Visual Analysis of Android Malware Behavior Profile Based on $PMCG_{droid}$: A Pruned Lightweight APP Call Graph

Yan Zhang[1,2,3], Gui Peng[1,2,3(✉)], Lu Yang[2,4], Yazhe Wang[1,2], Minghui Tian[2,4], Jianxing Hu[1,2,3], Liming Wang[2], and Chen Song[2]

[1] State Key Laboratory of Information Security, Beijing, China
[2] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{zhangyan,penggui,wangyazhe,hujianxing,wangliming,songchen}@iie.ac.cn
[3] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[4] Beijing JiaoTong University, Beijing, China
{13283023,15125043}@bjtu.edu.cn

Abstract. In recent years, there is a sharp increasing in the number of malicious APPs on the Android platform, so how to identify new type of Android malware and its malicious behaviors has been a hot research topic in the security community. This paper presents a visualization framework to help security analysts precisely distinguish malicious profiles of APPs. By labeling target nodes, adding implicit call edges, pruning harmless branches, and a few other operations, we generate a new kind of call graph: $PMCG_{droid}$. This graph not only has a sharp decrease in size comparing to the original APP call graph but also preserves the malicious core of malware well. Based on $PMCG_{droid}$, visual interfaces are designed to assist users in checking the malicious behavior profile of samples with rich user interactive operations. We study real world samples to prove the usability and efficiency of our approach.

Keywords: Android malware analysis · Malware visualization
Machine learning · Assisted manual analysis

1 Introduction

Currently Android malwares are widespread and uncurbed. G DATA security experts have discovered 9 million Android malware samples from 2012 to the first quarter of 2017 [34]. Meanwhile, new instances are gathered daily, and variants of existing families appear quickly too. Although researchers have applied multifarious automatic Android malware analysis techniques [8–13,18] to confront this serious security challenge, manual detection methods are still widely needed, for example, to identify, correct, and disambiguate intermediate results of automatic analysis tools [24], or to understand the malwares and their nature [21].

X. Lin et al. (Eds.): SecureComm 2017, LNICST 238, pp. 449–468, 2018.
https://doi.org/10.1007/978-3-319-78813-5_23

In consideration of the complexity of Android APP, experts bear a huge burden of work if only manual work used to analyze the samples. Therefore, it is urgent to explore semi-automated visualization analysis approaches to help analysts in reducing heavy workload. Visualization analysis tools take mass of basic trivial analysis works for human and show the machine analytic results in a visual way. Then the security staff can quickly grasp key information under the help of the visual displayed graphs or figures and some interactions, use professional knowledge to deal with something that machine cannot handle, and make precise judgement efficiently.

Since, there are many essential differences between Android applications and traditional personal computer applications. A mass of existing PC application visualization tools [25–31] cannot be directly applied to Android applications [7,33] and the exploitation of visualization for Android application has just started, practical tools are scarce [19,22]. Meanwhile, the existing individual malware visualization analysis methods for PC or for Android platforms rarely concern the visualization of the malicious code logic structure. However, code logic structure usually implies the whole picture of the malicious behaviors. It is worth to be processed and provided to the experts for analysis assistance.

The objective of this paper is providing a malicious code structure and malicious behavior profile visualization analysis method of Android malware. By labeling and appending nodes, adding implicit call edges, pruning harmless branches, and some other operations, we generate a new kind of Android APP call graph: $PMCG_{droid}$. $PMCG_{droid}$ aims to show the targeted risky code distribution and correlations inside an Android APP, helps users to figure out the malicious behaviors set.

We made the following contributions to the visual detection of Android malware in this paper:

(1) Advance a brand new graph $PMCG_{droid}$ which is a pruned lightweight Android APP call graph. Compared to the traditional call graph, $PMCG_{droid}$ not only narrows down the manually inspection scope of a sample but also reserves the core malicious profile effectively.
(2) Design visualization interfaces to display the $PMCG_{droid}$ graph of samples. In the interfaces, not only risky code can be figured out from the graph, but also the complete triggering chain and code logical combinations of such risky points can be revealed visually. Hence the whole malicious behavior profile and structure is clear to the users.
(3) Provide automated methods and user interactivity (implicit edges appending, convergence point analysis and subgraph generations etc.) to help the analysts quickly focus on most suspicious behaviors and explore code details to make accurate judgements.

We use a case study to illustrate the effectiveness and feasibility of our work. Through the analysis of a large number of malicious samples from the real world by using our method, we have a lot of interesting findings, which will be shown in this paper too.

2 $PMCG_{droid}$ Generation

2.1 Target Node Labeling

Our visualization analysis framework focuses on how to visually check the key parts and malicious behavior structure inside the APP's method call graph. Given a call graph of a sample APP, the most important inspection target nodes of it are method nodes that contain risky API calls.

Table 1. Risky APIS and their types

API	Occurrence frequency	Type
java.net.URL.openConnection	25856	Sink
android.telephony.TelephonyManager.getDeviceId	10478	Source
dalvik.system.DexClassLoader	2957	Suspicious
android.telephony.TelephonyManager.getLine1Number	4279	Source
android.telephony.SmsManager.sendTextMessage	4087	Sink
android.location.LocationManager.getLastKnownLocation	3955	Source

The risky APIs we concerned about are from the following 3 sets: (1) The key APIs which are restricted by the Android permission mechanism. (2) The sensitive APIs used by Arp et al. [8] in their machine learning features. (3) A set of malicious behavior most relevant risky APIs identified by us, based on the manual analysis of 300 popular malicious samples.

The number of target APIs directly affects the accuracy and complexity of the experimental results. The more APIs are detected, the more comprehensive the $PMCG_{droid}$ is generated, then the result will be more accurate. However, the cost is to increase the scale and complexity of $PMCG_{droid}$. In order to achieve a balance between the accuracy and complexity, we do a statistics over the public Drebin Android Malware database [39] to study these APIs' occurrence frequency in malware samples. Based on the frequency we identified 130 APIs as the target set finally, to achieve maximum accuracy while reducing the complexity of manual analysis.

Furthermore, according to the threat nature of these dangerous APIs, they are divided into three types: Source, Sink, and Suspicious. "Source" refers to those APIs that can access sensitive information in Android devices, for example, the APIs to read SMS, contact information, GPS Location etc. The APIs that may output sensitive information are "Sink", for example, the APIs to send out information by email, SMS, Bluetooth, network, or write information via SQL database, SharedPreference, file etc. The rest of the APIs are also dangerous and can be classified as "Suspicious". For example, DexClassLoader APIs may be used to execute code which is not installed as part of the application. The Table 1 lists six sample risky APIs and the category to which they belong.

