of Hydra-Bite is shown in Figure 4. And, for the convenience of the following description, definitions of terminology, abbreviated, are shown as follows:

(1) Traditional malicious applications: App_TM, normal applications: App_Norm.

(2) Key information (K_Info): The information can be read only when privacy-related permissions applied.

(3) Permission set (*PSet*): All permissions applied by the application form a set, marked as *PSet*, *PSet* can be divided

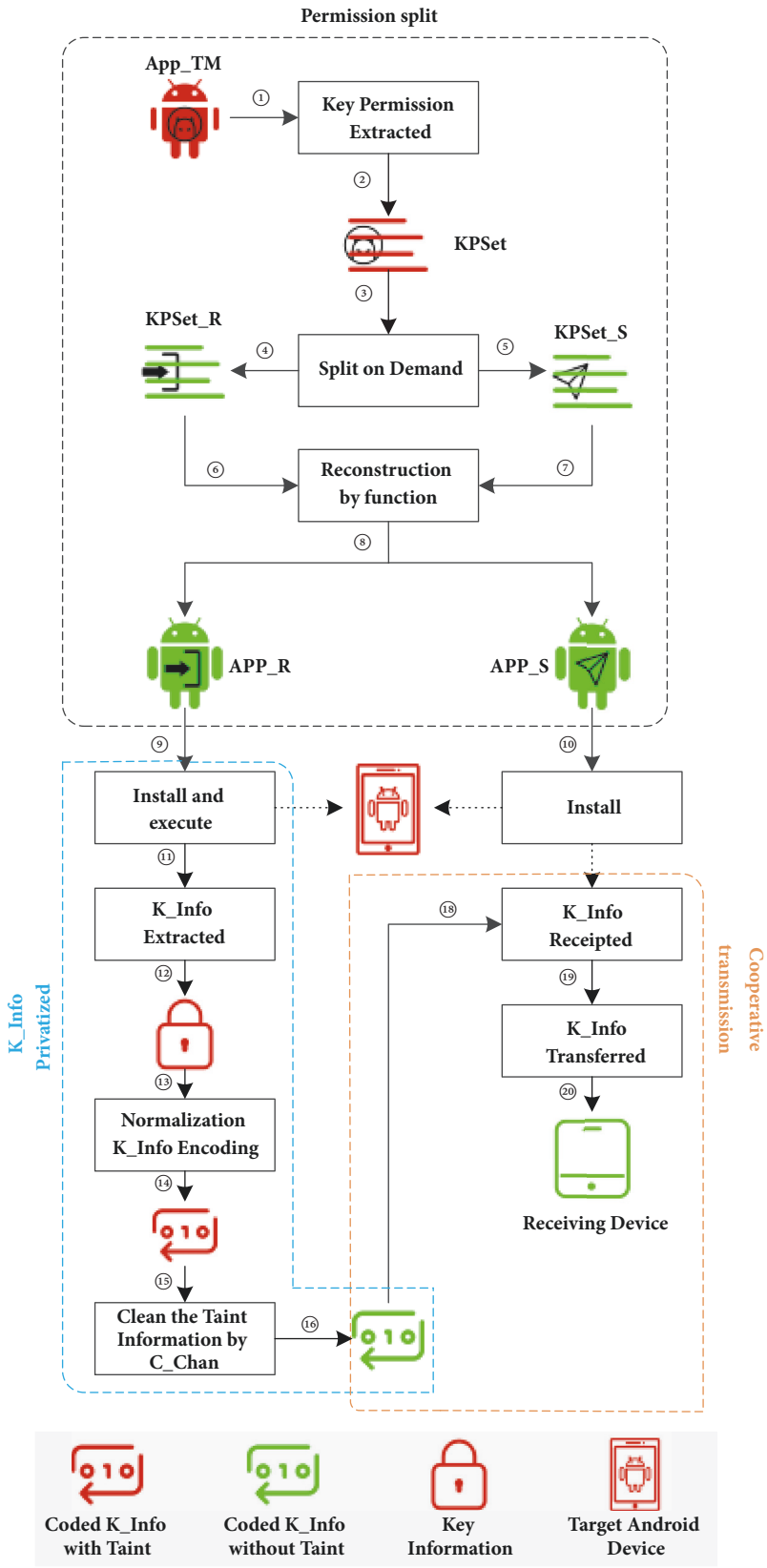

Figure 6: The general framework of Hydra-Bite.

into subsets *PSet_R* and *PSet_S* according to the process of information reading and sending. The relations between the above two subsets are as follows:

$$PSet\_R \cap PSet\_S = \emptyset.$$

$$PSet\_R \cup PSet\_S \subseteq PSet.$$

(4) Key permission set (*KPSet*): Permissions in *PSet* only related to the whole process of obtaining K_Info are selected to constitute a set, recorded as *KPSet*. The *KPSet* can be further divided into read permissions' set *KPSet_R* and send permissions' set *KPSet_S* according to API calls in process of obtaining K_Info. The relations of the above sets are as follows:

$$KPSet\_R \subseteq PSet\_R.$$

$$KPSet\_S \subseteq PSet\_S.$$

$$KPSet\_R \cup KPSet\_S \subseteq KPSet.$$

$$KPSet\_R \cap KPSet\_S = \emptyset.$$

$$KPSet \subseteq PSet.$$

(5) Key information with taint (K_Info_T): After K_Info is read, static taint tracing method will tag it, which is denoted as K_Info_T.

(6) Normalized coded key information with taint (K_Info_N_T): Normalized coded K_Info_T is written as K_Info_N_T.

(7) Normalized coded key information without taint (K_Info_N): After the taint cleaning process, K_Info_N_T's taint tag is washed off; the result is recorded as K_Info_N.

(8) Covert channel: It can be expressed as a triplet $\langle SRes, EM_h, EV_l \rangle$ [26] where *SRes* are shared resources in Android system; EM_h is a communication entity with higher security level which can modify *SRes*; EV_l is a communication entity with lower security level which can observe or perceive *SRes*. Communication from EM_h to EV_l is not allowed. Then the channel that completes the two-entity communication is called a covert channel.

(9) Cooperative applications group (Co_Apps): Co_Apps is created by our Hydra-Bite method, which can communicate with other applications which are in the group.

3.2. Process of Method. The method is composed of 3 parts: Co_Apps reformed before *KPSet* split modular, K_Info privatized modular, and cooperate transmission modular. The following steps will be described specifically based on Figure 6.

(1) Co_Apps Reformed before KPSet Split

(1) *KPSet* Extracted (① and ②). Our method parses App_TM's installation package to get the AndroidManifest file, from which *KPSet* is extracted according to API calls in process of obtaining K_Info.

(2) KPSet split (③, ④, and ⑤). Hydra-Bite splits *KPSet* into *KPSet_R* and *KPSet_S* according to the phases which are read and send in the process of App_TM obtaining K_Info;

(3) Co_Apps reformed (⑥⑦, and ⑧). The App_R and App_S are structured as Co_Apps, according to the *KPSet_R* and *KPSet_S* they only owned.

(2) K_Info Privatized

(1) K_Info read (⑨, ⑩, ⑪, and ⑫). After installing and executing on the target Android device, App_R will read the K_Info which is protected by the system security mechanism to the application layer. At this time, K_Info will be tagged by static taint method to become K_Info_T.

