Recommended Paper

Auto-creation of Robust Android Malware Family Trees

Kazuya Nomura[1,†1,a)] Daiki Chiba[2,†1,b)] Mitsuaki Akiyama[2,c)] Masato Uchida[1,d)]

Received: March 17, 2021, Accepted: September 9, 2021

Abstract: Malware targeting Android OS has been increasing for years and Android malware cyberattacks in particular are growing in number. To provide effective countermeasures against Android malware, we need to not only detect the malware at a certain point in time but also analyze the time-series changes in the malware, given that the family of Android malware will increase in number over time. In this paper, we propose a new method for automatically creating a "family tree" of Android malware that can represent how the newly detected Android malware relates to existing Android malware and its families and how they have changed over time. Our evaluation based on two actual Android malware datasets shows that our proposed family tree can accurately represent time-series changes between malware families.

Keywords: Android, malware, family tree

1. Introduction

The Android OS is an operating system that is used in a variety of devices such as smartphones. The Android OS runs on more than 2.5 billion devices and its worldwide market share will reach 85% in 2020 [2]. This makes Android an attractive target for attackers as well with more than 350,000 new Android malware variants discovered daily in 2018 [2]. Similar to PC malware, Android malware continues to increase in number and type or *family* over time. Thus, to develop sufficient countermeasures against Android malware, it is important to consider not only whether an Android Application Package (APK) can be detected as malware at a certain point in time but also its changes in the time series of their variations and evolving processes [3]. Furthermore, we should fully understand the threats by correlating Android malware with the corresponding attack campaigns and trends [4].

There are a number of studies that use machine learning to classify Android malware families. However, many of these have some shortcomings against actual cybersecurity threats. For example, the trends and properties of malware are ever-changing [4], but in machine learning, there is a problem called concept drift [3], where changes between training data and evaluation data cause a decrease in accuracy. For classifying such ever-changing malware, in this paper, we propose a new method for automatically creating a *family tree* of Android malware under more realistic and practical problem settings. Our proposed method creates an Android malware family tree via an unsupervised or semi-supervised approach to accommodate new types of software families. The created family tree is evaluated by features extracted by a method that closely considers time series changes. This family tree represents the ancestral and descendant-like relationships between multiple Android malware and the time series of their variations. This family tree enables us to understand how newly detected Android malware relates to historical malware and families. Furthermore, we can quickly identify new mutation-like families that emerge after a certain point in time. The information provided by our method is extremely useful for detecting ever-changing Android malware attacks and understanding their campaigns and attack trends.

The main contributions of our study are as follows:

- We developed a method for automatically creating a family tree of Android malware APKs that can represent how newly detected Android malware relates to existing Android malware and its families, and how they have changed over time.
- Our evaluation using two actual Android malware datasets shows that the family tree created by our proposed method accurately represents the relationships between Android malware families and correctly detects the emergence of new families.

2. Android Malware

2.1 Android Malware and Its Analysis

Typical Android malware includes adware, potentially unwanted program (PUP), ransomware, and trojan horses. The type of malware is identified as a *family* and is given the name of the family by anti-virus vendors. According to Zhang et al. [2], the raw family names given by anti-virus vendors are known to be inconsistent, with no standard naming rules.

Some Android malware includes certain schemes or program-

[1] Waseda University, Shinjuku, Tokyo 169–0051, Japan
[2] NTT, Musashino, Tokyo 180–8585, Japan
[†1] Presently with NTT Security (Japan) KK
[a)] kazuya-1997@asagi.waseda.jp
[b)] daiki.chiba@ieee.org
[c)] akiyama@ieee.org
[d)] m.uchida@waseda.jp

The preliminary version of this paper was published at Computer Security Symposium 2020 (CSS2020)/anti Malware engineering WorkShop 2020 (MWS 2020), Oct 2020. The paper was recommended to be submitted to Journal of Information Processing (JIP) by the program chair of MWS2020.

ming to avoid being easily detected by anti-virus software. For example, there are two known methods: obfuscating the code and repackaging, which embeds malicious code in benign Android applications and recompiles them [5].

Android malware analysis methods can be broadly divided into dynamic and static analyses. Dynamic analysis actually executes a malware program and monitors its behavior for analysis. For example, there are techniques that either monitor CPU usage and processes or capture malware network traffic. The dynamic analysis method is robust against obfuscation and can accurately reveal malware behavior. However, this method requires a comprehensive monitoring of malware behavior. For Android malware in particular, there are cases in which the malware operation requires on-screen user interaction which makes dynamic analysis difficult to conduct. Static analysis uses information that can be obtained without executing the malware. For example, methods have been developed that focus on the permissions required by Android applications and API calls provided by Java or Android Software Development Kit (SDK). Static analysis is difficult to apply to obfuscated code however we can conduct a fast and scalable analysis without the time and effort required to run Android malware.

2.2 Android Malware Detection

Several machine learning-based detection methods have been used for detecting Android malware in addition to signature-based detection methods using known malware information. Specifically, previous studies have used features obtained from dynamic or static analysis (in Section 2.1) to apply a binary classification as to whether the APK is malware [6], [7], [8], and to apply family classification within malware [2], [9].

When applying machine learning to a target such as Android malware APKs, which continue to increase in number over time, it is important to achieve robust detection performance that considers not only the detection performance at a certain point in time, but also the changes in the malware over time. With machine learning such a significant change in the nature of a target is called *concept drift* [3]. For example, when predicting malware family names, concept drift occurs at the time of the appearance of a new family that is not present in the training data. Pendlebury et al. [10] showed that several previous studies on detecting Android malware did not sufficiently consider such conceptual drift, and that detection accuracy was reduced when the detection algorithm was tested under realistic time-series-based experimental conditions.

3. Proposed Method

In this paper, we propose a new method for creating a *family tree* of Android malware that takes into account changes in the time-series which denote the trends of the attacks and appearances of new families. Our family tree is defined as a timeline of relationships among Android malware linking a similar malware family with similar malicious behaviors and purposes. As shown in the **Fig. 1**, our proposed method uses Android malware as the input and goes through two steps (`feature engineering` and `family tree creation`) and finally outputs a family tree.