Correspondingly, we label the nodes that contain the source, sink, or suspicious type APIs in their code as API-Source, API-Sink, and API-Suspicious

node. Some nodes may contain multiple labels at the same time, because they call different types of APIs in their method code.

Except risky API tagged node, there is another kind of nodes which are worth being concerned about. They are "third-party library" nodes that represent methods from some popular third party libraries imported by APP in programming phase, such as from AdMob [44], umeng [42], google map [43], and so on. The idea behind is to ascertain where the APP's risky behaviors inside come from, the APP itself or some third-party libraries. Currently, the third-party libraries detected by us are mainly advertising libraries. We use the following methods to identify third-party library code nodes. We collect the Software Development Kits of popular third-party libraries, record the key package names, class names, and method names in these libraries. Then the package name, class name and method name of every method node of APPs will be compared with the information recorded above to determine whether the node belongs to some third-party library or not.

2.2 Implicit Edge Generation

Generating an accurate call graph is crucial for static analysis. Special mechanisms for Android programs, such as Inter-Component Communication (ICC), component lifecycle, multithreading, etc., can cause discontinuities in the application method call flow. The existing Android or Java call graph generation tools cannot fill these vacancies. Thus, on the basis of the traditional call graph, we further add the missing method call flow and build a more complete call chain to show the whole picture of malicious behaviors. We call the supplementary edges as implicit edges. There are four kinds of implicit edges considered in $PMCG_{droid}$:

A. ICC Type: Android applications are composed of components. The communication between components utilizes explicit or implicit Intent to perform. Explicit Intent specifies the component to start by name (fully qualified class name), hence it connects the caller to the receiver component directly according to the specified component name; the implicit Intent passes the information of the caller component to those components whose Intent Filter declarations match the implicit Intent's Action, Category, and Data attribute content.

In order to fill the function call edge missing from ICC, we collect information about all the components in APP and their Intent Filter contents, Intent delivery methods and parameters by utilizing the IC3 [2] tool. Then, based on the metadata obtained, we simulated the Android system to match the Intents, both implicit and explicit, discover the call edges between components.

In particular, for the difference of *StartActivityForresult* [13], we add the call edge from the *setResult* of the callee component to the *onActivityResult* of the caller component.

B. Lifecycle Callback Type: Implicit call edges associated with the Activity/Service component lifecycle. Each Activity/Service component in Android has a full lifecycle, which contains different lifecycle callback methods such as

onStart, *onResume* etc [45,46]. The Android framework implicitly calls these methods to convert the component's lifecycle state, such as calling *onStart* to start the component and call *onPause* to pause the component. These lifecycle callback methods are not directly connected in the code, and their calling processes are completely dependent on the Android framework, so the call chains associated with these lifecycle method calls are also missing.

We check the lifecycle transition process of Activity and Service, consider each state transition process as an implicit call edge, add to our $PMCG_{droid}$ graph, so that the code executed in the whole component lifecycle can maintain coherence in our graphs.

In the life cycle of the activity, we currently ignored three kinds of state transition, *onPause*→ *onResume*, *onStop* →*onRestart*, and *onStop* →*onCreate*. In another words, we won't add implicit edges for these three state transitions. Although this ignorance will cause a small part of continuity lose, it helps us reduce many loops.

C. Thread Type: Usually an Android system service creates an auxiliary thread by two ways: Runnable and Handler [38]. In these two mechanisms there also exists the control chain missing phenomenon. For example the *start* method in the Runnable mechanism is used to start a thread, but the thread does not run immediately until the *run* method of the new thread is executed when system recourses are distributed to it. As for Handler mechanism, *sendmessage* method is used to send a message to *handlemessage* for processing, but the call chain from *sendmessage* to *handlemessage* does not exist naturally because the message passed through the framework.

D. Logic Connection Type: This type of implicit call side is primarily related to intent delivery. We find that the intention of the transfer exists in some method pairs, such as broadcast receivers and their registration methods. Broadcast receiver is actually triggered by the broadcast sender via Intent, this relationship is included in ICC Type already. However, broadcast receiver is under the control of broadcast register. The Register decides which broadcast the broadcast receiver should be registered to. There is an intension transmission. In order to complete the malicious behavior call chain, we add the call edge for the method of passing this intention.

2.3 Branch Pruning

The graphical scale of call graph of an Android APP is usually too huge to artificial analysis. We decompile 1000 APP's package files (APK files) whose size distribution range from 27 kilobytes to 32 megabytes, and calculated their method numbers one by one. Our statistics shows every 5 megabytes APK file contains 3702 functions in average. Hence, visually checking the original call graph is a heavy workload. It is necessary to narrow down the node inspection scope and reduce unnecessary detections for security analysts.

As we already discussed in Sect. 2.1, nodes containing risky APIs are considered most relevant to the malicious behaviors, hence we only need to focus on

the nodes and edges related to target nodes. Our proposal is to trim all nodes which have no directed path leading to any target risky API nodes. To make this description more precise, we define a Boolean function Dpath to indicate whether there is a directed path existing from one node to another node.

Definition 1. *For any directed graph (N,E), N is the node set while E is edge set, Dpath is a function defined over N, Dpath* : $N \times N \rightarrow \{1, 0\}$, *for* $\forall\ n_1, n_2 \in N$:

$$Dpath(n_1, n_2) = \begin{cases} 1, if\ n_1 = n_2\ or\ \exists\ \{e_1, e_2, \ldots, e_n\} \subset E\ s.t. \\ \quad source(e_1) = n_1,\ target(e_n) = n_2, \\ \quad source(e_{j+1}) = target(e_j)\ for\ j \geq 1; \\ 0, otherwise \end{cases}$$

Here source (e) is the start point of the directed edge e, while target (e) means the targeted node.

We define all the nodes to be removed from as set TN, while all the edges to be cut as set TE:

Definition 2. *Given a directed call graph:* $CG = (N_m, E_m)$, *TN is the greatest subset of* N_m, *s.t. for* $\forall n_i \in TN$, *for* $\forall n_j \in N_m$ *where* $label(n_j) \in \{API - Source, API - Sink, API - Suspicious\}$, $Dpath(n_i, n_j) = 0$.

Definition 3. *Given a directed call graph:* $CG = (N_m, E_m)$ *and TN, TE is the greatest subset of* E_m, *s.t. for* $\forall e_j \in TE, \exists n_i \in TN$ *s.t.* $target(e_j) = n_i | source(e_j) = n_i$.