(2) K_Info_T normalized (⑬ and ⑭). After read by App_R, K_Info needs to be translated into a uniform format that supports covert channel, because of various forms. Therefore, Hydra-Bite normalizes K_Info_T to K_Info_N_T through coding it.

(3) K_Info_N_T cleaned (⑮ and ⑯). If K_Info_N_T is transferred at this time, it will be detected because of the tagged taint. To clean the taint, C_Chan is designed in our method, so that the Hydra-Bite can get the K_Info_N.

(3) Cooperative Transfer

(1) Message receive (⑱). App_R opens the information receiving component of App_S to transfer K_Info_N, after K_Info privatized operation is done. Now the K_Info_N is private information of App_R.

(2) Information transmission (⑲ and ⑳). After receiving the K_Info_N, App_R opens the information transmission component and sends the K_Info_N to the receiving equipment through the *KPSet_S* which it owns.

In the above processes, the key steps which will be explained separately are as follows: *KPSet* Split on demand(①–⑤) and the taint cleaning by covert channel(⑬–⑯).

4. Key Issues Description

This section will give a detailed description of the key issues raised in permission split. They are key permission set split after redundancy removal, key information normalized, communication media selection, and taint cleaning through covert channel.

4.1. Key Permission Set Split after Removal of Redundancy. Permission set redundancy means that, in the process of information capture, the required number of permissions for App calls sensitive APIs is less than permission items in AndroidManifest file. To solve the problem that the granularity of Co_App's permissions is coarse caused by permission redundancy, this section proposes a deduplication algorithm for permission set before splitting it. The algorithm uses double layer's key-value mapping to extract key permission set. Then algorithm sets different labels for the items in key permission set. Algorithm 1 is the pseudo code of the algorithm's main loop.

```
(1)  define sApiList <apiMap<api_Name, perm_Name >>:
       List<Map<String, String>>
(2)  define kPermList <permMap<perm_Name, flag>>:
       List<Map<String, String>>
(3)  while sApiList ≠ ∅ do
(4)    for i = 0 to sApiList.size()
       //Traversing list of key-value mappings for sensitive APIs
(5)      map sApiList.get(i).getKey() to perm via the Mapping file
         provided by Pscout
       //Mapping sensitive APIs to permissions corresponded
(6)      sApiList.get(i).getValue() ⟵ perm
(7)      if perm is the first time show then
       //Remove the duplicate permissions after the mapping
(8)        swich(perm)
       //Classify permissions before add them into the second key-value mapping
(9)        case perm is transfer class permission:
(10)         permMap.getKey() ⟵ perm
(11)         permMap.getValue() ⟵ trans
(12)         add Map2 to kPList
(13)         clear Map2
(14)       case perm is information read class permission:
(15)         permMap.getKey() ⟵ perm
(16)         permMap.getValue() ⟵ read
(17)         add Map2 to kPList
(18)         clear Map2
(19)       case perm is other permission:
(20)         drop perm
(21)     else
(22)       drop perm
(23)     end if
(24)   end for
(25) end while
```

Algorithm 1: Main loop of key information split.

The purpose of the first layer's key-value mapping is to extract the permissions actually used by the App and then remove the repeated entries in it. This process is implemented as follows. Firstly, the algorithm sets the APIs that App actually invokes as the keys. Then the algorithm maps the APIs to the required permissions for invoking them, through the mapping file provided by PScout [13]. The reason for the implementation of the algorithm is that one permission can call multiple API in some cases. For example, the permission READ_CALL_LOG can invoke APIs which can read call category and call time. This layer's key-value mapping is shown in (5)-(6) lines of pseudo code, in Algorithm 1.

The purpose of the second layer's key-value mapping is to split the permissions actually used by the App, according to category. This process is implemented as follows. Firstly, the algorithm refines the real permission set by extracting permission items related to key information capture. The product of this process is key permission set. Then, the algorithm classifies and gives labels to the key permission set's items, according to reading or sending information. Finally, the algorithm sets items of key permission set as the keys and sets the labels for each item as values. Finally, Hydra-Bite can split the key permission set according to the labels. This layer's key-value mapping is shown in (7)–(23) lines of pseudo code, in Algorithm 1.

4.2. Key Information Normalized. There are two problems to be solved for the pretreatment of key information. One problem is for key information. Because of the diverse storage types and rich contents of this information in Android device, we counted the storage formats of some pieces of key information stored in the device and shown in Table 1. "▲" indicates that the key information exists in this format or content. It can be seen that some key information stored in Table 1 is different in type and content. The other problem is for *SRes*. Different shared resources have different forms of existence and communication bandwidth. Take the volume of the alarm clock and the brightness of the screen as an example; the thresholds of volume and brightness are 0–8 and 0–255 integer numbers, respectively. It can be seen that, due to the disunity of format and content, the adaptation between shared resources and different key information is difficult.

Shared resources are determined by the system, and it is difficult to modify. Therefore, in this section, aiming at the above problems, a method to normalize key information is proposed. The method is divided into format unified module

Table 1: Storage formats of some pieces of key information and their contents.

	IMEI Number	Fine Location	Contacts	Call Type	Call Date	SMS content
			Information Type			
String	▲		▲			▲
Double		▲				
Int				▲		
Long					▲	
			Information Content			
Number	▲	▲	▲	▲	▲	▲
Characters		▲	▲		▲	▲
Letter			▲			▲
Chinese			▲			▲
Other Languages			▲			▲

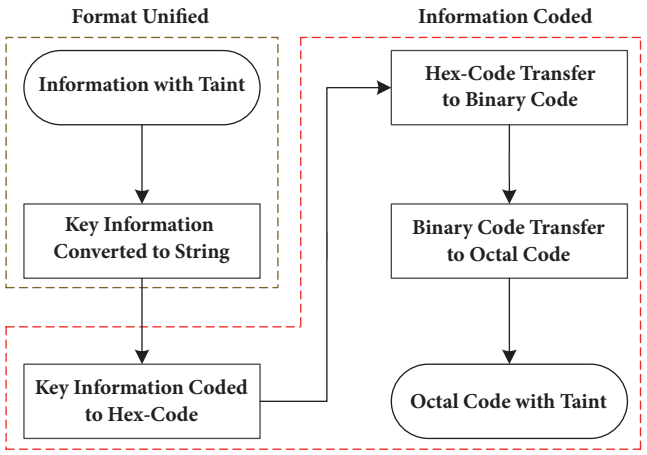

Figure 7: Key information normalization coding process.

and information coded module. And the flow chart is shown in Figure 7. The steps of the method are as follows.

Step 1. The key information to be processed is converted into string, which is convenient for subsequent coding.

Step 2. In order to unify the information format, all characters are converted into hex-code.

Step 3. Convert the hex-code into binary code and then turn binary code into octal code

For the convenience of shared resources' dynamic selection, each bit of coded information must be less than the bandwidth of the shared resource's threshold; besides the modification and observation of shared resources are a time-consuming process. Considering the time cost and bandwidth utilization rate, Hydra-Bite chose to turn the hex-code into octal code.

