AOMDroid: Detecting Obfuscation Variants of Android Malware Using Transfer Learning

Yu Jiang[1], Ruixuan Li[1(✉)], Junwei Tang[1], Ali Davanian[2], and Heng Yin[2]

[1] Huazhong University of Science and Technology, Wuhan, China
{m201773045,rxli,junweitang}@hust.edu.cn
[2] University of California, Riverside, Riverside, USA
{adava003,heng}@cs.ucr.edu

Abstract. Android with its large market attracts malware developers. Malware developers employ obfuscation techniques to bypass malware detection mechanisms. Existing systems cannot effectively detect obfuscated Android malware. In this paper, We propose a novel approach to identify obfuscated Android malware. Our proposed approach is based on the intuition that opcode sequences are more resilient to the obfuscation techniques. We first propose an effective approach based on TFIDF algorithm to identify distinctive opcode sequences. Then we represent the opcode sequences as images and reduce the problem of identifying an obfuscated malware to the problem of transforming two images to one another, i.e. unobfuscated malware representation to the obfuscated one. In order to achieve the above, we resort to the transfer learning. We implemented a prototype dubbed AOMDroid based on the proposed approach and extensively evaluated its performance of accuracy and detection time. AOMDroid outperforms four related works that we compared with, and has an accuracy rate of 92.26% in detecting Android obfuscated malware. In addition, AOMDroid supports the detection of 21 Android malware family types. Its malware family detecion accuracy rate is 87.39%. The average time spent by AOMDroid to detect a single Android application is 0.963 s.

Keywords: Android security · Malware detection · Malicious behavior family · Obfuscation · Transfer learning

1 Introduction

Android attracts many attackers. A recent report on Android malware from a security vendor shows that in 2019 alone, a total of 1.809 million malware samples, and 950 million malware attacks on mobile devices [1] were intercepted. In order to detect Android malware, existing systems resort to machine learning and report an acceptable detection rate. However, attackers can use obfuscation techniques to greatly reduce the effectiveness [2].

N. Park et al. (Eds.): SecureComm 2020, LNICST 336, pp. 242–253, 2020.
https://doi.org/10.1007/978-3-030-63095-9_14

We observe that opcodes are more resilient to obfuscation techniques. The ultimate behavior of an application is summarized in the opcode. In addition, we note that learning based on opcodes takes a short time, detection based on opcode features looks promising. Based on above intuitions, we design and implement a method for detecting Android obfuscated malware. We aim to make a correspondence between opcode features before and after obfuscation. A key challenge is an effective feature selection since not all opcodes are distinctive. So we design a feature selection algorithm based on a TFIDF [3] matrix.

Another challenge is making a mapping between opcode features before and after obfuscation. Hence, we resort to transfer learning. The goal of the learning task is effectively transforming obfuscated samples to their unobfuscated ones, or in other words, reducing the difference in opcode features before and after obfuscation. We represent the selected opcode features as images and adopt a domain adaptive method. The underlying assumption is that the source and target domains have the same feature and category space, but there is a certain difference between the feature distribution of two domains. The goal of domain adaptation is to use the labeled source domain data and unlabeled target domain data, to learn a classifier and predict the label of the target domain data. Finally, we take the unobfuscated sample set as the source domain and the obfuscated sample as the target domain. The loss function used in the image transfer model includes the classification loss and the adaptive loss, the adaptive loss represents the difference between the feature distribution of source and target domains.

We implemented our proposed approach in a prototype called AOMDroid, and evaluated it with four different related works. AOMDroid outperforms all related works in terms of detection accuracy of obfuscated malware and the detection time. In summary, our main contributions are:

- We propose a novel anti-obfuscation feature selection algorithm. Our key insight is to apply TFIDF to opcode features before and after obfuscation and group opcode sequences based on constructs such as classes and methods.
- We propose a classification algorithm based on transfer learning that effectively identifies obfuscated malware. Our key idea is to represent the opcode sequences as images, and hence reduce the problem of obfuscated malware detection to the problem of minimizing difference between image pixels before and after obfuscation of an application via an image transformation.
- We implement a prototype based on the proposed approach and evaluate it extensively from accuracy and detection time. In terms of accuracy, our prototype, AOMDroid, achieves 92.26% on obfuscated samples outperforming four related works. For two related works that do not claim anti-obfuscation, AOMDroid is faster in detection time, and offsets their detection accuracy by 16% to 18%. Compared to other two anti-obfuscation related works, AOMDroid is more than 20 times faster while achieving a better accuracy.