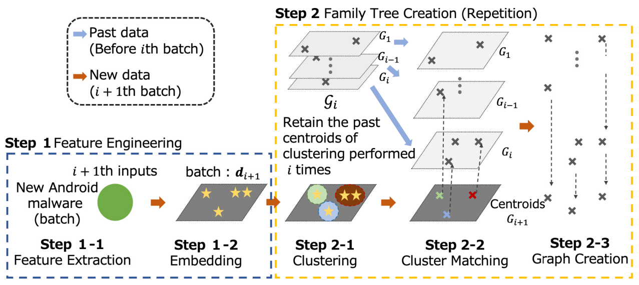
Past data
(Before ith batch)
New data
($i+1$th batch)
Step 2 Family Tree Creation (Repetition)
G_1
G_{i-1}
G_i
$\mathcal{G}_i$
Retain the past centroids of clustering performed i times
G_1
G_{i-1}
G_i
Centroids G_{i+1}
Step 1 Feature Engineering
$i+1$th inputs
New Android malware (batch)
batch : $\boldsymbol{d}_{i+1}$
Step 1-1 Feature Extraction
Step 1-2 Embedding
Step 2-1 Clustering
Step 2-2 Cluster Matching
Step 2-3 Graph Creation

Fig. 1 Overview of proposed method.

3.1 Step 1: Feature Engineering

We focused on the long-term and large numbers of Android malware to create a family tree. To this end, we extracted the features required for creating a family tree based on a fast and scalable static analysis method. Specifically, we used the MobSF [11] to extract features from Android malware.

3.1.1 Step 1-1: Feature Extraction

We selected two feature sets (permissions and API calls) that can capture the behavior and the family of the malware.

Permissions. In Android apps, the various privileges required for the apps are managed by a mechanism called *permissions*. Permission-based features have been used in various studies because they provide a clear indication of Android malware behavior. Arora et al. [12] showed that the use of permission-based features is important for robust Android malware detection. The required permissions of Android malware can be obtained from the manifest file (`AndroidManifest.xml`). For example, if an Android malware requires Internet access, the permission `android.permission.INTERNET` is defined in the manifest file. In this study, we extracted features from the permissions required by each Android malware using the following procedures.

(1) Each permission required by an Android malware is separated by a period and tokenized.

(2) A feature vector is created using one-hot encoding to see if the Android malware contains a token. However, generic tokens (`android` and `permission`) that do not indicate the content of permission are excluded.

API Calls. Android apps can be decompiled or in other words, the source code and its directory structure are readily available. This study focuses on API calls that can be identified from the source code. Aafer et al. [6] showed that there are some common API calls for malicious Android malware activities. For example, if `getRuntime().exec()` or `getRuntime()` of the class `java.lang.Runtime` is called in the source code of an Android app, then the app may fork a process or namely create a new process and execute an arbitrary command. Utilizing the ideas of Ref. [6], we selected 49 such API calls, reflecting the more recent Android OS and malware landscape. Specifically, we used the API call extraction rule [13] in MobSF [11] for practical implementation. Similar to the permissions, we used one-hot encoding to characterize whether an API call is used by an Android malware APK.

For each Android malware APK, we represented the extracted features based on the above permissions and API calls in the matrix **D**. In the matrix **D**, each row corresponds to an Android malware APK, and each column corresponds to one feature. Hereafter, each row of $\mathbf{D}$ is ordered by the malware appearance date. Notably, in this study, we considered time-series changes in the features, and we created features based on permissions and API calls observed at a certain point in time, or namely we avoided using future information as a leakage.

3.1.2 Step 1-2: Feature Embedding by Using UMAP

When features extracted from permissions and API calls are used, data with the same label are not necessarily distributed close to each other in the feature space. However, for well-known malware that has already been analyzed sufficiently, it may be possible to assign a label such as the exact family name. In this study, we proposed a method to embed the extracted features into a low-dimensional feature space using labels from a known family name. For feature embedding, we used Uniform Manifold Approximation and Projection (UMAP) [14]. The main idea of UMAP is based on the theory of Riemannian manifolds and algebraic topology. UMAP performs dimensional reduction so as to preserve the local and global distance relations between data points. Points that are close to each other in high-dimensional data are trained to be close to each other when embedded in low-dimensional data. In order to represent the closeness of data points, UMAP uses an undirected weighted graph based on the k-neighborhood method to represent the neighborhood relation of high-dimensional data. This graph is called a fuzzy topological set in the original UMAP paper. UMAP is trained to preserve the neighborhood relation indicated by the weights of this graph when embedded in a low-dimensional space. UMAP can also perform supervised embedding. By embedding the features using UMAP, the distance structure in the feature space appropriately reflects the known label information. As a result, UMAP and its features are expected to improve the accuracy of clustering, which is performed in Step 2-1.

It is also possible to embed new data without labels in the space once it has been embedded in a supervised manner. Furthermore, when data with properties different from known ones are input, the features are embedded at a distance away from the known labels. Owing to the aforedescribed properties, embedding new data can be easily applied to the real-time creation of the family tree proposed herein.

3.2 Step 2: Family Tree Creation

This step uses the features extracted in Step 1 mentioned above to create a family tree for Android malware. As shown in Fig. 1, we created a family tree through an iterative process composed of the following three sub-steps: *clustering*, *cluster matching*, and *graph creation*. The underlying concept of this family tree creation is to make associations for new and not yet fully analyzed Android malware APKs available on a daily basis using historical malware data. As explained in Section 2, new families of Android malware continue to emerge over time and do not belong to the known families. Therefore, we selected an approach based on clustering that considers classification into pre-specified classes.

Step 2-1: Clustering. We use a clustering method to cluster multiple malware APKs obtained over time based on the extracted features of each Android malware APK. Hereafter, we refer to each row of data $\mathbf{D}$ (the feature vector of each malware APK) as a *sample*. We split the data $\mathbf{D}$ into m partitions in a time-series order and call each submatrix $(\mathbf{D}_1, \mathbf{D}_2, \cdots, \mathbf{D}_m)$ a *batch*. A batch describes features of multiple malware in a matrix. In addition, for $\mathbf{D}$, we obtain the standard deviation of the norm for each sample and divided $\mathbf{D}$ by it to adjust the scale of the norm as a preprocessing step.