The number of bytes will increase after the information is encoded. This paper calculated the number of bytes after the information is encoded. When n is a number, the number of bytes before encoding and the number of bytes after encoding is (1). When n is a letter, the relationship is (2).

$$\text{Size}(n) = 3n - \left\lceil \frac{n}{3} \right\rceil, \quad n \text{ is a positive integer} \tag{1}$$

$$\text{Size}(n) = \begin{cases} 3n - \left\lfloor \frac{n}{3} \right\rfloor, & (n = 3i + 1,\ n, i \text{ are positive integers}) \\ 3n - \left\lceil \frac{n}{3} \right\rceil, & (n \neq 3i + 1,\ n, i \text{ are positive integers}). \end{cases} \tag{2}$$

4.3. Communication Media Selection. Another problem Hydra-Bite solved is the selection of shared resource. There are many kinds of shared resources, and the communication bandwidth between them is different. These shared resources are easily occupied by irregular user operations, which leads to low taint cleaning efficiency and the concealment of methods.

In this section, to solve the above problems, a communication media selection algorithm is proposed. By dynamically querying the occupancy status of shared resources, the algorithm iterates the largest *SRes* whose occupancy status is false. The result is the optimal shared resource.

The flow chart in Figure 8 and the pseudo code in Algorithm 2 describe the main loop of the algorithm. Firstly, a set of "medium-state" key-value mappings is used to identify the occupancy status of *SRes*. Hydra-Bite sets *SRes* as the "key" and sets the occupancy status as the "value." Algorithm queries *SRes* in real time and dynamically identifies its status, so as to realize the screening of available resources ((4)-(5) lines of pseudo code). Then Hydra-Bite queries the *SRes* in the "key-value" mapping table and recursively iterative "*fastKey*" with higher bandwidth to deploy *SRes* efficiently ((6)-(7) lines of pseudo code). The specific process of communication media dynamical selection is as follows.

Step 1. Define the state list *statList*<*SRes*> and its internal "key-value" map *SRes*< *SRes*, *OccStat*>, and set the names of shared resources as the "key" and occupancy status as the "value." Then judge whether the *statList* is empty, if *statList*

```
(1)  while statList ≠ ∅ do
(2)    declare fastKey : String
(3)    for i 0 to statList.size() by incr do //Traverse the list of key-value maps
(4)      while statList.get(i).getKey() is not occupied do
           //Determine whether the "key" in the current key-value map is occupied or not
(5)        Query bandwidth of statList.get(i).getKey()
             in speedMap<res, bandwidth>
           //Query unoccupied "key" bandwidth in the "SRes – Bandwidth" key table
(6)        if bandwidth > value.speedMap(fastKey)
             or value.speedMap(fastKey) = null then
(7)          fastKey ⟵ statList.get(i).getKey()
           //Iteration shared resources which has higher bandwidth
(8)        end if
(9)      end while
(10)   end for
(11)   return fastKey //Return the result of the iteration
(12) end while
```

Algorithm 2: Main loop of optimal shared resource selection.

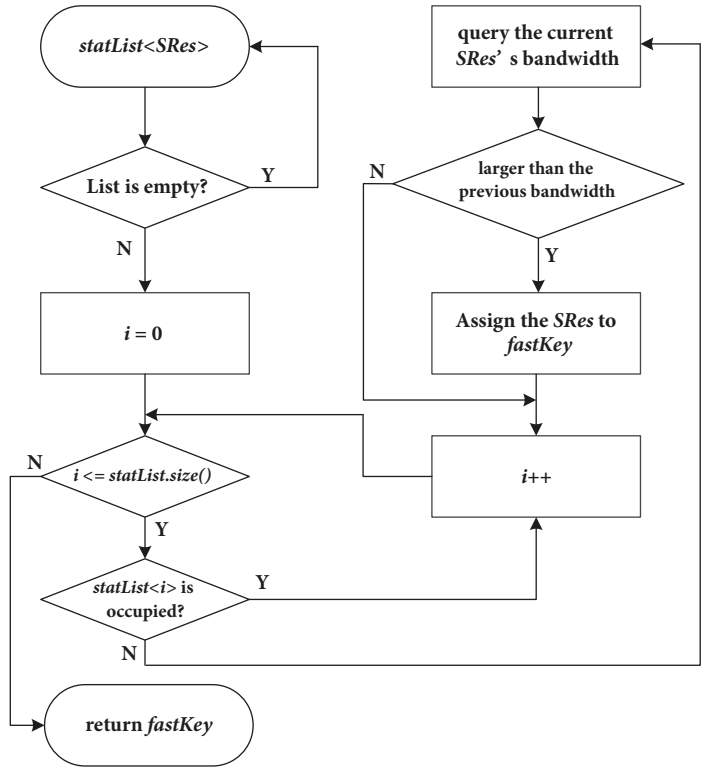

Figure 8: Main circulation flow chart of communication medium selection algorithm.

is empty, it indicates that the preset shared resources are all occupied by the user operation, and the current device state is not suitable for cleaning operation. If *statList* is not empty, then the state list is traversed.

Step 2. The traversal begins with the zero value in the *statList*. If the current state of *SRes* occupancy is true, the next value of *statList* will be judged. If the current state of *SRes* occupancy is false, then query the current *SRes* bandwidth.

Step 3. Compare the bandwidth of the current *SRes* with the previous bandwidth. If the current *SRes*'s bandwidth is large, assign the current bandwidth to the *fastKey*. If the current bandwidth value is small, then the first *SRes*'s bandwidth is still *fastKey*.

Step 4. No matter what *SRes* bandwidth value *fastKey* uses in the previous step, continue traversing *statList* until the loop end condition is satisfied.