2.4 Convergence Point Discovery

We detect three kinds of Convergence Point (CPoint) to help analysis potential information leak in an APP automatically.

Independent CPoint: such CPoint node directly or indirectly calls an API-source node and an API-sink node concurrently. More strictly speaking, it should be the nearest CPoint for at least one pair of (API-Source, API-Sink). The CPoint may call API-Source to get sensitive info and send out by API-Sink node.

API-Sink Node as CPoint: if one node containing data sending code directly or indirectly calls a API-Source node, it may get the sensitive data first from the API-Source node, then send out by itself.

API-Source Node as CPoint: if one node containing data getting code directly or indirectly calls a API-Sink node, it may send out the sensitive data collected by the get information API of itself.

2.5 Splitting Shadow Node

It is a common phenomenon that nodes may be tagged with a variety of labels. Actually we want to set every kind of node an independent color to assist users'

analysis in our visualization tool. So in the last step of generating $PMCG_{droid}$, we introduce the concept of shadow nodes to ensure that each node has only one label. If a node contains N labels, the node is divided into N nodes, by keeping one main node in the original call chains of the method and adding $N-1$ shadow nodes which have and only have bi-directional edges with the main node. That means, the main node maintains the call relationships with the other nodes in the original call chains, while shadow nodes just represent $N-1$ labels of the main node.

2.6 $PMCG_{droid}$ Definition

By labeling and appending nodes, adding implicit call edges, pruning harmless branches and shadow node splitting, we generate a new kind of Android APP call graph: $PMCG_{droid}$. The $PMCG_{droid}$ graph is defined by a quintuples = (N_p, E_p, Label, f_l, f_c), where N_p is the set of nodes in the graph and E_p = $\{(n_i, n_j)\}$ is the set of edges/connections between nodes. The adjacency matrix _ij indicates that an explicit or implicit call exists from n_i to n_j (_ij = 1) or that the call is absent (_ij = 0).

In our default $PMCG_{droid}$ version, *Label* is a string set with five values which is "API-Sink", "API-Source", "API-Suspicious", "third-party library", and "normal", because in the $PMCG_{droid}$, there are five types of nodes, API-Sink, API-Source, API-Suspicious, third-party library, and normal (any nodes that are not tagged to the first four types are normal nodes). *Label* is used to label the nodes with function f_l. That is to say, f_l maps elements in the set N_p to a value in set *Label*. f_c is a Boolean function defined over Set N_p, it maps every node to Boolean value 0 or 1. It indicates whether a node is a convergence point or not.

3 Visualization

For helping manual analysis, a set of interfaces are built to present the $PMCG_{droid}$.

3.1 Visualization Encode

In order to show the relationship between these nodes in $PMCG_{droid}$, we distinguish the nodes in various colors and sizes, while arrows in different colors and types representing different types of calling. As we can see in Fig. 1(b), we use green circles to represent the third-party libraries and gray circles to represent the normal nodes. Besides, we mark the API-sink and API-source nodes with red and brownish red colors respectively. We use yellow circles to represent API-Suspicious nodes. Black one-way arrows and red one-way arrows stand for explicit call edges and implicit call edges individually, and blue bidirectional arrows represent shadow link nodes. For further pushing convergence points forward, we set them two times the sizes of the normal ones.

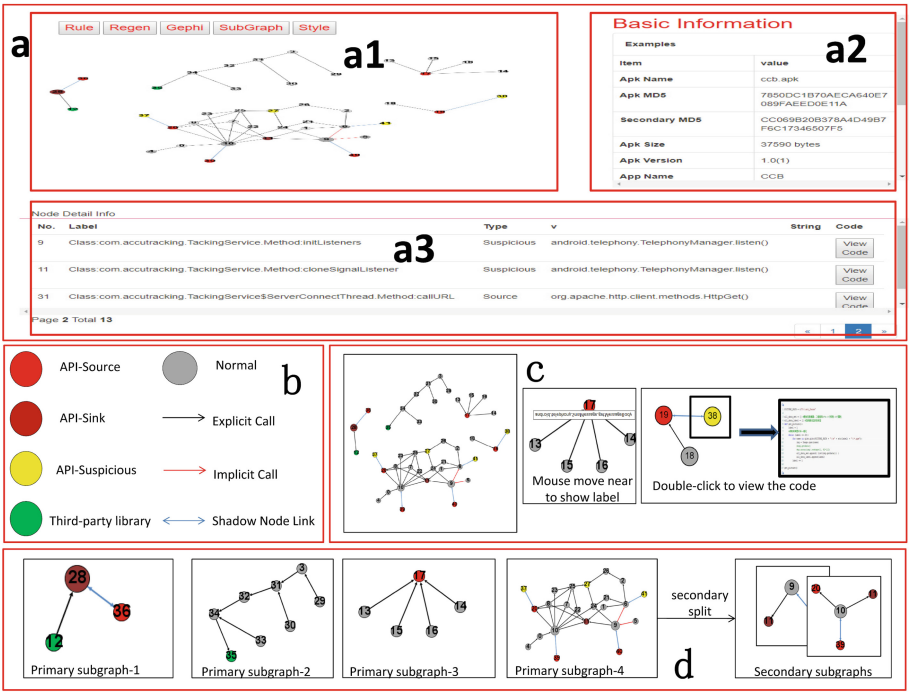


Fig. 1. System interface (a, c and d) and visualization encode (b). Main workspace view (a), including $PMCG_{droid}$ panorama with default force-directed layout (a1), APP basic information (a2) and node detail information overview (a3). The interaction methods of a1 (c) including mouse move, single, and double click. Primary and secondary subgraph view (d).

3.2 Integrated Visualization Interfaces

Based on $PMCG_{droid}$, we develop integrated visualization web interfaces (as Fig. 1 shown) to help users to inspect malicious behaviors from Android APPs.

Users can upload their own APP to check the APP's $PMCG_{droid}$ graph in the interface. The result is presented in the workspace area of the interface as shown in Fig. 1(a1). Figure 1(a3) shows the details of nodes including method name, class name and tag information. Inside the workspace area of Fig. 1(a1), tag details of every node will be shown when the mouse is moved near to it. When double clicking on the node, there would be a message popping up and showing the source code of the node.

In order to help visualization analysis, it is necessary to annotate the key information for each node, so we have defined four kinds of tag information: the first one is correlated to implicit call edge. If the node is a caller correlated to an implicit call edge, the tag of the node shows the method name the node call and corresponding parameters. If the node is a callee correlated to an implicit call edge, the tag shows its own method name and class name. Specially, if the

callee is triggered by implicit Intent, its tag also shows the value of the Intent Filter.