We use the clustering method to cluster the samples contained in each batch $\mathbf{D}_i$, $(i \in \{1, 2, \cdots, m\})$ to obtain c_i clusters. We define the centroid $\mathbf{g}_{i,l}$, $(l \in \{1, 2, \cdots, c_i\})$ of each c_i cluster as the average of the feature vectors of the samples belonging to that cluster. We also define the set of centroids of a cluster as $G_i = \{\mathbf{g}_{i,1}, \mathbf{g}_{i,2}, \cdots, \mathbf{g}_{i,c_i}\}$ and their union as $\mathcal{G}_i = \bigcup_{j=1}^{i} G_j$. The set $\mathcal{G}_i$ contains all centroids of each cluster generated by clustering of the batches $\mathbf{D}_1, \mathbf{D}_2, \cdots, \mathbf{D}_i$.

Moreover, various algorithms can be used for clustering. In the evaluation of this study, we use the X-means method [15], which can automatically determine the optimal number of clusters. The X-means method is widely used with many implementations available and it is computationally fast.

Step 2-2: Cluster Matching. When we obtain a new batch $\mathbf{D}_{i+1}$, we apply clustering to the new $\mathbf{D}_{i+1}$ and obtain c_{i+1} clusters and a set of centroids G_{i+1}. We now search for past clustering results that are close to the current clustering results. Specifically, for each c_{i+1} centroid $(\mathbf{g}_{i+1,1}, \mathbf{g}_{i+1,2}, \cdots, \mathbf{g}_{i+1,c_{i+1}})$, which are elements of sets of centroid G_{i+1}, we use the k-nearest neighbor method to search k neighborhood centroids from elements of $\mathcal{G}_i$. We use a Euclidean distance as a distance measure in the k-nearest neighbor method. For a certain centroid of cluster $\mathbf{g}_{i+1,l}$, $(l \in \{1, 2, \cdots, c_{i+1}\})$ in $\mathbf{D}_{i+1}$, we obtain (1) $Neighbor_{i+1,l}$, the set with k neighborhoods explored from $\mathcal{G}_i$, and (2) $Distance_{i+1,l}$, the set with the distances between those k neighborhoods and $\mathbf{g}_{i+1,l}$. We set k to c_1, which is the number of clusters in $\mathbf{D}_1$. We apply this setting because when we match the second cluster to the first batch $\mathbf{D}_1$, we cannot obtain more than the number of neighbors compared to the first cluster.

Step 2-3: Graph Creation. We define the family tree $P = (N, E)$ as a directed acyclic graph (DAG), where N is a set of nodes corresponding one-to-one with the elements in the set of centroids of the cluster $\mathcal{G}_i$ and E is a set of edges with information of distance d. We create nodes $n_{i+1,l}$, $(l \in \{1, 2, \cdots, c_{i+1}\})$ corresponding to the centroid $\mathbf{g}_{i+1,l}$ of the each cluster obtained from $\mathbf{D}_{i+1}$. We connect the created nodes to the nodes corresponding to the past centroid which is an element of $Neighbor_{i+1,l}$. However, we do not connect a node if the distance between nodes exceeds its threshold θ. By setting this threshold, we obtain a relationship between clusters in which the nodes of the closer clusters are connected and maintain a time-series relationship that is useful as a family tree.

4. Evaluation

4.1 Dataset

We evaluated the effectiveness of our method by using two actual Android malware datasets as shown in **Table 1**. A dataset by ArgusLab [16] contains 24,650 Android malware APKs and their ground truth family name labels. The malware family names

Table 1 Overview of the Android malware dataset used in the experiments.

Dataset	# of Malware APKs	Observation Periods	# of Malware Families
ArgusLab [16]	24,474	2009/10/14~2016/5/14	71
AndroZoo [17]	25,740	2010/09/10~2020/9/16	471

Table 2 Malware families included in the ArgasLab dataset.

Family	# of Malware APKs
Airpush	7,843
Dowgin	3,356
FakeInst	2,168
Mecor	1,817
Youmi	1,300
Fusob	1,275
Kuguo	1,199
other families	5,516
Total	24,474

Table 3 Malware families included in the AndroZoo dataset.

Family	# of Malware APKs
Dowgin	4,082
Kuguo	2,869
SMSreg	1,910
gappusin	1,801
airpush	1,744
adwo	1,557
other families	11,777
Total	25,740

were given based on the detection results by VirusTotal [18], and were labeled based on the content of the detection results by more than 50% of anti-viruses. **Table 2** shows the number of APKs and malware families contained in the ArgusLab dataset.

We also used the AndroZoo [17] dataset for evaluation which includes more recent data and a larger number of families of malware APKs. In this study, we used a random sampling of 27,851 malware APKs detected by more than 10 antiviruses on VirusTotal from the AndroZoo dataset, which contains approximately 14 million malware APKs. This dataset does not contain ground truth family name labels. Therefore, based on the detection results of VirusTotal, AVClass2 [19] was used to assign the correct ground truth label. **Table 3** shows the number of APKs and malware families contained in the AndroZoo dataset.

The information for "the date and time when the malware was created" and "the date and time when the malware was observed" are not included in either dataset; thus, it is not possible to follow the time-series changes that we focus on in this study. Therefore, we additionally used the API provided by VirusTotal to obtain all dates on which all APK files in the dataset were first observed. As a result, we found that the malware APKs in the ArgasLab dataset were observed from October 14, 2009, to May 14, 2016, and those in the AndroZoo dataset were observed from September 10, 2010, to September 16, 2020, as shown in the "Observation Periods" in Table 1. The time evolution of the seven families of malware (with more than 1,000 samples) in each dataset shown in Tables 2 and 3 are shown in **Figs. 2** and **3**. We can see two aspects from these figures. First, the family and percentage of malware vary depending on the timespan of the dataset. Second, there is a new family of malware that appears after a certain point in time and the concept-drift occurs at that point.