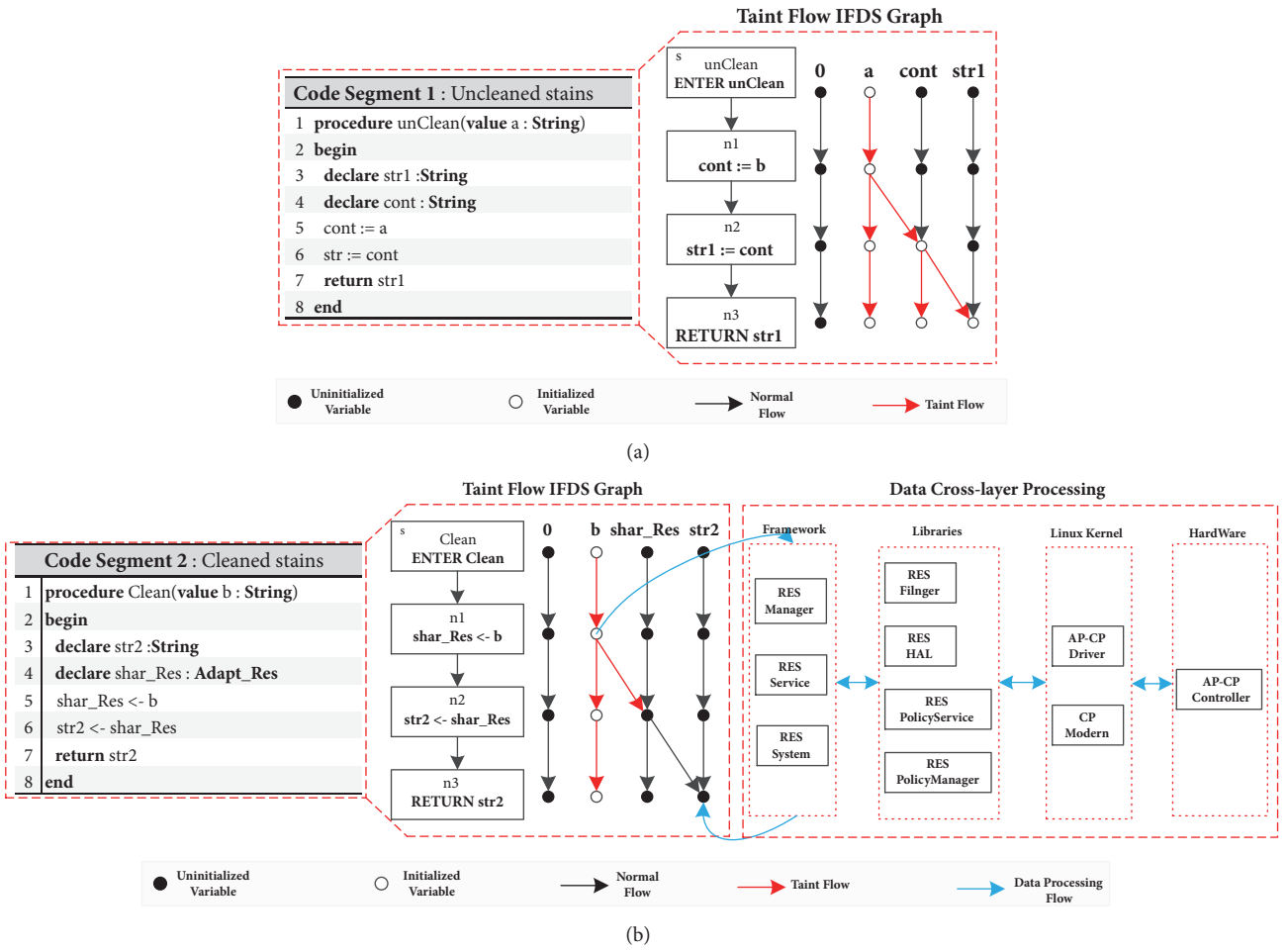

FIGURE 9: Taint analysis for different mode of transmission. (a) Transfer information directly via variables. (b) Transfer information via covert channel.

4.4. Taint Cleaning through Covert Channel. In the discussion of this section, readers need to know the definition of covert channel in Section 3.1 [26] and the IFDS algorithm [28].

Data flowing to the lower system layer are hard to be tracked by static taint tracking method (STT). Thus taint carried by EM_h cannot be tracked continuously by STT in the process of flowing to *SRes* at the bottom of operation system. The same reason can be obtained; Source of information read by EV_l from *SRes* cannot be tracked by STT. This communication is not allowed because the two entities do not communicate via security policy. At this point $\langle SRes, EM_h, EV_l \rangle$ has formed a covert channel communication triples.

Figure 5 shows the principle of EM_h, EV_l to get rid of the taint tracking by covert channel. The tainted data flow is shown in red directed line in this figure. On the left of Figure 9(a), the code segment 1 shows the process by which EM_h (**a**) propagates the information it carries through variable **cont** to EV_l (**str1**). On the right of Figure 9(a), the IFDS Graph is the results of data-flow analysis based on graph reachability of **cont** and **str1** using IFDS algorithm [28]. In Figure 9(b) the code segment 2 shows the communication entities **b** and **str2** realize the purpose of transmitting the key information by the covert channel through modification and reading *SRes* synchronous. Hydra-Bite needs to go through the framework layer, the library layer, and the kernel layer until the hardware layer to modify and read *SRes*. Take volume as an example; to change the volume, the volume manager(AudioManger) needs to call the system volume service(AudioService) to enter the volume system(AudioSystem), which can operate the AP-CP driver in the hardware abstraction layer (HAL). At this point STT is difficult to tag shar_Res(EM_h) and str2(EV_l).

5. Experimental Results

To verify the effectiveness of the Hydra-Bite method, this paper verifies the threat posed by the Hydra-Bite method to the privacy information itself through key information acquisition experiment and demonstrates the functional accessibility of Hydra-Bite. The experiment of run-time overhead is used to verify that the Hydra-Bite method is sufficient to transmit enough information in a limited time, which demonstrates the practicability of the method in time

Table 2: Test equipment and corresponding information.

Serial Number	Android Kernel Version	Android API	Market Share	Device Model
1	Android4.0	15	0.40%	Galaxy Note II
2	Android4.1	16	1.70%	MI 2
3	Android4.3	18	0.70%	MI 2S
4	Android4.4	19	12.00%	MI 3
5	Android5.0	21	5.40%	MI 4LTE-CU
6	Android5.1	22	19.20%	MI 4-LTE
7	Android6.0	23	28.10%	OPPO-A57
8	Android7.0	24	28.50%	MI 6
Total	—	—	96.00%	—

Table 3: The results of Co_Apps read-receive-send K_Info.

Permission	Android 4.0	Android 4.1	Android 4.3	Android 4.4	Android 5.0	Android 5.1	Android 6.0	Android 7.0
SendInfo Read the Key Information								
Location	▲	▲	▲	▲	▲	▲	★	★
Device State	▲	▲	▲	▲	▲	▲	▲	▲
Contact	▲	▲	▲	▲	▲	▲	★	★
SMS Message	▲	▲	▲	▲	▲	▲	★	★
WiFi State	▲	▲	▲	▲	▲	▲	▲	▲
Call Log	▲	▲	▲	▲	▲	▲	▲	▲
GetInfo Receive the Key Information								
Location	▲	▲	▲	▲	▲	▲	▲	▲
Device State	▲	▲	▲	▲	▲	▲	▲	▲
Contact	▲	▲	▲	▲	▲	▲	▲	▲
SMS Message	▲	▲	▲	▲	▲	▲	▲	▲
WiFi State	▲	▲	▲	▲	▲	▲	▲	▲
Call Log	▲	▲	▲	▲	▲	▲	▲	▲
GetInfo Send the Key Information								
Internet	▲	▲	▲	▲	▲	▲	▲	▲
SMS	▲	▲	▲	▲	▲	▲	★	★

overhead. The antikilling performance experiment is used to verify that Hydra-Bite method can void existing mainstream killing method, which demonstrates the tolerance of Hydra-Bite method in the face of the killing method.

5.1. Key Information Acquisition Experiment

(1) Key Information Acquisition Experiment's Set. In this section, to evaluate the performance of capturing key information, the Hydra-Bite method will be implemented in 8 different Android versions. Those versions are Android 4.0, Android 4.1... Android 7.0. Experimental device models and corresponding Android kernel version, Android API, and market share are shown in Table 2, where the market share uses the Google official website for week 1, 2018 statistics: developer.android.com.