2 Background

Obfuscation. Application obfuscation can convert files in the application installation package into forms that functionally equivalent but hard to understand.

It mainly include string encryption, benign code insertion, variable name obfuscation, class name obfuscation, resource obfuscation, API reflection obfuscation, and permission obfuscation. For instance, Permission obfuscation will modify permissions in AndroidManifest.xml. API reflection obfuscation will convert system calls into Java reflection calls. String encryption, variable name obfuscation, and class name obfuscation will obfuscate strings, variable names, and developer-defined class names that is incomprehensible. Due to the low coverage and efficiency in dynamic detection, we consider the static detection method. However, the effect of static detection methods will be significantly reduced due to obfuscation technology. A study shows that attackers can use obfuscation technology for 2000 Android malware samples, to reduce the detection accuracy from 65% to 5.8% of 58 mainstream antivirus engines [2].

Example. We review the malware code for intercepting SMS as shown below. The code uses the Broadcast to steal the SMS, employs the Intent with SMS to start a Service. In addition, It sends SMS content with the device ID to the malicious server. A detection method based on API calls can be constructed by detecting a sequence of getDeviceId, concat and sendData calls. This sequence contains three operations that can describe the behavior of obtaining a device ID, adding it to a message, and sending a message stealing the privacy. When the application adopts obfuscation techniques, it may use benign code obfuscation, insert irrelevant API to interrupt the API subsequence used to judge malicious code. It may also use API reflection obfuscation to convert an API direct call into an indirect API call, which makes it impossible to extract the API features.

```
public class monitorMessage extends BroadcastReceiver {
  public void onReceive(Context context, Intent intent) {
    SmsMessage mSms = SmsMessage.create();
    Intent mIntent = new Intent(Malicious.class);
    mIntent.putExtra("mSms", encrypt(mSms.getMessageBody()));
    startService(mIntent);
  }
}
public class sendMessageToNetwork extends Service {
  public void onStartCommand(Intent intent) {
    TelephonyManager manager = new TelephonyManager();
    String mId = manager.getDeviceId();
    URL maliciousUrl = new URL("http://xxx.com");
    maliciousUrl.sendData(mId.concat(intent.get("mSms")));
  }
}
```

3 Detection Scheme

3.1 Overview

The overview of our system is shown in Fig. 1. It includes: (a) selecting anti-obfuscation features; and, (b) constructing opcode image transfer model. We propose a anti-obfuscation malware classifier. The key idea is to select features for classification that obfuscation would have little effect on them. We start with a set of opcode sequences which represent three constructs: classes, methods and words. Class level, method level and word level opcode sequences refer, respectively, to fragments of opcode sequences for a single class, a single method and a single opcode within the application. We obfuscate a training sample and compare its features before and after obfuscation, and select features that have more weights in identifying malware even with obfuscation in place. We use Term Frequency-Inverse Document Frequency (TFIDF) to assign weights to different features and select those with a higher weight. TF is the frequency of opcode features in applications, and IDF is: if fewer applications contain an opcode feature, the opcode feature are good for distinguishing between categories. Next, we represent selected features as an image before and after obfuscation. Finally, we transfer the problem of classifying malware to image classification.

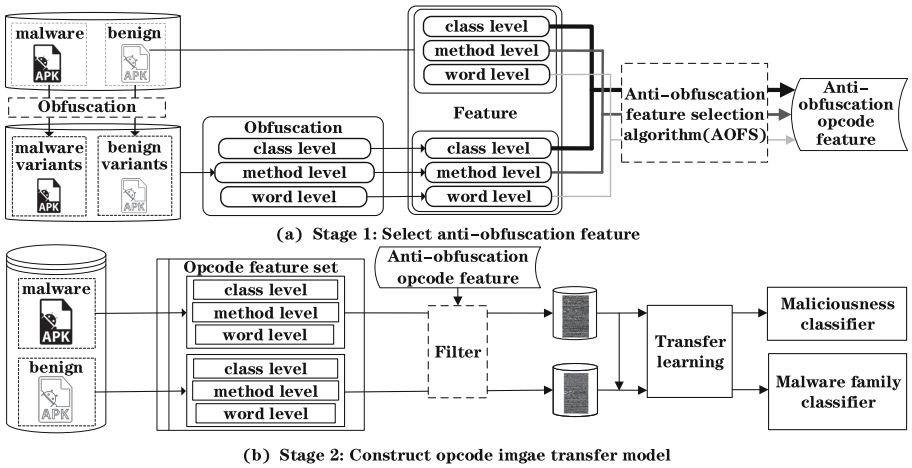

Fig. 1. Structure of the system.