The second kind of tag information shows the third-party libraries the node belongs to if it is a third-party library node. Third, for API-Sink, API-Source, and API-Suspicious nodes, their tags show the sink APIs, source APIs and suspicious APIs they call respectively. Finally, for all kinds of nodes, some other information they contains could be risky, such as URL and telephone number, so the last type of tag shows constant string like this.

3.3 Subgraph

Although the scale of $PMCG_{droid}$ has been decreased greatly, when a malware contains numerous risky behaviors, the graph is still too complicated to analysis. Therefore, we further propose a risky behavior slice function to separate the $PMCG_{droid}$ graph into several subgraphs (Fig. 1(d)) for analysts to view.

There are two kinds of subgraphs as Fig. 1(d) shown. The first four graphs are the primary subgraphs of $PMCG_{droid}$. They are independent of each other and there are no edges between them. Analyst can only focus on a single primary subgraph rather than the entire $PMCG_{droid}$.

The graphs on the right side are the secondary subgraphs which only shows the risky paths around one single convergent point. The second subgraph is generated from each CPoint. For each CPoint, find all the API-Sink and API-Source nodes it can reach in the directed graph. The nodes and edges on the path from the CPoint to its reachable API-Sink or API-Source nodes make up a secondary subgraph relevant to this CPoint.

4 Case Study

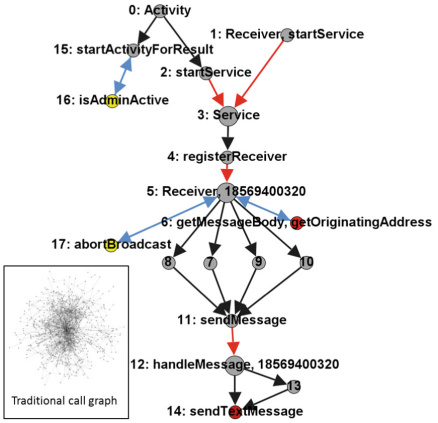
In this section, we demonstrate the effectiveness and feasibility of our interfaces with a case study. The case study discusses how to reveal the malicious behavior profile of a special malware sample in a public family. Then, we present some other findings based on our large scale analysis.

4.1 Reveal Malware's Malicious Behaviors

We randomly choose a sample from a popular malware family named Fakeinst. Then we found the description about Fakeinst malware family in f-secure website [40]. It says: "Fakeinst malware appear to be installers for other applications; when executed however, the malware send SMS messages to premium-rate numbers or services."

However, we still do not know exactly what malicious behaviors will be triggered by the APP and how. Now we open our visualization interface to see what its real behaviors are. By using our tool to generate the $PMCG_{droid}$, the $PMCG_{droid}$ and nodes' label information are shown in Fig. 2. The $PMCG_{droid}$ is much smaller than the traditional call graph in the lower left corner in size,

decreasing by 96.1%. To simplify the introduction, we removed a few nodes which are irrelevant to the malicious behaviors. Based on the Fig. 2, we start the research work.



No.	Node Label Information
16	Suspicious API: android.app.admin.DevicePolicyManager.isAdminActive
15	startActivityForResult: action= android.app.action.ADD_DEVICE_ADMIN
0	Activity: class name= com.msae.rebt.MainActivity Activity: action= android.intent.action.MAIN
2	startService: target name= com.msae.rebt.sgter
1	Receiver: class name= com.msae.rebt.BCRcer Receiver: action= android.intent.action.BOOT_COMPLETED startService: target name= com.msae.rebt.sgter
3	Service: class name= com.msae.rebt.sgter
4	registerReceiver: action= android.provider.Telephony.SMS_RECEIVED
5	Receiver: class name= com.msae.rebt.SgterMesReceiver Receiver: action= android.provider.Telephony.SMS_RECEIVED String: 18569400320
17	Suspicious API: android.content.BroadcastReceiver.PendingResult.abortBroadcast
6	Source API: android.telephony.SmsMessage.getMessageBody Source API: android.telephony.SmsMessage.getOriginatingAddress
11	sendMessage: class name= android.os.Handler
12	handleMessage: class name= com.msae.rebt.SgterMessageHandler String: 18569400320
14	Sink API: android.telephony.SmsManager.sendTextMessage

Fig. 2. Case 1 (package name: com.message.send, MD5: 4E850BF087512F14A7A EA84909982569)

We start from node 0 which is the entry node of the program. At first, we come to inspect the short call chain 0 → 15 → 16. Node 16 calls *android.app.admin.DevicePolicyManager.isAdminActive*. It determines whether the given administration component is currently active in the system. Node 15 calls *startActivity* with the Intent action value *android.app.action.ADD_DEVICE_ADMIN* to register the device manager. By checking the code we confirm that the APP will be registered as a device manager when it starts, which makes it difficult to uninstall the APP.

Next, we investigate the call chains: 0 → 2 → 3 → 4 → 5 → 10(7/8/9) → 11 → 12 → 14 and 5↔6, 5↔17. These call chains can be further divided into four stages: stage A: 0 → 2, stage B: 3 → 4, stage C: 5 → 10 → 11, 5↔6 and 5↔17, and stage D: 12 → 14. Every two adjacent stages are connected by an implicit call edge.

Stage A starts the service (node 3) in stage B by calling function *startService* with an explicit Intent. Stage B registers a broadcast receiver (node 5 in stage C) which monitors *android.provider.Telephony.SMS_RECEIVED*. This allows the APP to directly receive incoming SMS messages. Node 6, 17 are shadow nodes split from node 5. Based on the information of the nodes 6, 17 in the table on the right side of the Fig. 2, it can be seen that on the one hand, it gets the contents of the SMS message and the sender's mobile phone number; on the other hand, the node 5 aborts the current broadcast to prevent any other APPs from receiving the SMS message. Then, the stage C sends

the sensitive data to the stage D through the Handler mechanism. Finally, stage D sends the data to the telephone number "18569400320" via function *android.telephony.SmsManager.sendTextMessage*.

Furthermore, by checking the node 5 and 17's code, we find the node 5 also checks whether the received message is from a specific attacker. If the answer is positive, it will call 17 to block this message and do things according to the attacker's indication. This action is remotely controlled by the attacker.

Last but not least, in the upper right corner, node 1 is a broadcast receiver which monitors the phone's boot broadcast *intent.action.BOOT_COMPLETED*. According to the call chain $1 \rightarrow 3$, node 1 also starts the service node 3. So the APP will start the malicious service when phone boots up automatically.