(2) Key Information Acquisition Experiment's Results and Discussion. App_R and App_S are constructed to evaluate the Hydra-Bite's ability of capturing key information. App_R is constructed to read and clean the key information. App_S is constructed to receive the cleaned information from App_R and send to receive devices. In the 8 versions of Android listed in Table 2, this section tested the capturing ability of Co_Apps to six key information items listed in Table 3 which are geographic location, device status, equipment state, etc. Those key information items are marked by the Android system as requiring dangerous permission to access. The experimental results are shown in Table 3: "▲" and "★," respectively, represent that this operation is successful and this operation needs to be granted permission dynamically.

In Table 3, data, respectively, indicate the results of the operation: App_R to get key information, App_S to receive App_R's messages, and App_S to send key information outside. Before Android 6.0, Co_Apps can read key information directly and send out. After Android 6.0, there are inquiries when Co_Apps reads and sends. The reason for the above results is that the system after Android 6.0 adopts dynamic

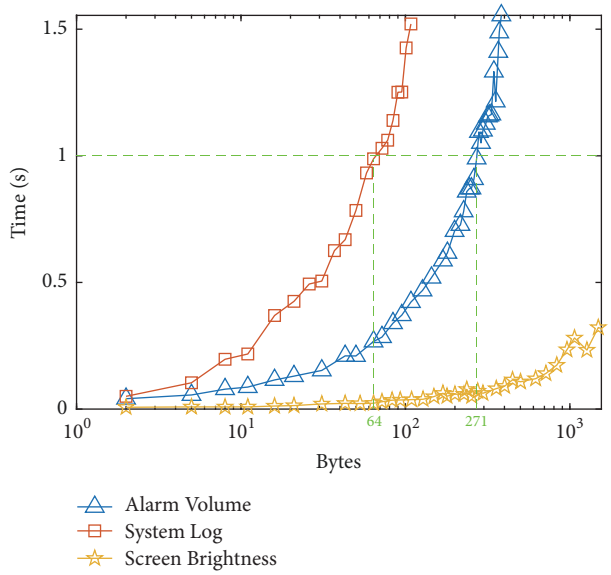

Figure 10: Overhead of taint cleaning.

permission granting mechanism and lets the user decide whether or not to grant app permission at run-time.

5.2. Running Time Overhead Performance Experiment

(1) Run-Time Overhead Performance Experiment's Settings. Right in the Hydra-Bite method EM_h the taint cleaning operation is carried out via covert channels, where access to the underlying resource is required twice per byte operation, which can be time-consuming. This paper illustrates the practicality of the Hydra-Bite approach in terms of time cost by examining the time overhead of the taint cleaning process.

In the experiment of running overhead performance, this section uses the device model MI_NOTE to test the time overhead of three covert channels which are "alarm volume," "screen brightness," and "system log." The device is equipped with Android 6.0 system and other hardware parameters that affect performance are listed below. The experimental device uses a 4-core CPU, whose basic frequency is 2.45GHz. The running memory of device is 3.00GB. And the kernel's version is 3.4.0-gf4b741d-00639-ge918701.

(2) Run-Time Overhead Performance Experiment's Results and Discussion. Run-time overhead performance experiment's results are shown in Figure 10. The results of each covert channel are marked with different symbols and colors. The green dotted line represents the amount of data that can be cleaned within a second.

It can be seen that when coded information has the number of bytes, system log covert channel's time overhead is the most expensive, because Android system processes generate a large number of logs. Filtering out encoded information from these logs is time-consuming. Screen brightness covert channel time overhead is minimum, because the value access and the brightness change are completed by different system modules, and this "separation" makes it easier to avoid waiting for hardware response. Cleaning rates of the above three covert channels are 78B/s, 280B/s, and 5.5KB/s, which are enough to transmit a certain amount of information.

Table 4: Part of antivirus engine in VirusTotal.

Serial Number	Antivirus Engine	Country
1	Alibaba	CHN
2	Antiy-AVL	CHN
3	Baidu	CHN
4	BitDefender	ROU
5	Arcabit	POL
6	CAT-QuickHeal	IND
7	Comodo	USA
8	Jiangmin	CHN
9	Kaspersky	USA
10	Kingsoft	CHN
11	McAfee	USA
12	Microsoft	USA
13	Qihoo-360	CHN

Table 5: Source of malicious sample.

Serial Number	Origin	Country	Quantity
1	VirousShare	USA	109
2	MalGenome	USA	73
3	GitHub	USA	26
4	zeltser.com	USA	19
5	bbs.pediy.com	CHN	17
6	bbs.kafan.cn	CHN	15
7	Google Group-	USA	14
9	bbs.duba.net	CHN	14
10	52pojie.cn	CHN	13
11	bbs.mumayi.net	CHN	12
12	Others	—	26
—	Total	—	338

5.3. Permission Set Split Effect Experiment

(1) Permission Set Split Effect Experiment's Setup. In this part, VirusTotal platform is used as an effective detection tool for collaborative application group. The platform is a multiengine file scanning tool created by Sistemas in 2004 [21]. And VirusTotal detects uploaded files through multiple security engines to determine whether there is malicious behavior. Some of the antivirus engines used by VirusTotal are listed in Table 4.

This paper chooses 10 Apps to be split from the malicious App set from Table 5 as samples. These samples all have redundancy of permission, and they apply multiple key information read permissions and sending class permissions. The sending class permissions, read class permissions, and VirusTotal detection rates of samples are listed in Tables 6, 7, and 8, respectively. "▲" indicates the samples apply for the

Table 6: Malicious sample's permission attributes for the transfer category.

Number of Sample	MD5 Value of Sample	Internet	SEND_SMS	CALL_PHONE
Mal_1	744c9f9ef5a3ad2559174523f1fd664d	▲	▲	▲
Mal_2	844bc220827f50539c67d09c3998a0da	▲	▲	■
Mal_3	899c92f0db1ec69e091795f4ddd251df	▲	▲	▲
Mal_4	4914c06560cdc3dfaca7c81eea9a33eb	▲	■	■
Mal_5	5192ad05597e7a148f642be43f6441f6	▲	■	▲
Mal_6	5895bcd066abf6100a37a25c0c1290a5	▲	▲	■
Mal_7	8947eae5c65df02d9c538b12ddaf636f	▲	■	▲
Mal_8	2908873c8ab99faa94ffe596499bd8f9	▲	■	▲
Mal_9	4884112ac7e599bd4dc20ccc91ce870c	▲	▲	■
Mal_10	375151412aff0b21d72207f08665d16d	▲	▲	▲

Table 7: Malicious sample's permission attributes for the read category.

Number of Sample	MD5 Value of Sample	Fine Location	Device State	Contacts	SMS content	WiFi State	Call Log
Mal_1	744c9f9ef5a3ad2559174523f1fd664d	▲	▲	■	▲	▲	▲
Mal_2	844bc220827f50539c67d09c3998a0da	▲	▲	▲	▲	■	▲
Mal_3	899c92f0db1ec69e091795f4ddd251df	▲	▲	■	▲	▲	▲
Mal_4	4914c06560cdc3dfaca7c81eea9a33eb	■	▲	▲	▲	▲	▲
Mal_5	5192ad05597e7a148f642be43f6441f6	▲	▲	▲	▲	▲	▲
Mal_6	5895bcd066abf6100a37a25c0c1290a5	▲	▲	▲	■	▲	▲
Mal_7	8947eae5c65df02d9c538b12ddaf636f	■	▲	▲	▲	■	■
Mal_8	2908873c8ab99faa94ffe596499bd8f9	▲	▲	▲	■	▲	■
Mal_9	4884112ac7e599bd4dc20ccc91ce870c	■	▲	▲	▲	▲	▲
Mal_10	375151412aff0b21d72207f08665d16d	▲	▲	▲	▲	■	▲

Table 8: Detection results of samples in VirusTotal platform.