3.2 Anti-obfuscation Feature Selection

Anti-obfuscation Feature Selection (AOFS) follows two design goals. Our selected features can distinguish the unobfuscated sample and be slightly affected by obfuscations. Opcodes, ultimately, characterize the behavior of Android applications and obfuscations impact relatively little on them. For instance, permission obfuscation influence little on opcode features. Yet, our experiments show that feature selection significantly improve the detection effect.

Algorithm 1: AOSF:ANTI-OBFUSCATION FEATURE SELECT

Input: Dictionary array of non-zero frequency features and frequencies of samples in unobfuscated datasets A_0, dictionary array of all non-zero frequency features and their frequencies in the set of obfuscated variants $A_{obfuscation} = [A_1, A_2, \cdot, A_n]$, opcode feature name set $featureNames$, number of selected features K.

Output: Selected anti-obfuscation opcode feature name set.

$n, m, t \leftarrow size(A_{obfuscation}, A_0, featureNames)$
initial array $tf_m, idf_m, tfidf_t, diff_t$ as zeros array
for $i \leftarrow 1$ ***to*** m **do**
  foreach $name, frequency \in A_0[i]$ **do**
    $tf[i] \leftarrow tf[i] + frequency$
    $idf[name] \leftarrow idf[name] + 1$
for $i \leftarrow 1$ ***to*** m **do**
  foreach $name, frequency \in A_0[i]$ **do**
    $tfidf[name] \leftarrow tfidf[name] + \frac{frequency}{tf[i]} * \log_2 \frac{m}{idf[name]+1}$
for $i \leftarrow 1$ ***to*** m **do**
  for $j \leftarrow 1$ ***to*** n **do**
    initial array $diffTmp_t$ as zeros array
    $featureFreqNot0Set \leftarrow \Phi$
    foreach $name, frequency \in A_0[i]$ **do**
      $diffTmp[name] \leftarrow frequency$
      $featureFreqNot0Set \leftarrow featureFreqNot0Set \cup name$
    foreach $name, frequency \in A_j[i]$ **do**
      $diffTmp[name] \leftarrow abs(diffTmp[name] - frequency)$
      $featureFreqNot0Set \leftarrow featureFreqNot0Set \cup name$
    foreach $feature \in featureFreqNot0Set$ **do**
      $diff[feature] \leftarrow diff[feature] + diffTmp[feature]$
for $i \leftarrow 1$ ***to*** t **do**
  $weight[i] = \frac{\frac{tfidf[i]}{m}+1}{\frac{diff[i]}{m*n}+1}$
return $topK(weight, K, featureNames)$

We collect opcode features of an application, and categorize them based on the granularity of obfuscation techniques. obfuscation techniques can be divided into three levels: class, method and word. In order to collect opcode features, we apply an obfuscation technique and extract opcode sequences at the same level of granularity e.g. for a method level obfuscation, we collect opcode sequences of a method. We use hash values to represent sequences on method or class levels. For the word-level opcode sequences, the extracted predefined opcode types, such as invoke-direct and iput-object. We also collect the frequency of each feature. Because the method level and class level feature frequency matrix is relatively sparse, we only extract the frequency information of non-zero frequency features.