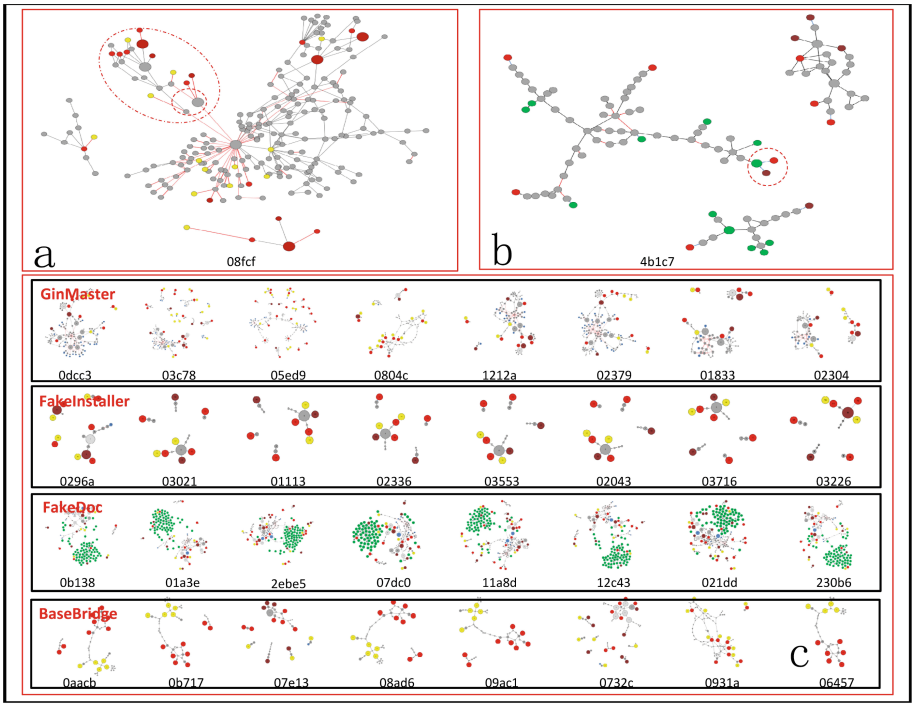


Fig. 3. Other interesting findings. (a) Weak connection structures imply repackage possibilities. (b) It is easy to distinguish which risk is induced by third-party libraries. (c) $PMCG_{droid}$ graphs resemble each other in same family, and differ between different families.

4.2 Other Findings

Besides the abilities above, we analyze a large number of samples in the virus database by using our method, we also find out some interesting phenomenon that could be considered as visual signal tips to help the experts with their analysis.

The first tip is when the connection between two complicated areas is weak and only built by few of nodes and edges, there is a possibility of repackaging. For example, Fig. 3(a) shows the $PMCG_{droid}$ of a confirmed repackaged APP. The malicious methods which are inserted into the original APP mainly concentrate in the red circle. This malicious part connects to the original code only through two nodes. This kind of connection is apparently a "weak" connection. The second one is when a API-Suspicious node and a third-party library node appear in pair, it means that the risk is introduced by third-party library (as Fig. 3(b) shown). And the most interesting one is this: for parts of malware families in Drebin [39] and Malgenome database [37], we found their $PMCG_{droid}$ graphs resemble each other in same family and differ with other families quite a lot. This implies that users may visually compare newly emerged malicious samples with existing samples to simply identify and classify them for these families. For example, in Fig. 3(c), we list four of such kind of families from Drebin database: GinMaster, FakeInstaller, FakeDoc, and BaseBridge. The finding makes us believe that we can further our work to use $PMCG_{droid}$ as an effective visual feature for malware family identification.

5 Evaluation of the Tailored Malicious Profile

In this section, we evaluate the performance of $PMCG_{droid}$ as a malicious profile tailored from the original APK. We conducted two experiments to check its following capabilities comparing to the original call graph: size sharply decreased and malicious core reserved.

Our data set consists of 4910 malware (M-set) and 4979 benign software (N-set). Among them, the M-Set comes from the previously mentioned Drebin Android malware set, while the benign APPs in N-set are collected from Google Play. All applications in N-set were submitted and detected by VirusTotal [41] before April 1, 2017, and no virus was reported by any Antivirus engine in VirusTotal. Based on this dataset, we conduct the following experiments.

5.1 Scale Reduction Experiments

In order to prove that the $PMCG_{droid}$ graph can effectively reduce the size of APP's call graph, we use 173 malware families in M-set, and pick 100 benign APPs from N-set as a benign family. Then we generate call graphs and $PMCG_{droid}$ graphs for all APPs in these families. After that, we do a statistics over the scale of them, calculate the average node and edge difference in number between the two kinds of graphs of each family.

From the Table 2, we can see that for malware, the number of nodes in $PMCG_{droid}$ is reduced by 94.4% from the number of nodes in the traditional call graph on average, and the number of edges decreases by 96.3%. That is to say, the $PMCG_{droid}$ graphs are not bigger than 5.6% of the original call graphs usually. The number of nodes in benign family is down by 92.0% and edges are down by 94% in average. We also show the top 3 and last 3 node and edge

number difference of malware families in the Table 2. For example, the family Gasms's difference is (97.0%, 97.9%), while the former number stand for node difference and the later one stand for edge difference.

Among them, the family Gasms achieves the highest node average reduce proportion of 97.0%, while the lowest family CellShark also reached 69.0%. For edges, the highest decreasing proportion reaches 98.0%, the lowest is 75.7%. Hence, $PMCG_{droid}$ graph can greatly reduce the scale of call graph.

5.2 Malicious Core Reservation Experiments

In order to verify that the $PMCG_{droid}$ still retains the core of malicious behavior in the software, we conducted a machine learning experiment. In this experiment, we extract features from $PMCG_{droid}$ graphs to see if they can be used to automatically distinguish between malware and benign applications. The Table 3 shows all the feature sets we extracted, where F1 and F2 represent the total number of nodes and edges in the $PMCG_{droid}$ respectively. F3 represents the diameter of the $PMCG_{droid}$ graph G, that is, the length of the longest call chain.

F4 is a set of features that represent the number of nodes per kind of *Label*. F5 represents the average of the degrees of each label type of node. The degree of the node is defined as the number of other nodes connected to the node. In the directed graph, the degree of the node is divided into indegree and outdegree. Outdegree refers to the number of edges pointing from the node to other nodes, and indegree refers to the number of edges pointing from the other nodes to the node. Correspondingly, F6 and F7 represent the average of outdegree and indegree of each type of node, respectively.