Number of Sample	MD5 Value of Sample	VirusTotal Detection Result	VirusTotal Alarm Rate
Mal_1	744c9f9ef5a3ad2559174523f1fd664d	33/58	52.34%
Mal_2	844bc220827f50539c67d09c3998a0da	33/57	57.89%
Mal_3	899c92f0db1ec69e091795f4ddd251df	33/55	60.00%
Mal_4	4914c06560cdc3dfaca7c81eea9a33eb	37/57	64.91%
Mal_5	5192ad05597e7a148f642be43f6441f6	49/62	79.03%
Mal_6	5895bcd066abf6100a37a25c0c1290a5	49/62	79.03%
Mal_7	8947eae5c65df02d9c538b12ddaf636f	45/62	72.58%
Mal_8	2908873c8ab99faa94ffe596499bd8f9	32/54	59.26%
Mal_9	4884112ac7e599bd4dc20ccc91ce870c	32/57	56.14%
Mal_10	375151412aff0b21d72207f08665d16d	42/56	75.00%

permission and "■" indicates the samples do not apply for the permission. MD5 values for the corresponding samples on the VirusTotal platform are also listed.

Hydra-Bite splits the permission set according to whether there is redundancy or whether it is classified or not and uses "_NE_NTS, _E_NTS, _NE_TS, _E_TS" markers in the suffix. The meaning is shown in Table 9.

(2) Permission Set Split Effect Experiment's Results and Discussion. The experimental results of cooperative application group's antikilling performance evaluation are shown in Figure 11. The detection result in Figure 11 refers to the average alarm rate of Apps in the application group. The results of each sample are marked with different symbols and colors. Four types of results are separated from the

Table 9: The meaning of the suffix of the split result.

Serial Number	Marker	Meaning
1	_NE_NTS	**No** Elimination of Redundancy & **No** Split by Taxonomy
2	_E_NTS	Elimination of Redundancy & **No** Split by Taxonomy
3	_NE_TS	**No** Elimination of Redundancy & Split by Taxonomy
4	_E_TS	Elimination of Redundancy & Split by Taxonomy

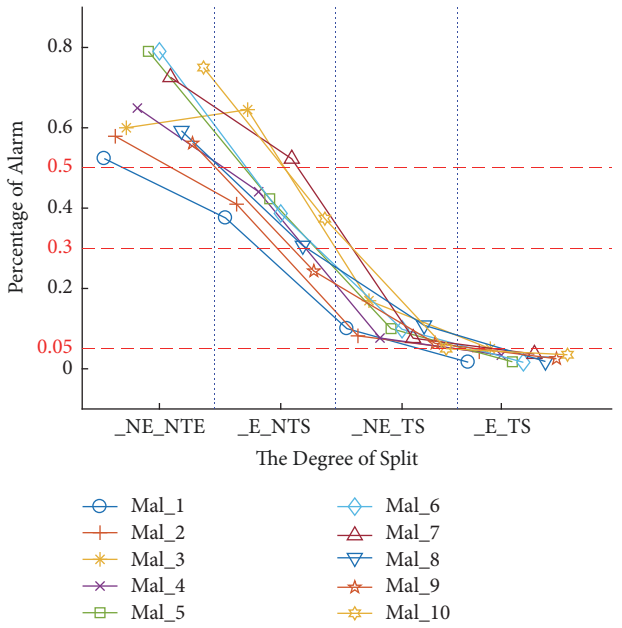

Figure 11: Alarm rate of different degree of resolution for samples.

blue vertical dotted lines. The red dotted line represents the threshold of different split methods. The following can be seen:

(1) The permission set is separately redundant or split by taxonomy, which can improve the antikilling performance of cooperative app group.

(2) The permission set is redundant and split by taxonomy, which can get the best antikilling performance of cooperative app group.

(3) The permission set is separately split by taxonomy which will enhance antikilling performance more than the result produced by being separately redundant.

The reason is that the antivirus engines are sensitive to potential information disclosure. Through simple elimination of permission set's redundancy, individual App within the cooperative application group reconstructed by HydraBite, there is still a potential risk of information disclosure for the Apps.

5.4. Antikilling Performance Experiment

(1) Antikilling Performance Experiment's Setup. In the antikilling performance experiment, our experimental samples are divided into 6 parts:

(1) Malware samples are the same as the contents of Tables 6–8.

(2) DroidBench samples are randomly selected from the DroidBench test set, and a microbenchmark suite is proposed and has been described in detail in the literature [9].

(3) Samples without taint clean and transmission directly: the sample set is shown in the 1st category, Table 10. After reading the key information, the samples do not clean the taint tagged by security methods. They send information to the external device directly through the corresponding external transmission permission. There is not collaboration between this kind of samples, so the column IPC is filled with "■."

(4) Samples with taint clean and transmission directly: the sample set is shown in the 2nd category, Table 10, adding a taint cleaning module to the previous samples. Three covert channels of volume, system log, and screen brightness are used to clean the taint. They send information to the external device directly with no IPC between them.

(5) Cooperative application samples without taint cleaning. The sample set is shown in the 3rd category, Table 10. In 1st category, the samples' permission sets are split and refactored to this sample set. The App in the group communicates directly with Intent. "_R" suffix represents an application that owns information read permissions. "_S" suffix represents an application that owns information send permissions.

(6) Cooperative application samples with taint cleaning: the sample set is shown in the 4th category, Table 10, adding a taint cleaning module to the previous samples.

Experimental environment is set as follows:

(1) *Android 6.0 Device*. The experimental sample sets are installed in the device described in Section 5.2 (1) to test whether the samples can be installed smoothly.

(2) *VirusTotal Platform*. The VirusTotal platform is mentioned in Table 6 and the literature [24]. This section uses VirusTotal to test the samples' antivirus performance.

(3) *Androguard [27]*. Carry out "permission-API" detection, and report dangerous samples of API calling combination. The Python version that builds Androguard is 2.7.10, and the mapping file is provided by PScout [13]. The running environment of Androguard: Androguard runs in Win 10 Professional Edition with the register width of 64 bits. The necessary Python, JDK, and SDK version are 2.7.10, 1.8, and 18, respectively.

(4) *FlowDroid [9]*. This tool's running environment is listed below. The system environment for the tool is Win 10 Professional Edition, with 64 bits' register width. In the system, the JDK version and Android API are 1.8. Besides, some of the official FlowDroid Support packages are unavailable

Table 10: 3–6 sample sets' attribute.