The details of AOFS algorithm is illustrated in Algorithm 1. First, we compose a TFIDF matrix based on opcode features. The $tfidf_{jk}$ is the TFIDF value of the k_{th} opcode feature of the j_{th} Android application in the unobfuscated dataset. The difference indicators of the opcode features before and after obfuscation is the sum of the absolute values of the frequency differences in the feature frequency matrix before and after obfuscation in the unobfuscated dataset. Assuming that the number of obfuscations in the obfuscation technique set is n, the number of Android applications in the unobfuscated dataset is m, there are t types of opcode features in the obfuscation technique set. Before the obfuscation, the opcode feature frequency matrix is $A^{(0)}$. After the obfuscation, The obfuscated opcode feature frequency matrix is $A^{(1)}, A^{(2)}, \cdots, A^{(n)}$. $A_j^{(i)} = a_{j1}^{(i)}, a_{j2}^{(i)}, \cdots, a_{jt}^{(i)}$ is the opcode feature word frequency vector of the j_{th} Android application in the i_{th} dataset. $a_{jk}^{(i)}$ is frequency of the k_{th} opcode features of the j_{th} Android application in the i_{th} dataset. Assuming that w_k represents weights of the k_{th} opcode features, it is calculated as shown in Formula 1.

$$w_k = \frac{\frac{\sum_{j=1}^{m} tfidf_{jk}}{m} + 1}{\frac{\sum_{j=1}^{m}\sum_{i=1}^{n} |a_{jk}^{(0)} - a_{jk}^{(i)}|}{m*n} + 1} \tag{1}$$

AOSF filters out ineffective features in a hierarchical manner. First, it filters out the class level features that do not meet the preset threshold. Then, it filters out the method level features from remaining classes. Similarly, the word level opcode features are filtered out. Next, we sort opcode features. Features of different class level opcodes are sorted by the class name, and for the same class name, the order of methods in the class is used as the sort basis. Finally, we get opcode sequence to represent the application by connecting opcode features.

3.3 Detection Model

We transfer the problem of malware classification to image domain adaptation. We represent opcode features by the AOSF algorithm as an image. We build a model based on transfer learning [14] with the Resnet [5]. We train it by feeding unobfuscated applications with their obfuscated versions. The model goal is to transfer the obfuscated version to the unobfuscated one with the minimum loss.

Image Representation. We represent the selected opcode sequence features as a grayscale image. We assign a value between 00 and FF to an opcode based on Dalvik bytecode encoding rules [4]. Then, we divide opcode sequences to 256 opcodes per line. If the end line is less than 256 opcodes, we fill it with 0. Finally, The opcode features is converted into a grayscale image, as is shown below.

$$\begin{gathered} s_{11} \quad s_{12} \quad s_{21} \quad \cdots \\ \rightarrow invoke-direct \quad return-void \quad sget-object \quad \cdots \\ \rightarrow 112 \quad 14 \quad 98 \quad \cdots \\ \rightarrow \text{grayscale image of the opcode sequence} \end{gathered}$$

Transfer Learning Model. Android malware detection can be divided into two tasks: the detection of the existence of malware code and malware families. We train a transfer learning model such that these two tasks can be done simultaneously. We feed two inputs into our network: unobfuscated dataset representation as the source domain, and the corresponding obfuscated versions as the target domain. We note that feature and labels of both domains are the same, but the specific distribution of the images and their labels are different. The model realizes the transfer the features from the source (unobfuscated) domain to the target (obfuscated) domain. It can minimize the difference in the distribution between the source and target domains.

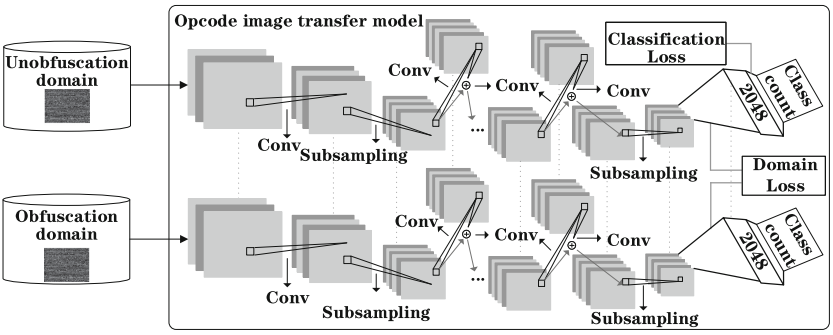

Fig. 2. Training phase of the Android malware obfuscated variants detection model.