F8 represents the average reversal ripple degree of all nodes. In the directed unweighted graph, the number of all nodes that can be reached in the reverse direction from the node V is called the reversal ripple degree of the node V.

The F9 feature set represents the number of occurrences of the 130 risky APIs we selected in the nodes of the $PMCG_{droid}$ graph. Considering that most of the risky APIs need to apply for specific permissions can we use, we will apply the permissions as a feature set F10.

Table 2. The top/last 3, benign and average difference

	1	2	3		1	2	3	Average
Node top 3	Gasams (97.0%, 97.9%)	Fakeview (96.8%, 97.8%)	Generic (96.6%, 97.7%)	Node last 3	Mobilespy (71.2%, 81.3%)	Flexispy (70.0%, 81.9%)	CellShark (69.0%, 77.5%)	Benign (92.0%, 94%)
Edge top 3	Jifake (95.3%, 98.0%)	Gasms (97.0%, 97.9%)	GlodEagl (95.3%, 97.9%)	Edge last 3	CgFinder (76.4%, 78.8%)	CellShark (69.0%, 77.4%)	FakePlayer (76.2%, 75.7%)	Malware (94.4%, 96.3%)

Based on the feature set of F1-F10, We selected the random forest classification algorithm to classify. In the classification process, we use ten-fold cross-validation to obtain more accurate results. The result of this experiment is shown

Table 3. $PMCG_{droid}$ features and classification results

Features of $PMCG_{droid}$: G = (N_p, E_p, label, f_l, f_c)		
F1: $\|N_p\|$	F2: $\|E_p\|$	F3: Diameter(G)
F4: {$\|Node_{label}\|$}	F5: {AvgDegree($Node_{label}$)}	F6: {AvgOutdegree($Node_{label}$)}
F7: {AvgIndegree($Node_{label}$)}	F8: AvgRRDegree(G)	F9: {OccurenceNum(riskyAPI)}
F10: {AppliedPermission}		
Classification results		
Method (data set)	TPR	FPR
$PMCG_{droid}$ ($PMCG_{droid}$ data)	96.2%	1.1%
Drebin ($PMCG_{droid}$ data)	98.2%	2.6%
Drebin (Drebin data)	94%	1%

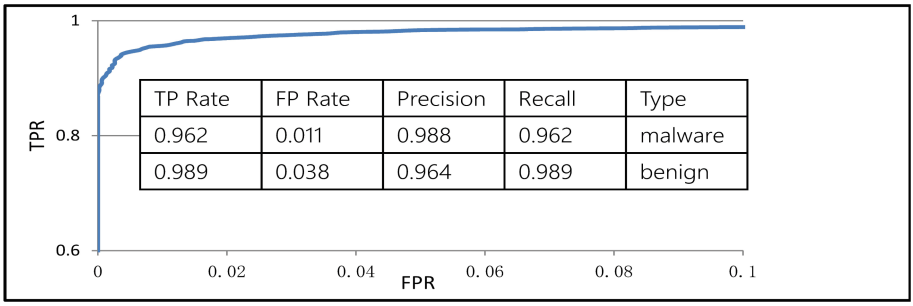


Fig. 4. ROC curve of $PMCG_{droid}$

in Fig. 4 as ROC curve. It detects 96.2% of the malware samples at a false-positive rate of 1.1%.

We compare the performance of the $PMCG_{droid}$ machine learning approach with related machine learning approaches for Android malware. So far, we know the best way to classify the results is Drebin, which in its own data set achieves TPR 94%, FPR 1% results, significantly outperforms the other approaches. Before it, approaches such as kirin [4], Peng et al. [17] provide a detection rate between 10%–50% at such false-positive rate. Since Drebin did not publish the benign application set it used, we used Drebin's feature extraction method and classification algorithm to classify our data set to compare our results. The experimental results are shown in Table 3. Drebin in our data set, still performed well, achieves TPR 98.2%, FPR 2.6%.

Hence, our classification results are very close to Drebin. Considering that our feature set dimension is only 1571, which is much lower than Drebin, we have reason to believe that though pruned large scale of nodes and edges, $PMCG_{droid}$ still gains a good performance in the automatic distinction between malicious and benign applications. This result confirms its retention of malicious core parts of malware.

6 Related Work

6.1 Android Malware Automatic Analysis

A large body of research has studied methods for analyzing and detecting Android malware. These methods can be roughly categorized into static analysis, dynamic analysis, and machine learning.

Static and dynamic methods intend to identify anomaly behaviors of suspicious samples by checking package code or runtime feature patterns. For example, Zhou et al. [3] extract permissions from APP packages, and then propose a permission-based behavioral footprint scheme to detect new samples of known Android malware families. SCanDroid [11] uses the data flow analysis method for static analysis and detects whether the data flow is consistent with the permissions automatically. AndroidLeaks [12] creates a call graph of an application's code and then perform a reachability analysis to determine if sensitive information may be sent over the network. Droidchecker [36] uses control flow search and stain analysis to automatically analyze possible sensitive data leaks from high permission store to low permission store. They and other static analysis approaches such as [2,23,38] all cannot tell what the whole malicious behavior picture is when they detected an abnormal signal.

Dynamic analysis approaches [5,18,33] monitor the behavior of applications at run-time. They usually suffer from a significant overhead. Among them, only DroidScope [33] is focused on revealing APP's malicious intent and inner workings by collecting detailed native and Dalvik instruction traces, profile API-level activity, and tracking information leakage. However, these data are too fragmental. Users need to use their own imagination to mosaic them into a full picture as shown in their case study.

As for recognizing malware automatically using learning methods, lots of methods have been proposed. Peng et al. [17] apply probabilistic learning methods to the permissions of applications for detecting malware. Puma [6] extracts static features based on permissions' usage, and evaluates the effectiveness of different classifiers, including random trees, random forests, naive Bayesian, and Bayesian networks. Similarly, the methods Crowdroid [16], Droid-Mat [15], MAST [14], Drebin [8], and AMDHunter [50] use features statically extracted from Android applications as there feature vectors. Although the classification effect is getting better and better, most of them cannot help explaining what makes a malware. Only Drebin can infer the risky combination of static properties. But that is still not very clear how the malicious behavior happens for every APP.