Serial Number of Type	Type of Sample	Name of Sample	Target Information	IPC Transmission Mode	Sending Mode
1	Non-Cooperative Transfer Case without Taint Cleaned	Capture1	IMEI Number	■	SMS Message
		Capture2	Location	■	Internet
		Capture3	Contact	■	SMS Message
2	Non-Cooperative Transfer Case with Taint Cleaned	SoundClean	IMEI Number	■	SMS Message
		LogClean	Location	■	Internet
		ScreenClean	Contact	■	SMS Message
3	Co_Apps without Taint Cleaned	NSound_R	IMEI Number	Intent	■
		Sound_S	■	■	SMS Message
		NLog_R	Location	Intent	■
		Log_S	■	■	Internet
		NScreen_R	Contact	Intent	■
		Screen_S	■	■	SMS Message
4	Co_Apps with Taint Cleaned	CSound_R	IMEI Number	Intent	■
		Sound_S	■	■	SMS Message
		CLog_R	Location	Intent	■
		Log_S	■	■	Internet
		CScreen_R	Contact	Intent	■
		Screen_S	■	■	SMS Message

due to an update or lack of resources. This paper changes all slf4j package provided officially into a 1.8.0 beta version.

(2) Antikilling Performance Experiment's Results and Discussions. To evaluate the surviving performance of Hydra-Bite in the real device, antivirus engine, "permission-API" mapping detection, and taint tracking detection, this section uses the above settings in Section 5.4 (1) to carry out experiments, the results are shown in Table 11. The meanings of each column in Table 11 are as follows.

"Source" column records the number of samples' information read permissions and APIs.

"Sink" column records the number of samples' transmission permissions and APIs.

"Android 6.0" column records samples and whether they triggered alarm in the installation experiment.

"VirusTotal" column records samples' alarm number on the VirusTotal platform and the hole number of participating engines.

"Androguard" and "FlowDroid" columns record experiment results by the two tools. The symbols' meanings that appear in Table 11 are described in the following.

"▲": Detect the insecure object and alert.

"▲": false positive, alert for an object that meets the security rules.

"★": Do not detect the insecure object.

"★": Detect the potential insecure object and do not alert, because there is no object to cooperate with it.

This section analyzes the experimental results in Table 11 as follows:

(1) *Malware Samples*. All these samples trigger alerts in the Android6.0 installation experiment. And their alarm rate in the VirusTotal platform is higher than other sample sets. Because of packers, obfuscation technology, Androguard and FlowDroid cannot analyze them.

(2) *DroidBench Samples*. With the obvious action of capturing key information, this sample set has a high alarm rate in VirusTotal platform and low successful installation rate in Android system.

(3) *Samples without Taint Clean and Transmission Directly*. The samples do not clean the taint tagged by security methods. And they send information to the external device directly. The above behavior triggers more alerts on the VirusTotal platform, with an average alarm rate of 29.70%. "Androguard" and "FlowDroid" also generate alarms for them.

(4) *Samples with Taint Clean and Transmission Directly*. Because the taint is cleaned, FlowDroid does not produce an alarm for it. The alarm rate triggered by this group of samples on the VirusTotal platform has been reduced, average alarm rate is 15.04%. And Androguard generates alarms for them because of no collaborative application group.

(5) *Cooperative Application Samples without Taint Cleaning*. Because the taint is not cleaned, FlowDroid produce an alarm for the information reading App.

(6) *Cooperative Application Samples with Taint Cleaning*. "Androguard" and "FlowDroid" do not generate alarms for this group of samples, because of the cooperative application and the cleaned taint. Moreover, the alarm rate caused by this set of samples is significantly reduced on the VirusTotal platform, and the average alarm rate is 5.85%.

It can be seen that, in the FlowDroid, the Source point of the send App are detected, but the FlowDroid does not alarm it. The reason is that it receives the information without

Table 11: Test results of anti-killing performance.

Sample Origin	Test Case	Source (Permission/ Sensitive API)	Sink (Permission/ Sensitive API)	Android 6.0 Warning (T/F)	VirusTotal [21] (Warnings/ Detector)	Androguard [22] (Total Permissions/ Sensitive APIs)	FlowDroid [9] (Source/Sink)
Malicious Sample (Table 5)	Mal_1	5/■	3/■	T	33/58	■	■
	Mal_2	5/■	2/■	T	33/57	■	■
	Mal_3	5/■	3/■	T	33/55	■	■
	Mal_4	5/■	1/■	T	37/57	■	■
	Mal_5	6/■	2/■	T	49/62	■	■
	Mal_6	5/■	2/■	T	49/62	■	■
	Mal_7	3/■	2/■	T	45/62	■	■
	Mal_8	4/■	2/■	T	32/54	■	■
	Mal_9	5/■	2/■	T	32/57	■	■
	Mal_10	5/■	3/■	T	42/56	■	■
DroidBench Test Case [9]	Merge1	1/1	1/1	T	27/54	4/▲ ▲	▲/▲
	DirectLeak1	1/1	1/1	T	27/55	4/▲ ▲	▲, ▲/▲
	ArrayAccess1	1/1	1/1	F	28/61	4/▲ ▲	▲, ▲/▲
	Button3	1/1	1/1	T	9/57	5/▲ ▲	▲/★
	ContentProvider1	1/1	1/1	T	25/55	4/▲ ▲	▲ ∗19, ▲/▲, ▲
	FieldSensitivity1	1/1	1/1	F	23/54	4/▲ ▲	▲, ▲/▲
	Loop1	1/1	1/1	T	33/59	4/▲ ▲	▲, ▲/▲
	ImplicitFlow1	1/1	0	F	12/57	2/▲ ★	▲ ▲/▲
	ActivityLifecycle2	1/1	1/1	T	32/62	4/▲ ▲	▲, ▲/▲
	Reflection1	1/1	1/1	T	28/55	4/▲ ▲	▲ ∗2, ▲/▲
	Echoer	1/1	1/1	T	17/62	3/▲ ★	▲/▲
	IntentSink2	1/1	1/1	T	30/61	2/▲ ▲	▲ ∗3, ▲/▲, ▲
	EventOrdering1	1/1	1/1	F	28/59	6/▲ ▲	▲, ▲/▲
	Executor1	1/1	1/1	T	21/58	4/▲ ★	▲/★
	JavaThread2	1/1	1/1	F	22/60	4/▲ ▲	▲, ▲/▲
	AsyncTask1	1/1	1/1	T	18/55	3/▲	▲ ∗2, ▲/▲
Non-Cooperative Transfer Case without Taint Cleaned (Table 10)	Capture1	1/1	1/1	F	14/59	2/▲ ▲	▲/▲
	Capture2	1/1	1/1	F	21/60	2/▲ ▲	▲/▲
	Capture3	1/1	1/1	F	17/56	2/▲ ▲	▲/▲
Non-Cooperative Transfer Case (Table 10)	SoundClean	1/1	1/1	F	10/62	4/▲ ▲	★/★
	LogClean	1/1	1/1	F	11/61	6/▲ ▲	★/★
	ScreenClean	1/1	1/1	F	13/62	5/▲ ▲	★/★
Co_Apps without Taint Cleaned (Table 10)	NSound_R	1/1	1/1	F	2/62	3/★	▲/▲
	Sound_S	1/1	1/1	F	5/60	2/★	★/★
	NLog_R	1/1	1/1	F	2/59	4/★	▲/▲
	Log_S	1/1	1/1	F	3/62	2/★	★/★
	NScreen_R	1/1	1/1	F	3/63	4/★	▲/▲
	Screen_S	1/1	1/1	F	6/57	2/★	★/★
Co_Apps with Taint Cleaned (Table 10)	CSound_R	1/1	1/1	F	2/62	3/★	★/★
	Sound_S	1/1	1/1	F	5/60	2/★	★/★
	CLog_R	1/1	1/1	F	2/59	4/★	★/★
	Log_S	1/1	1/1	F	3/62	2/★	★/★
	CScreen_R	1/1	1/1	F	3/63	4/★	★/★
	Screen_S	1/1	1/1	F	6/57	2/★	★/★

taint and sends it directly. Therefore Hydra-Bite can clean taint which is tagged by FlowDroid, it can resist detection tools based on "permission-API," and it has a high successful installation rate and a low VirusTotal alarm rate, and the results showed that Hydra-Bite method has enough threat to user privacy in antikilling performance.