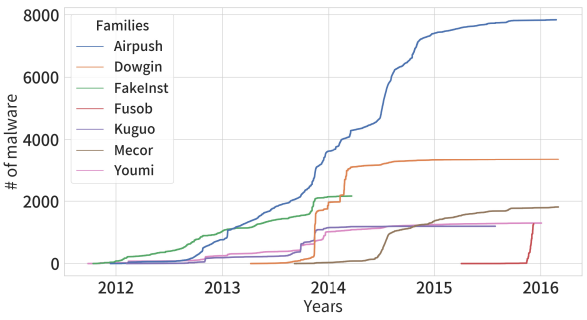

Fig. 2 The appearance of malware families in ArgusLab dataset.

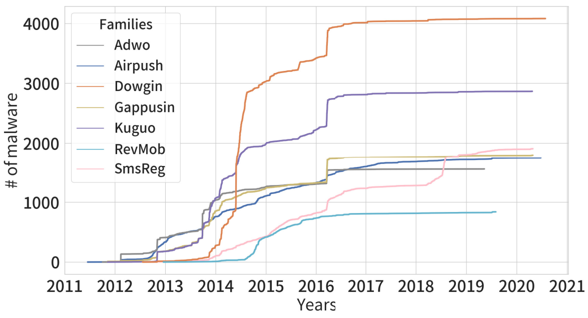

Fig. 3 The appearance of malware families in AndroZoo dataset.

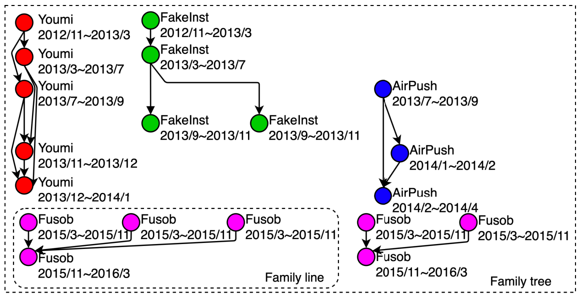

Fig. 4 Example of created Android malware family tree.

4.2 Creating An Android Malware Family Tree

We apply the method shown in Section 3 to our datasets (ArgasLab and AndroZoo) to generate a family tree. To improve the computational efficiency, we use permissions that commonly appear in two or more malware. The dimension of the space in which the features were embedded in the UMAP was set to 50. In this study, we randomly sampled 50% of the malware in each dataset (12,237 malware APKs in the ArgasLab dataset, and 12,870 malware APKs in the AndroZoo dataset) and trained UMAP using 50% of the malware APKs, to show that the accuracy of the family tree can be improved by using the family information of a part of the dataset. The remaining 50% of the malware was not used for UMAP learning but was used for creating a family tree.

Result Thereafter, we split the malware APKs into batches of 1,000 APKs in order of their first-seen dates (see Step 2-1). **Figure 4** shows a part of the family tree created when the distance threshold in our method is $\theta = 0.1$. In this figure, the date progresses from top to bottom, and the color of the node indicates the most frequent family in the cluster corresponding to that node.

Discussion This automatically generated family tree shows that related families are joined together as the same family, and by following the graph backwards, it is easy to correlate an APK with

Table 4 Clustering metric results.

Dataset	Used Feature	Evaluation Metric	Average Score±Standard Deviation
ArgasLab	Original Extracted Feature (Unsupervised)	Average percentage of the most frequent family	0.930±0.004
		homogeneity_score	0.854±0.009
	UMAP Embedded Feature (Supervised)	Average percentage of the most frequent family	0.931±0.015
		homogeneity_score	0.984±0.001
AndroZoo	Original Extracted Feature (Unsupervised)	Average percentage of the most frequent family	0.651±0.015
		homogeneity_score	0.506±0.009
	UMAP Embedded Feature (Supervised)	Average percentage of the most frequent family	0.904±0.004
		homogeneity_score	0.833±0.010

historical malware and its family. Since the family tree is created based on the features extracted from API calls and permissions, it is even possible to link different malware families with similar behavior. The accuracy of the family tree in such a case is evaluated in Section 4.4.

4.3 Evaluation of Clustering

Here we aim to evaluate the clustering results and validity of the family tree. To this end, we validated the accuracy of the family classification for each cluster before connecting the edges of the family tree (Step 2-1). To evaluate the accuracy, we used the ground truth labels contained in the dataset. Specifically, we use the following two metrics to evaluate the clustering accuracy.

Average Percentage of the Most Frequent Family. This metric averages the percentage of the most family within each cluster. Although this metric is intuitive and straightforward, it has the disadvantage of not being able to consider the size of each cluster or the number of mixed families.

Homogeneity_score. To compensate for the disadvantage of the above metric, we also used the homogeneity_score [20] to evaluate the clustering results. The homogeneity_score h is defined as follows when dividing the N data with n different class labels $K = \{k_1, k_2, \cdots, k_n\}$ into m clusters $C = \{c_1, c_2, \cdots, c_m\}$:

$$h = \begin{cases} 1, & H(K|C) = 0 \\ 1 - \frac{H(K|C)}{H(K)}, & \text{otherwise} \end{cases}$$

where H is the entropy. In addition, H is defined as follows if the number of data points in class k and cluster c is $a_{k,c}$:

$$H(K|C) = -\sum_{c=1}^{|C|}\sum_{k=1}^{|K|} \frac{a_{k,c}}{N} \log \frac{a_{k,c}}{\sum_{k=1}^{|K|} a_{k,c}}$$

$$H(K) = -\sum_{k=1}^{|K|} \frac{\sum_{c=1}^{|C|} a_{k,c}}{N} \log \frac{\sum_{c=1}^{|C|} a_{k,c}}{N}.$$

We considered each of the family names as the class labels K. Thus, the homogeneity_score increases to nearly 1 when each cluster contains only one family.

Result **Table 4** shows the means and standard deviations of the two metrics mentioned above for 10 rounds of clustering for each dataset. We also show a comparison between the results obtained by completely unsupervised clustering on the extracted features as they are without UMAP, and the results obtained by clustering on vectors embedded with features in a supervised manner using UMAP.