Our model is shown in Fig. 2. Both networks share the same network direction and parameters during the training. Our classification network calculates classification loss and domain adaptive loss. Hence, the model can improve the classification accuracy for unobfuscated dataset and reduce the difference of feature distribution before and after obfuscation. The domain adaptive loss is the Maximum Mean Discrepancy (MMD) [13] distance between the data distribution of the unobfuscated application image and the data distribution of the obfuscated application image. As is shown in Eq. 2, the image distributions of the unobfuscated application image and the obfuscated application image are expressed, respectively, by X_U and X_O. $\phi(x)$ represents features of the image after the network's input layer and hidden layer. L is the overall loss function, L_C represents the classification loss, and λ is the weight of the adaptive loss.

$$MMD(X_U, X_O) = || \frac{\sum_{x_u \in X_U} \phi(x_u)}{|X_U|} - \frac{\sum_{x_o \in X_O} \phi(x_o)}{|X_O|} || \tag{2}$$

$$L = L_C + \lambda * MMD^2(X_U, X_O)$$

4 Evaluation

We evaluate AOMDroid, a prototype implementation based on the proposed approach. The evaluation provides insights about the AOMDroid efficiency as well as detection time. Furthermore, we compare AOMDroid with four related works and show that it outperforms all of them in analyzing obfuscated malware.

4.1 Dataset and Configuration

The original dataset contains 5560 malware and 4631 benign samples. We use AVPASS [2] to obfuscate samples. Obfuscation methods include string encryption, inserting benign code obfuscation, variable name obfuscation, class name obfuscation, resource obfuscation, API reflection obfuscation and permission obfuscation. The original malware comes from Drebin [6], and marked with the malware family label. benign samples come from the application market and verified by VirusTotal. AVPASS supports seven obfuscation methods by default that have been cited in the related works [7]. For malware families detection, unobfuscated dataset contains 4615 samples constructed from 21 malware families, the test dataset contains 32513 obfuscated samples.

For the original dataset, we use the successful decompilation samples. The unobfuscated and API reflection obfuscated dataset is used as the source and target domain respectively. All sample labels in the target domain are not added to the training, so seven obfuscated datasets are selected as the test dataset. When the learning rate is 0.01 and the batch size is 16, it reaches the highest accuracy rate. Furthermore, the training epochs are 200. The experiments were done on a computer with an Intel Core(TM) i7-8750H CPU and 16 GB of memory.

Table 1. Effect evaluation of malware detection based on transfer learning. The result is accuracy of detection. U is unobfuscated samples. A is API Reflection obfuscated samples. S is String encryption samples. V is variable obfuscated samples. K is class name obfuscated samples. E is insert benign class code samples. R is resource obfuscated samples. P is permission obfuscated samples. VA is samples of seven obfuscations.

Detection type	U	VA	A	S	V	K	E	R	P
Maliciousness	96.49%	92.26%	91.33%	93.81%	94.20%	83.46%	94.24%	94.30%	94.28%
Malware family	98.22%	87.50%	93.70%	93.81%	55.33%	93.42%	93.83%	93.84%	87.39%

4.2 Detection Effect and Performance of AOMDroid

We measure the accuracy of the proposed model in detecting maliciousness with and without obfuscation. The detection accuracy rate of the existence of malicious code in unobfuscated dataset is 96.49%, and the F value is 96.74%. For obfuscated variants, they are 92.26% and 93.19% respectively, as shown in

Table 1. We apply seven types of obfuscated variants to the unobfuscated dataset. The detection effect on class name obfuscation is low, because the class name obfuscation will change the order of the opcode sequence in the class and change image structure.

Similarly, we measure the malware family detection accuracy both for unobfuscated and obfuscated malware. The detection rate of malicious families against unobfuscated malware is 98.22%, while for seven malware obfuscated variants is 87.50%, as shown in Table 1.

The performance evaluation of AOMDroid considers two parts. The feature selection time is 305.63 s, refers to the time adopted the anti-obfuscation feature selection algorithm. The average time for malware prediction of one application is 0.963 s. Feature selection is in one process. For malware prediction, we divide datasets into 10 subsets, and test in ten processes.

4.3 Comparison with Prior Work

We compare AOMDroid with four related works: Drebin [6], PikaDroid [7], MCSC [8] and RevealDroid [9]. Drebin and PikaDroid are open source but MCSC and RevealDroid are not. We implemented MCSC and tested versus that but for RevealDroid, we rely on their reported numbers. PikaDroid and RevealDroid claim to support anti-obfuscation. Table 2 and 3 summarize the results.