Among the existing automated analysis methods, some of the static analysis methods focus on the implicit call study such as [9–11,49,51]. Arzt et al. [9] provide a precise model of Android's lifecycle allows the analysis to properly handle callbacks invoked by the Android framework. Cao et al. [51] have done further research on detecting implicit control flow transitions through the Android framework. Reina et al. [10] dynamically observe interactions between the Android components and the underlying Linux system to reconstruct

higher-level behavior. Zhang et al. [49] also contributed to broken links connection when generating call graphs. The detail of these approaches provides lots of references and tools for us in matching the implicit edges. Fuchs et al. [11] provide another tool for reasoning about data flows in Android applications. It focuses on not only the-component but also the inter-APP data flow. We think it is possible for us to try connecting $PMCG_{droid}$ graphs of two APPs together for conspiracy analysis in the future.

Besides, large body of Android data leak research work [12,13,18] help us consider the sources (API-Source type) and sinks (API-Sink type) more comprehensively. For example Enck [18], Beresford et al. [1] only take network sinks of data into considerations. Droidtrack [22] focuses on the message outlet. In SCandal [48], API calls that can transfer data to the network, file or SMS are considered as sinks. Then in our design, we take all the above sinks into considerations and add Bluetooth, email, and multimedia message outlet to make our detections more complete.

6.2 Malware Visualization Work

In 2015, Wagner et al. [24] provide a systematic overview and categorization of malware visualization systems from the perspective of visual analytics. Current individual malware analysis visualizations referred in this paper [25–32,47] are all personal computer platform malware checking methodologies.

What's more, most of the sample features considered in these approaches for building visualization systems, such as the network activity of a malware sample [31], system calls issued over time [27], reversed bytes/byte segments/the repeated bytes sequences of the sample file [25,32], dynamically captured system activities [47], are not logical structure features embedded in the source or decompiled code. Only approaches of Quist, Chan et al. [26,29,30] are a little similar to our approach in constructing structural code profiles. Quist et al. [26,29] monitor and track program execution to construct a directed graph of all the basic blocks of an executable. Chan et al. [30] construct sample minigraph, which is a static control flow graph, to help monitoring and visualizing the dynamic executive path of binary creature. They use their graphs in the reverse engineering process to aid the Run-time debugging of malware, instead of directly helping understanding the malware behaviors.

As for visualizations aiming at supporting the Android malware analysis, the research has just started. Park et al. [20] focused on the checking visual similarity among Android malwares and deciding the degree of similarity. González et al. [21] apply neural projection architectures to analyze malware APPs data and characterize malware families. Both of them aim at analyzing the Android malware family similarity rather than individual malware checking. Androgurad [35] provides a basic generation and view function for Android call graph and control flow graph. However, it does not provide further capability of malware profile detection. Thus, it is more like a data provider rather than a visualization tool. Oscar else [19] proposed a tool to view a list of restricted API functions used at runtime of the application, but they cannot show the full calling chain for

that API and the correlation. Base data of [19,22] is dynamic monitored, which is not as comprehensive and informative as static code since dynamic executions cannot cover all the code paths.

7 Conclusion and Future Work

In this paper, we present a visualization analysis method to help Android security experts to study the structural malicious profiles of APPs. Our method is mainly based on a brand new kind of lightweight APP call graph $PMCG_{droid}$. This graph not only restores the malicious core of malwares for visually checking, but also behaves well in machine learning classification as feature sources. By designing visual interfaces with rich interactions, we show how to assist users in checking the APP's malicious behaviors and their entire triggering paths.

Our current work mainly focuses on sensitive APIs as target objects. In other scenarios, users can set their own targets, for example, code about encryption and decryption (may be used for shelling and shelling-off), advertisements, reflection calls and so on, to meet different security analysis needs or visual needs. Our framework is extensible to meet these requirements just by modifying some labeling rules.

Although we can detect the behavior of developers trying to dynamically load code by detecting related APIs such as "DexClassLoader", our current approach does not work on dynamically loaded code. Meanwhile, though our image similarity results inside same malware family indicate that $PMCG_{droid}$ may be suitable for clustering analysis of malware, we have not done this work yet. We will study them in the future.

In the future, we will further expand the visualization and artificial analysis assistance capability of $PMCG_{droid}$. Also we will study how to visualize the C/C++ code threats inside APPs.

Acknowledgment. We thank the anonymous reviewers for their insightful comments. Our work was supported by the National Key Research and Development Program of China (No. 2017YFB0801900), Key Program of the Chinese Academy of Sciences (No. ZDRW-KT-2016-02, ZDRW-KT-2016-02-6, Y6X0061105), and Youth Innovation Promotion Association of CAS (No. 1105CX0105).

References

1. Beresford, A.R., Rice, A., Skehin, N., Sohan, R.: MockDroid: trading privacy for application functionality on smartphones. In: 12th Workshop on Mobile Computing Systems and Applications, pp. 49–54. ACM (2011)
2. Octeau, D., McDaniel, P., Jha, S., Bartel, A., Bodden, E., Klein, J., Le Traon, Y.: Effective inter-component communication mapping in android with epicc: an essential step towards holistic security analysis. In: 22nd USENIX Security Symposium, pp. 543–558. USENIX (2013)
3. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: NDSS, pp. 50–52. NDSS (2012)

4. Enck, W., Ongtang, M., McDaniel, P.: On lightweight mobile phone application certification. In: 16th ACM Conference on Computer and Communications Security, pp. 235–245. ACM (2009)
5. Sun, M., Wei, T., Lui, J.: Taintart: a practical multi-level information-flow tracking system for android runtime. In: 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 331–342. ACM (2016)
6. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P.G., Álvarez, G.: PUMA: permission usage to detect malware in android. In: Herrero, Á., et al. (eds.) Advances in Intelligent Systems and Computing, vol. 189, pp. 289–298. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-33018-6_30
7. Acar, Y., Backes, M., Bugiel, S., Fahl, S., McDaniel, P., Smith, M.: SoK: lessons learned from android security research for appified software platforms. In: Security and Privacy IEEE, pp. 433–451 (2016)
8. Arp, D., Gascon, H., Rieck, K., Spreitzenbarth, M., Hbner, M.: DREBIN: effective and explainable detection of android malware in your pocket. In: NDSS. NDSS (2014)
9. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: ACM SIGPLAN Notices, vol. 49, no. 6, pp. 259–269 (2014)
10. Reina, A., Fattori, A., Cavallaro, L.: A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In: EuroSec, April 2013
11. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: automated security certification of android (2009)
12. Gibler, C., Crussell, J., Erickson, J., Chen, H.: AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale. In: Katzenbeisser, S., Weippl, E., Camp, L.J., Volkamer, M., Reiter, M., Zhang, X. (eds.) Trust 2012. LNCS, vol. 7344, pp. 291–307. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30921-2_17
13. Li, L., Bartel, A., Bissyandé, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., McDaniel, P.: IccTA: Detecting inter-component privacy leaks in android apps. In: 37th International Conference on Software Engineering, vol. 1, pp. 280–291. IEEE Press (2015)
14. Chakradeo, S., Reaves, B., Traynor, P., Enck, W.: Mast: triage for market-scale mobile malware analysis. In: The Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks, pp. 12–24. ACM (2013)
15. Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., Wu, K.-P.: Droidmat: android malware detection through manifest and API calls tracing. In: Information Security IEEE, pp. 62–69. IEEE (2012)
16. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for android. In: ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 15–26. ACM (2011)
17. Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C., Molloy, I.: Using probabilistic generative models for ranking risks of android apps. In: 2012 ACM Conference on Computer and Communications Security, pp. 241–252. ACM (2012)
18. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. Comput. Syst. (TOCS) **32**(2), 1–29 (2014)