Discussion It can be seen from the table that both completely unsupervised clustering and clustering with UMAP recorded high scores on the ArgusLab dataset. In particular, when UMAP is applied, the homogeneity_score is very high at 0.93, indicating that not only is one cluster occupied by a single family, but it is also less likely to be misclassified across multiple clusters. Meanwhile, the accuracy of the unsupervised clustering for the AndroZoo dataset is low. This is likely because the AndroZoo dataset contains more recent malware samples as well as malware from longer periods of time and more varieties of families. However, when UMAP is applied and the family tree is created in a supervised manner, the accuracy improves significantly. These results indicate that the clustering necessary to create a family tree can be performed with stable accuracy. It can also be seen that if even a part of the malware family data can be obtained, then supervised learning with UMAP can be used for more accurate clustering.

4.4 Evaluation of Family Tree

To make the family tree useful for estimating malware families and behavior, each connected set of nodes in the created family tree should be occupied by the same family. Now, we evaluate whether each connected set of nodes in the family tree has these characteristics.

To carry out this evaluation, we consider a tree structure in the family tree that considers whether nodes are connected to each other, regardless of the direction of the graph. Hereafter, this tree structure or subgraph is referred to as the *family line*. In Fig. 4, we present an example of a family line in our family tree. To indicate the validity of the family tree, we consider the *family line* to which the nodes are connected as one large cluster and use the ground truth family name labels to evaluate the accuracy of the large clusters.

Result As shown in Fig. 4, the same family does not necessarily belong to the one family line. When we investigate the malware traffic in each family, we found that the combination of domain names connected by each malware in each family tree differs greatly. These results show that communication behavior is different even for malware which has the same given family name. Results also show that the family tree created by our method correctly captures these characteristics.

Figure 5 shows the change in average accuracy depending on the threshold of the distance to cut the edges. The horizontal axis shows the threshold for cutting the edges in the family tree, and the vertical axis shows the homogeneity_score. The line colors indicate the difference in the datasets, where the crosses indicate the results of unsupervised family tree creations and the circles indicate the results of the supervised family tree creation using UMAP. The vertical axis shows the homogeneity score, which is the stricter of the two scores described in Section 4.3. The homo-

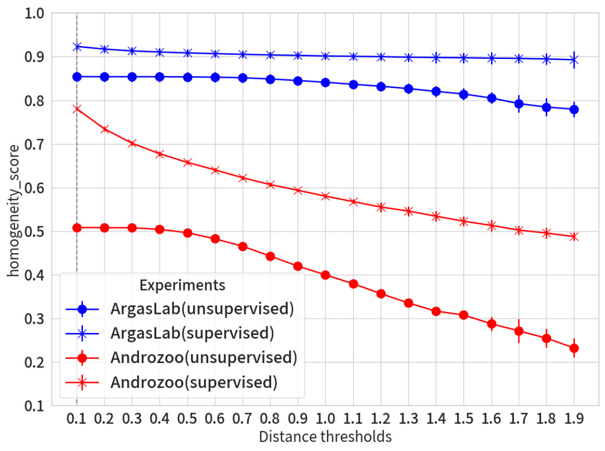

Fig. 5 Evaluation of family tree.

geneity_score was calculated by considering each family line as a cluster $C = \{c_1, c_2, \cdots, c_m\}$.

Discussion In this evaluation experiment, we compared the homogeneity_score by considering each cluster connected in the family tree as a "family line" and the family line as a new cluster. Therefore, if unrelated clusters are connected to each other as a *family line* using the method proposed in this study, the homogeneity_score will be reduced compared to the accuracy of simple clustering. However, if the distance threshold θ is set to 0.1, we cannot observe a significant difference from the homogeneity_score of clustering shown in Table 4. Thus, the clusters connected in the family tree are valid connections in terms of family classification, and it is possible to estimate the malware family that belongs to a new family.

We now discuss the tradeoff between the distances connecting the edges of the graph. The smaller the distance threshold used to cut the edges, the closer the connection of the clusters to the family tree and the more concise their association with previous malware samples. In contrast, the greater the distance threshold used to cut the edges, the greater the number of connected clusters and the larger the number of nodes that can be referenced by the previous malware samples. In addition, we can obtain many suggestions for neighborhood malware samples. In the following experiments, we set a distance threshold of 0.1.

4.5 Evaluation of Concept-drift

4.5.1 Evaluation of Concept-drift Robustness

We evaluate the robustness of our proposed method when new malware families appear in the dataset or when concept-drift occurs. Our method creates feature vectors based on permissions and API calls as described in Step 1. When the SDK, Android APIs, and libraries are updated, the permissions and API calls used by Android malware may change, requiring feature redesign and model recalculation. To account for this change, we generated one-hot vectors using tokens that appeared before half of the malware in the dataset appeared and set limits on the number of features used. Supervised learning with UMAP was also performed in the same way or namely only data until emergence of half of the malware APKs in the entire dataset was used.

Result We performed the same experiment as in Section 4.2, except for the feature extraction. As in Section 4.2, **Figs. 6** and **7**

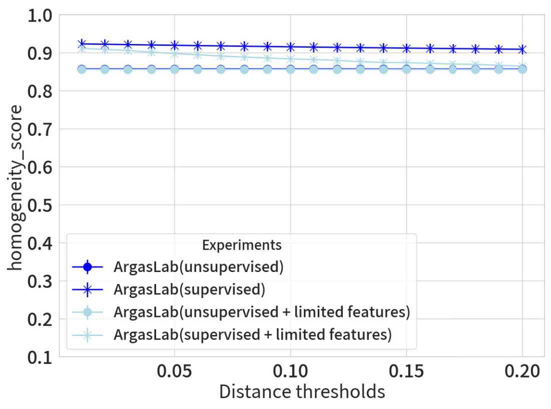

Fig. 6 Evaluation of Concept-drift robustness of family tree (ArgasLab Dataset).