6. Conclusion

In this paper, the key information disclosure through Hydra-Bite privacy leak path is researched. The purpose is to alert researchers to promote the progress of security work against collusion attacks and taint cleaning. The Hydra-Bite method is a malicious application variant that threatens user privacy in the context of application-scale operations. Hydra-Bite splits and reorganizes the traditional privacy stealing Trojans into a collaborative application group through the permission split module and uses the taint cleaning module to wash the taint tagged by the static taint tracking method on the communication entity which carrying the key information through the Android covert channel sends the cleaned key information to other devices through the collaborate sending module. The principle analysis and experimental results show that Hydra-Bite is less controlled to current security mechanisms in terms of performance and antikilling performance compared to traditional privacy stealing Trojans. Our next study will focus on improving existing static taint tracking mechanisms to tag key information with "more viscous" taints.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

The work presented in this paper is supported by the National Natural Science Foundation of China (nos. U1636219, 61602508, 61772549, U1736214, and 61572052), the National Key R&D Program of China (nos. 2016YFB0801303, 2016QY01W0105), Plan for Scientific Innovation Talent of Henan Province (no. 2018JR0018), and the Key Technologies R&D Program of Henan Province (no. 162102210032).

References

[1] M. Alazab, V. Monsamy, L. Batten, R. Tian, and P. Lantz, "Analysis of malicious and benign android applications," in *Proceedings of the 32nd IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW '12)*, pp. 608–616, June 2012.

[2] H. L. Thanh, "Analysis of malware families on android mobiles: detection characteristics recognizable by ordinary phone users and how to fix it," *Journal of Information Security*, vol. 4, no. 4, pp. 213–224, 2013.

[3] A. J. Alzahrani and A. A. Ghorbani, "SMS mobile botnet detection using a multi-agent system," in *Proceedings of the the 1st International Workshop on Agents and Cyber Security*, pp. 1–8, Paris, France, May 2014.

[4] A. Castillo C, "Android malware past, present, and future," White Paper of McAfee Mobile Security Working Group, 2011.

[5] R. Schlegel, K. Zhang, X. Zhou et al., "Soundcomber: a stealthy and context-aware sound Trojan for smartphones," in *Proceedings of the Network and Distributed System Symposium (NDSS '11)*, pp. 17–33, 2011.

[6] F. Zhao, W. Shi, Y. Gan, Z. Peng, and X. Luo, "A localization and tracking scheme for target gangs based on big data of Wi-Fi locations," *Cluster Computing*, vol. 3, pp. 1–12, 2018.

[7] W. Q. Shi, X. Luo, F. Zhao, Z. Peng, Q. Cheng, and Y. Gan, "Geolocating a WeChat user based on the relation between reported and actual distance," *International Journal of Distributed Sensor Networks*, vol. 4, no. 14, 2018.

[8] W. Y. Liu, X. Y. Luo, Y. M. Liu et al., "Localization algorithm of indoor Wi-Fi access points based on signal strength relative relationship and region division," *Computers, Materials & Continua*, vol. 55, no. 1, pp. 71–93, 2018.

[9] S. Arzt, S. Rasthofer, C. Fritz et al., "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[10] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of 16th ACM Conference on Computer and Communications Security*, pp. 235–245, ACM, November 2009.

[11] M. Nauman, S. Khan, and X. Zhang, "Apex: extending Android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS '10)*, pp. 328–332, Beijing, China, April 2010.

[12] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: user attention, comprehension, and behavior," in *Proceedings of the 8th Symposium on Usable Privacy and Security (SOUPS '12)*, Washington, DC, USA, July 2012.

[13] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: analyzing the Android permission specification," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS '12)*, pp. 217–228, ACM, October 2012.

[14] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys '11)*, pp. 239–252, July 2011.

[15] P. P. F. Chan, L. C. K. Hui, and S. M. Yiu, "DroidChecker: analyzing android applications for capability leak," in *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '12)*, pp. 125–136, April 2012.

[16] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting Android apps for component hijacking vulnerabilities," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS '12)*, pp. 229–240, October 2012.

[17] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek, "Covert: compositional analysis of Android inter-app permission leakage," *IEEE Transactions on Software Engineering*, vol. 41, no. 9, pp. 866–886, 2015.

[18] Y. Y. Ma, X. Y. Luo, X. Y. Li, Z. Bao, and Y. Zhang, "Selection of rich model steganalysis features based on decision rough set

α-positive region reduction," *IEEE Transactions on Circuits and Systems for Video Technology*, 2018.

[19] Y. Zhang, C. Qin, W. M. Zhang, F. Liu, and X. Luo, "On the fault-tolerant performance for a class of robust image steganography," *Signal Processing*, vol. 146, pp. 99–111, 2018.

[20] X. Y. Luo, X. F. Song, X. Y. Li et al., "Steganalysis of HUGO steganography based on parameter recognition of syndrome-trellis-codes," *Multimedia Tools and Applications*, vol. 75, no. 21, pp. 13557–13583, 2016.

[21] V. Total, "VirusTotal-free online virus," *Malware and URL Scanner*, vol. 15, no. 2, pp. 226–241, 2012.

[22] Androguard, 2013, https://github.com/androguard/androguard.

[23] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern Smartphones," in *Proceedings of the Proceeding of the 28th Annual Computer Security Applications Conference (ACSAC '12)*, pp. 51–60, New York, NY, USA, December 2012.

[24] E. Miluzzo, M. Jadliwala, S. Balakrishnan et al., "Tapprints: your finger taps have fingerprints," in *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, pp. 323–336, 2012.

[25] A. Maiti, M. Jadliwala, J. He et al., "Side-channel inference attacks on mobile keypads using smartwatches," https://arxiv.org/abs/1710.03656.

[26] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A. Sadeghi, and B. Shastry, "Practical and lightweight domain isolation on Android," in *Proceedings of the the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, p. 51, Chicago, Ill, USA, October 2011.

[27] R. A. Kemmerer, "Shared resource matrix methodology: an approach to identifying storage and timing channels," *ACM Transactions on Computer Systems*, vol. 1, no. 3, pp. 256–277, 1983.

[28] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 49–61, January 1995.