19. Somarriba, O., Zurutuza, U., Uribeetxeberria, R., Delosières, L., Nadjm-Tehrani, S.: Detection and visualization of android malware behavior. J. Electr. Comput. Eng. **2016**, 6 (2016)
20. Park, W., Lee, K.H., Cho, K.S., Ryu, W.: Analyzing and detecting method of android malware via disassembling and visualization. In: International Conference on Information and Communication Technology Convergence, pp. 817–818. IEEE (2014)
21. González, A., Herrero, Á., Corchado, E.: Neural visualization of android malware families. In: Graña, M., López-Guede, J.M., Etxaniz, O., Herrero, Á., Quintián, H., Corchado, E. (eds.) ICEUTE/SOCO/CISIS -2016. AISC, vol. 527, pp. 574–583. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-47364-2_56
22. Sakamoto, S., Okuda, K., Nakatsuka, R., Yamauchi, T.: DroidTrack: tracking and visualizing information diffusion for preventing information leakage on android. J. Internet Serv. Inf. Secur. **4**(2), 55–69 (2014)
23. Grace, M., Zhou, Y., Zhang, Q., Zou, S., Jiang, X.: Riskranker: scalable and accurate zero-day android malware detection. In: The 10th International Conference on Mobile Systems, Applications, and Services, pp. 281–294. ACM (2012)
24. Wagner, M., Fischer, F., Luh, R., Haberson, A., Rind, A., Keim, D.A., Aigner, W.: A survey of visualization systems for malware analysis (2015)
25. Conti, G., Dean, E., Sinda, M., Sangster, B.: Visual reverse engineering of binary and data files. In: Goodall, J.R., Conti, G., Ma, K.-L. (eds.) VizSec 2008. LNCS, vol. 5210, pp. 1–17. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85933-8_1
26. Quist, D.A., Liebrock, L.M.: Visualizing compiled executables for malware analysis. In: International Workshop on Visualization for Cyber Security, pp. 27–32. IEEE (2009)
27. Trinius, P., Holz, T., Gbel, J., Freiling, F.C.: Visual analysis of malware behavior using treemaps and thread graphs. In: International Workshop on Visualization for Cyber Security, pp. 33–38. IEEE (2009)
28. Grgio, A.R.A., Santos, R.D.C.: Visualization techniques for malware behavior analysis. In: Proceedings of SPIE - The International Society for Optical Engineering, vol. 801905–801905-9 (2011)
29. Quist, D., Liebrock, L.M.: Reversing compiled executables for malware analysis via visualization. Inf. Vis. **10**(10), 117–126 (2011)
30. Chan, L.Y., Chuan, L.L., Ismail, M., Zainal, N.: A static and dynamic visual debugger for malware analysis. In: Communications, pp. 765–769. IEEE (2012)
31. Zhuo, W., Nadjin, Y.: MalwareVis: entity-based visualization of malware network traces. In: The Ninth International Symposium on Visualization for Cyber Security, pp. 41–47. ACM (2012)
32. Donahue, J., Paturi, A., Mukkamala, S.: Visualization techniques for efficient malware detection. In: IEEE International Conference on Intelligence and Security Informatics, pp. 289–291. IEEE (2013)
33. Yan, L.K., Yin, H.: DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In: The 21st USENIX Conference on Security Symposium, p 29. USENIX (2013)
34. G DATA news. https://www.gdata-software.com/news/2017/04/29715-350-new-android-malware-apps-every-hour
35. Androguard. https://github.com/androguard/androguard/
36. Chan, P.P.F., Hui, L.C.K., Yiu, S.-M.: Droidchecker: analyzing android applications for capability leak. In: The Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks, pp. 125–136. ACM (2012)

37. Android malware genome project. http://www.malgenomeproject.org/
38. Wang, K., Zhang, Y., Liu, P.: Call me back!: attacks on system server and system apps in android through synchronous callback. In: The 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 92–103. ACM (2016)
39. The Drebin dataset. https://www.sec.cs.tu-bs.de/~danarp/drebin/index.html
40. TROJAN. https://www.f-secure.com/v-descs/trojan_android_fakeinst.shtml
41. VirusTotal. https://www.virustotal.com/
42. Umeng. http://www.umeng.com/
43. Google maps android API. https://developers.google.com/maps/documentation/android-api/
44. AdMob. https://www.google.com/admob/
45. The life cycle of activity. https://developer.android.com/guide/components/activi-ties.html#Lifecycle
46. The life cycle of service. https://developer.android.com/guide/components/service-s.html#Lifecycle
47. Chner, T., Pretschner, A., Ochoa, M.: DAVAST: data-centric system level activity visualization. In: Eleventh Workshop on Visualization for Cyber Security, pp. 25–32. ACM (2014)
48. Kim, J., Yoon, Y., Yi, K., Shin, J.: SCANDAL: Static Analyzer for Detecting Privacy Leaks in Android Applications. Mobile Secur. Technol. Los Alamitos (2012)
49. Zhang, X., Aafer, Y., Ying, K., Du, W.: Hey, you, get off of my image: detecting data residue in android images. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9878, pp. 401–421. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45744-4_20
50. Huang, H., Zheng, C., Zeng, J., Zhou, W., Zhu, S., Liu, P., Chari, S., Zhang, C.: Android malware development on public malware scanning platforms: a large-scale data-driven study. In: 2016 IEEE International Conference on Big Data (Big Data), pp. 1090–1099. IEEE (2016)
51. Cao, Y., Fratantonio, Y., Bianchi, A., Egele, M., Kruegel, C., Vigna, G., Chen, Y.: EdgeMiner: automatically detecting implicit control flow transitions through the android framework. In: NDSS. NDSS (2015)