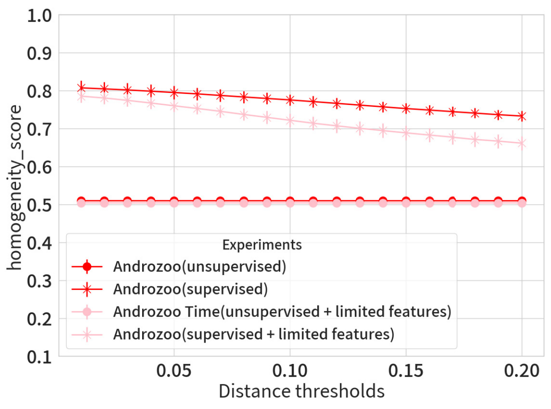

Fig. 7 Evaluation of Concept-drift robustness of family tree (AndroZoo Dataset).

show the change in the accuracy of the family tree depending on the threshold in each dataset.

The accuracy when the extraction of features is not limited is shown by the dark-colored line in the graph. The accuracy when feature extraction is limited is shown by the light-colored line in the graph.

Discussion Comparing both lines, it was found that although a slight decrease in accuracy was observed, the restriction on feature extraction based on temporal information had little impact on accuracy. In addition, the decrease in accuracy when UMAP was used on the AndroZoo dataset was slightly larger than the decrease in accuracy on the ArgasLab dataset. This difference may be due to the fact that there is a six-year gap between the appearance of half of the data used for training and the end of the dataset. However, we can say that it maintains sufficient accuracy compared to clustering with completely unsupervised learning.

4.5.2 Evaluation of Concept-drift Detection

Next, we show that our method can be used to detect concept-drift and contribute to preventing a decrease in the accuracy of general machine learning models that classify malware families. To carry out this evaluation, we implement a new multiclass classification that predicts which family a given Android malware belongs to and evaluated its accuracy. We used all families of malware shown in Table 1 and their family trees for the dataset, extracting the same features as our method, and training the multiclass Random Forest model. We trained and evaluated the Random Forest model by shifting the testing samples by 1,000 and using all the samples preceding each testing sample as a training sample. Thus, refering to Table 1, this means that the Random Forest model was evaluated 24 times (24,474 samples for the ArgasLab Dataset) and 25 times (25,740 samples for the AndroZoo Dataset), respectively. To evaluate the model fairly, we conducted the above evaluations 10 times to minimize the influence of the initial seed and parameters. In other words, we evaluated 10 sets of 24 or 25 total trainings. For training the RandomForest model, we used the same features based on the one-hot vector as in the proposed method, and for generating the family tree we used the features embedded in UMAP. As in Section 4.5.1, the features used to create the family tree were based only on the data that appeared until exactly half of the malware in the each dataset appeared. Embedding by UMAP was applied in the same manner. The family tree is only used to detect concept drift based on the number of malware in a new family line that is not connected to any parent node in the family tree.

In Section 4.5.2, the Random Forest model used for validation is referred to simply as the *model*. Our method of concept-drift detection is based on the criterion of whether a *new node* appears that is not connected to any of the past nodes while generating the family tree in each batch. These new nodes can be considered to have weak relationships with past malware families and it is very probable for new malware families. We consider that a concept-drift occurs when even one of the new nodes appears or when the number of malware APKs belonging to a new node exceeds a certain number and we retrain the model. In this evaluation, we set this retraining threshold to 150 (for the ArgusLab dataset) or to 400 (for the AndroZoo dataset). To compare with state-of-the-art concept drift detection algorithms, we chose the D3 [21] algorithm that detects concept drift in an unsupervised manner. We evaluated the properties of such concept-drift detection in five different models, as given below.

- Retraining it every time (*All Train Model*)
- Retraining only when the number of malware APKs belonging to a new node exceeds 150 or 400 (*#150 Train Model* or *#400 Train Model*)
- Retraining once every two times (*Half Train Model*)
- Retraining only when the D3 algorithm detects concept drift (*D3 Train Model*)
- Retraining it the first time only (*First Train Model*)

Result **Tables 5** and **6** show the average accuracy of each model over 10 sets of evaluations, and the total number of times the retraining of the model was skipped as a result of concept-drift detection. **Figures 8** and **9** show the variation in accuracy. We call indexes of dataset that sub-divided into 24 segments (for the ArgasLab Dataset) or 25 segments (for the AndroZoo dataset) a *batch*. The horizontal axis of the graph shows the batch, and the vertical axis shows the average accuracy.

Discussion From the tables and figures we can conclude the following three facts. First, we can see that the accuracy of the model decreases when the number of retraining periods is significantly reduced. In particular, if the model is not retrained for a long period of time (e.g., *First Train Model*), it is not possible to classify a family of new malware, and the accuracy is significantly reduced. Second, our method for retraining when

Table 5 Accuracy evaluation of retrained models (ArgasLab Dataset).

Model	Avg. Accuracy	# of Skipped Retraining Times
All Train Model	0.895	0
#150 Train Model	0.883	5
D3Train Model	0.883	6
Half Train Model	0.853	12
First Train Model	0.438	24

Table 6 Accuracy evaluation of retrained models (AndroZoo Dataset).

Model	Avg. Accuracy	# of Skipped Retraining Times
All Train Model	0.607	0
#400 Train Model	0.601	3
D3 Train Model	0.446	12
Half Train Model	0.567	24
First Train Model	0.178	20

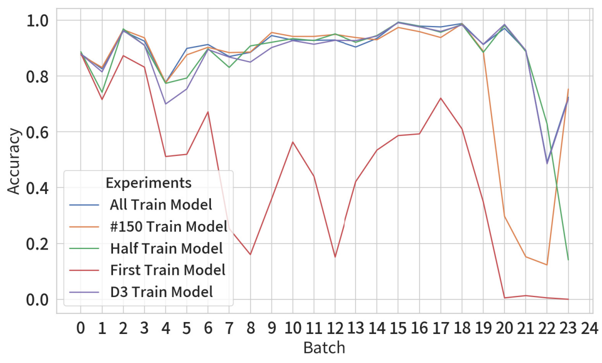

Fig. 8 Accuracy change of retrained models (ArgasLab Dataset).