Table 2. Obfuscated malware detection effect and performance evaluation with prior advanced methods that don't claim to support anti-obfuscation. The detection effect is trained on unobfuscated dataset and API reflection obfuscated variants and tested on samples in obfuscated dataset. The predict time is in seconds and tested for average single Android application in unobfuscated dataset by ten progresses.

Method	Average accuracy of malware obfuscated variants detection	Predict time
AOMDroid	**For permission obfuscated and seven obfuscated variants are 94.28% and 92.26% respectively**	**0.96**
Drebin [6]	For permission obfuscated variants is 77.17%	0.99
MCSC [8]	For seven obfuscated variants is 76.64%	3.67

Comparison with Drebin. Drebin and AOMDroid both train by unobfuscated dataset and API reflection dataset. AOMDroid detects permission obfuscated variants with a 94.28% accuracy while Drebin is 77.14%. Drebin is not effective in detecting obfuscated variants because it does not consider the obfuscation techniques. In terms of performance, AOMDroid is more efficient than Drebin in feature extraction. The average prediction time of AOMDroid for a single application is 0.963 s while Drebin is 0.998 s.

Comparison with MCSC. MCSC obfuscated variant detection accuracy is 76.64% while AOMDroid is 92.26%. MCSC is trained by unobfuscated dataset and API reflection dataset, when detecting malware obfuscated variants. In terms of prediction time and detection effectiveness, many hash calculations are required for extracting images in MCSC, so AOMDroid has advantages than MCSC.

Table 3. Obfuscated malware detection effect and performance evaluation with prior advanced methods that claim to support anti-obfuscation. The detection effect of AOMDroid and PikaDroid is based on 200 random unobfuscated samples and 1400 obfuscated samples. The predict time of AOMDroid and PikaDroid is in seconds and tested for average single Android application malware of 1600 samples by sixteen progresses. The detection effect and predict time of RevealDroid are obtained from its paper.

Method	Malware obfuscated variants detection accuracy	Predict time
AOMDroid	**For seven obfuscated variants is 93.56%**	**0.96**
PikaDroid [7]	For seven obfuscated variants is 93.46%	26.39
RevealDroid [9]	For four obfuscated variants drops to 85%	31.37

Comparison with PikaDroid. PikaDroid [7] uses machine learning models to detect Android malware based on API context features. It supports detection of obfuscated malware. We randomly selects 100 benign applications and 100 malicious applications from the original dataset, and uses seven obfuscation methods to generate 1400 obfuscated applications. Unobfuscated applications and permission obfuscated applications are used as source and target domain samples respectively, for training of AOMDroid. PikaDroid uses same training sets. PikaDroid uses program flow graph features based on API context, hence, AOMDroid has obvious advantages in detection time and detection accuracy.

Comparison with RevealDroid. RevealDroid [9] is not open source and reproduction is difficult. Obtaining key insights from the paper, AOMDroid has advantages in anti-obfuscation detection accuracy, the breadth of supporting obfuscation technologies, and the prediction time. RevealDroid detects whether the four obfuscated variants are malicious applications, the accuracy drops by more than 10%. Moreover, AOMDroid takes an average of 0.963 s to predict in parallel, while RevealDroid takes 31.3682 s.

5 Related Work

Android malware detection involves machine learning [6,7,9–12]. DroidAPI Miner [10] offers a machine learning approach based on API call features. Mariconti et al. [11] build markov chains of system APIs to characterize specific logical behaviors. Drebin [6], in addition to detecting maliciousness of Android applications, identifies malicious family. Recently, Hou et al. [12] construct a malware detection system based on heterogeneous information networks with deep

learning. The similar work to ours is MCSC [8]. It proposes SimHash to transfer application opcodes to images, and uses convolutional neural networks, but performs poorly. Similar to MCSC, we represent opcode features as images, but our method is anti-obfuscation and feature processing and model are different from MCSC. However, they are all not anti-obfuscation enough.

The accuracy of the aforementioned Android malware detection techniques would deteriorate when the malware is obfuscated. Several related works tried to address this problems [7,9]. PikaDroid [7] uses the API context, but it cannot effectively detect API reflection obfuscated Android malware. Another work, RevealDroid [9] employs multiple features and is resilient to API reflection obfuscation, but still lacks generality.