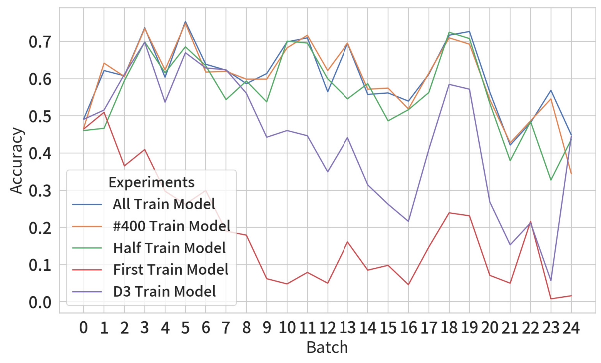

Fig. 9 Accuracy change of retrained models (AndroZoo Dataset).

a concept-drift is detected (*#150 Train Model* and *#400 Train Model*) requires less training and is comparable in accuracy to the model based on retraining every time (*All Train Model*). Third, the detection of concept-drift by our method is comparable to the state-of-the-art concept-drift detection algorithm. In short, our results show that the concept-drift detection of our method and the retraining based on it can be performed at the appropriate time.

4.6 Repackaging Malware

We evaluated the accuracy of the clustering with a particular focus on malware generated by repackaging. We identified 24 families in the ArgasLab dataset that are known to be generated by repackaging legitimate apps based on a historical analysis and evaluated our method using a sub-dataset of 1,963 malware belonging to these families. **Table 7** shows the clustering scores calculated through the same procedure shown in Section 4.4.

The malware generated by repackaging contains features and source code from authentic and benign Android apps and is gen-

Table 7 Clustering evaluation for repackaging malware.

Used Feature	Evaluation Metric	Avg. Score±SD
Original Extracted Feature	Avg. percentage of the most frequent family	0.870± 0.024
(unsupervised)	homogeneity_score	0.704± 0.026
UMAP Embedded Feature	Avg. percentage of the most frequent family	0.933± 0.006
(Supervised)	homogeneity_score	0.898± 0.006

erally difficult to detect and classify. Table 7 shows that the proposed method can classify repackaged malware although the accuracy is slightly reduced.

4.7 Observing New Malware Families

We show that the appearance of a new malware family can be observed by creating a family tree. In the unsupervised learning method, as shown in Fig. 2 which indicates the number transition of detections based on the malware family, we can see in particular that the "Fusob" family appeared in large numbers within a short period of time after 2015. In our family tree, shown in Fig. 4, the pink nodes at the bottom correspond to the Fusob family. These pink nodes emerge suddenly from one batch to form a family tree, which is consistent with the actual ground truth results in our dataset. Thus, our family tree allows us to know that a new malware family, with less relation to existing malware families, has emerged at some point in time. When using the UMAP method, we confirmed that the emergence of this Fusob family was detected as a concept-drift by the automatically generated family tree and that the model was subsequently retrained to maintain the accuracy of the model, in the experiments described in Section 4.5.2.

4.8 Benign Apps

Although our proposed method was designed to create a family tree for malware APKs, we evaluated it to show that it can also be applied to the creation of a family tree for benign APKs. We also used the version information of the benign application to discuss the implications of the family tree.

In this evaluation, we first evaluate whether a benign application A with some multiple versions can be connected as a family tree using the same features as malware. Next, we generate features and create a family tree with multiple versions of the benign application B, and we examine what information is reflected in the family tree. The benign applications used in this evaluation were obtained from the Google Play Store retroactively from the latest version, referring to the method proposed by Yasumatsu et al. [22].

4.8.1 Connections Inside Families

The family tree generated by the proposed method should connect clusters of the same malware family to each other and represent them as a single family line. In this experiment, instead of using the same malware family, we used benign APK files of multiple versions of a benign application A to evaluate whether the families were connected. For a single benign application A, we extracted the features of each version and embedded them using a trained UMAP model, and then we created a family tree. We used 122 multiple versions of the benign application A, from version 2.0.0, to version 6.40.1. Since the AndroZoo dataset contains newer and more variant malware families, the same features

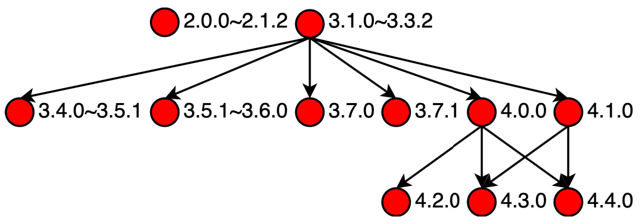

Fig. 10 Example of benign APK connection.

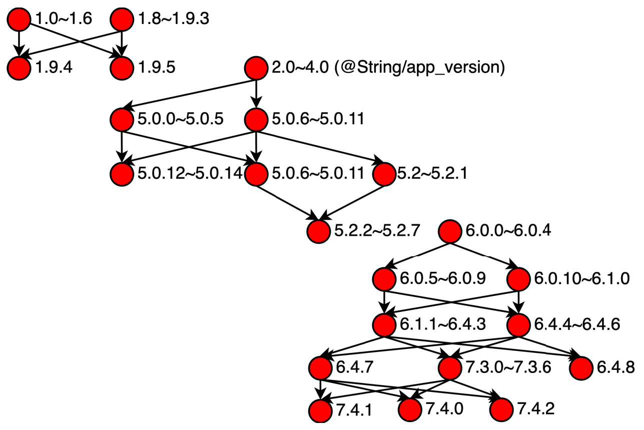

Fig. 11 Family tree of single benign application.

as the AndroZoo dataset were extracted, and then the extracted features were embedded using the UMAP model trained on the AndroZoo dataset.

Result **Figure 10** shows an example of a generated family tree with the threshold $\theta = 0.1$, which was determined to be appropiate in this study. The label of each node shows the version information contained in each cluster of the family tree. This version information is extracted from `AndroidManifest.xml`.

Discussion We found that all the clusters except for the clusters before version 2.1.2 were connected, and the family lines were connected to each other within the same family. In addition, there was more than three times the difference in the size of the APK file between version 2.1.2 and version 3.1.0 or later, where the family tree is broken. The internal structure of the application, such as the libraries used was also found to be significantly different. We also performed a dynamic analysis to confirm the difference between version 2.1.2 and version 3.1.0 or later. As a result, we were able to confirm that the GUI and functions of the applications differed greatly.