6 Limitation

AOMDroid would not perform well when analyzing the behavior of packed applications because the packing technologies will hide the actual behavior of the application. AOMDroid analyzes the Android application by static analysis. However, Some application packing technologies encrypt the Android application installation package, and the actual application executable file will be restored only when the application is actually running [15]. Another limitation is analysing Android applications that use dynamic loading technology to load the executable file that does not exist in the original installation package [16].

7 Conclusion

This paper proposes a transfer learning based approach to detect Android malicious obfuscated variants. We design a feature selection algorithm based on TFIDF, and represent selected features as images. Our transfer model learns the transformation between unobfuscated and their obfuscated applications, and hence identifies malware even after obfuscation. We evaluated the effectiveness and efficiency of AOMDroid, a prototype based on our proposed approach, by comparing it with four related works. AOMDroid outperforms all related works in terms of detection accuracy of obfuscated malware and the detection time.

Acknowledgement. This work is supported by the National Key Research and Development Program of China under grants 2016YFB0800402 and 2016QY01W0202, National Natural Science Foundation of China under grants U1836204, U1936108, 61572221, 61433006, U1401258, 61572222 and 61502185, and Major Projects of the National Social Science Foundation under grant 16ZDA092.

References

1. 360, Android malware special report in 2019 (2020). http://blogs.360.cn/post/review_android_malware_of_2019.html. Accessed 3 June 2020

2. Jung, J., Jeon, C., Wolotsky, M., Yun, I., Kim, T.: AVPASS: leaking and bypassing antivirus detection model automatically. In: Black Hat USA Briefings (Black Hat USA) (2017)
3. Salton, G., Yu, C.T.: On the construction of effective vocabularies for information retrieval. In: Proceedings of the 1973 International Conference on Research on Development in Information Retrieval, SIGIR, pp. 48–60. ACM (1973)
4. Google: Dalvik Opcode (2020). https://source.android.com/devices/tech/dalvik/dalvik-bytecode. Accessed 3 June 2020
5. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, pp. 770–778. IEEE (2016)
6. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: DREBIN: effective and explainable detection of Android malware in your pocket. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium, NDSS. ISOC (2014)
7. Allen, J., Landen, M., Chaba, S., Ji, Y., Chung, S.P.H., Lee, W.: Improving accuracy of Android malware detection with lightweight contextual awareness. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC, pp. 210–221. ACM (2018)
8. Ni, S., Qian, Q., Zhang, R.: Malware identification using visualization images and deep learning. Comput. Secur. **77**, 871–885 (2018)
9. Garcia, J., Hammad, M., Malek, S.: Lightweight, obfuscation-resilient detection and family identification of Android malware. In: Proceedings of the 40th International Conference on Software Engineering, ICSE, p. 497. IEEE/ACM (2018)
10. Aafer, Y., Du, W., Yin, H.: DroidAPIMiner: mining API-level features for robust malware detection in Android. In: Proceedings of the 9th International Conference on Security and Privacy in Communication Networks, SecureComm, pp. 86–103. ACM (2013)
11. Mariconti, E., Onwuzurike, L., Andriotis, P., Cristofaro, E.D., Ross, G.J., Stringhini, G.: MaMaDroid: detecting Android malware by building Markov chains of behavioral models. In: Proceedings of the 24th Annual Network and Distributed System Security Symposium, NDSS. ISOC (2017)
12. Hou, S., Ye, Y., Song, Y., Abdulhayoglu, M.: HinDroid: an intelligent Android malware detection system based on structured heterogeneous information network. In: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, SIGKDD, pp. 1507–1515. ACM (2017)
13. Pan, S.J., Tsang, I.W., Kwok, J.T., Yang, Q.: Domain adaptation via transfer component analysis. IEEE Trans. Neural Networks **22**(2), 199–210 (2011)
14. Tzeng, E., Hoffman, J., Zhang, N., Saenko, K., Darrell, T.: Deep domain confusion: maximizing for domain invariance. arXiv 1412.3474 (2014)
15. Duan, Y., et al.: Things you may not know about Android (Un)packers: a systematic study based on whole-system emulation. In: Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS. ISOC (2018)
16. Xue, Y., et al.: Auditing anti-malware tools by evolving Android malware and dynamic loading technique. IEEE Trans. Inf. Forensics Secur. **12**(7), 1529–1544 (2017)