4.8.2 Family Tree of Single Benign Applications

Result We generated features for all 83 versions of a certain benign application B (from version 1.0 to version 7.4.2) and gave them the same labels in UMAP to create a family tree. **Figure 11** shows the example of a generated family tree with the threshold $\theta = 0.1$.

Discussion From the figure, we can see that the family tree reflects the version information and is connected in the order of each version. However, there are places where the family tree is disconnected, even though it is a single application. First, there is a family tree whose major version is 1.0, and a family tree whose version information is defined as "@string/app_version". For those versions whose version information is defined as "@string/app_version", referring to the string definition from the actual file, we found that the major version is assigned between 2.0 and 4.0. As represented by the version definition, we can see that the family tree is disconnected where the structure of the application has changed significantly. There is also a family line starting with major version 6.0. When we performed dynamic analysis and compared version 5.2.7 and version 6.0.0, we found that the application GUI was very different. We also found that the version of the Android SDK used has increased from 19 to 22.

This result shows that a family tree can also reflect the characteristics of a family line and suggests the "evolution" such as changes in the structure of the application.

5. Limitations

We will now discuss three limitations of our method. First, there are certain cases in which we cannot accurately obtain the permission information. For example, if the permission information is defined by the runtime permission, and is not written in `AndroidManifest.xml`, then our method cannot extract the permission-based features. Second, there are cases in which the API call information cannot be obtained. Specifically, in the case of an app that uses the API dynamically through reflection, the current implementation of our method does not allow us to obtain the API. Third, because our method does not analyze the code implemented in the native code, this case is outside the scope of our study. Fourth, in the case of repackaging malware, our method has slightly lower classification accuracy as discussed in Section 4.6. Our method mainly targets only Android malware having malicious code.

6. Related Work

We summarize related work from four perspectives: visualization of relationships among malware, detection of Android malware, family classification of Android malware, and concept drift detection for Android malware detection.

6.1 Visualization of Relationships among Malware

Oyen et al. [23] proposed a method for creating a family tree for Windows malware. Their method uses a Bayesian network to generate a family tree based on a directed acyclic graph (DAG) and generates a local family tree specific to each malware family. Whereas this method manually determines the parent-child relationship between nodes and generates a family tree focusing on individual families, our method simply feeds the dataset to a time series and automatically generates a global family tree that includes multiple families.

Erd'elyi et al. [24] and Ardimento et al. [25] proposed methods to visualize the similarity of malware in dendrograms using hierarchical clustering. These studies provide a phylogenetic tree using a dendrogram, although the hierarchy is based on the similarity of samples and does not reflect time-series information as in our approach.

Jang et al. [26] proposed a system of software lineage inference system for Windows malware and goodware binaries. Haq et al. [27] proposed a method for creating a lineage by entering a set of Windows malware for each family. Although the concept of lineage is similar to that of our family tree, our method does not create a series limited to a certain family or software, as these methods do.

In summary, our method differs from theirs in three major points: (1) we create a global family tree that includes multiple malware families, (2) we also classify families within the tree, and (3) we target Android malware.

6.2 Detection of Android Malware

We outline some typical studies that detect Android malware. Aafer et al. [6] proposed DroidAPIMiner, an Android malware detection system based on the features of API information which performs binary classification of malware and non-malware using the k-nearest neighbor method. Arp et al. [7] proposed a system called Drebin that detects malware by performing binary classification using a support vector machine (SVM) based on static features extracted from API calls and manifest files. Mariconti et al. [8] proposed a system called MaMaDroid that uses a Markov chain based on abstracted method calls to detect the behavior of Android malware. Suarez-Tangil et al. [28] proposed DroidSieve, a fast Android malware detection system that employs static features that are especially resistant to obfuscated Android malware. McLaughlin et al. [29] proposed a method to detect Android malware by applying deep convolutional neural network (CNN) to a sequence of opcodes disassembled from Android APKs. Kim et al. [30] proposed an Android malware detection framework that applies multimodal deep learning using various features that can be extracted from Android APKs.

In summary, all of the above studies are complementary to our method. In other words, by inputting the Android malware identified in these studies into our proposed method, we can produce a family tree of their relationships.

6.3 Family Classification of Android Malware

So far, the previous studies have mainly focused on binary classification of Android malware. In this section, we summarize the studies on classification and clustering focusing on malware families that further break down Android malware. Yang et al. [31] proposed DroidMiner, which can capture the behavioral patterns of Android malware using behavioral graph representation from known Android malware and diagnose which malware family it belongs to. Cai et al. [32] proposed DroidCat, which uses features obtained by running Android APKs to determine whether the input APK is malware and if so, which malware family it should be classified into. Mirzaei et al. [9] proposed AndrEnsemble, a system to identify Android malware families by ensembling sensitive API calls extracted by aggregating the call graphs of Android malware belonging to different families. Very recently, Zhang et al. [2] proposed a classification method based on the clustering results. This method estimates the ground truth label based on the detection results of VirusTotal and uses the similarity of the malware source codes and manifest information such as permissions in addition to the detection results of VirusTotal, to generate vectors using deep learning and apply clustering.

In summary, unlike our method, these methods cannot detect changes in malware over time or trace the detection results back in time and cannot create a global family tree containing multiple Android malware families.

6.4 Concept Drift Detection for Android Malware Detection

Pendlebury et al. [10] and Jordaney et al. [33] focused on concept-drift in machine learning-based Android malware detection. Especially, Pendlebury et al. showed that several previous studies on detecting Android malware did not sufficiently consider such conceptual drift, and the detection accuracy was reduced when the detection algorithm was tested under realistic time-series-based experimental conditions.

In our study, concept-drift detection was a secondary effect of our proposed family tree. Thus, the aim of these studies is significantly different from ours, which creates a family tree based on a time-series malware dataset.

7. Conclusion

We propose a new method for creating a family tree that is based on the time-series changes in Android malware. Our evaluation using two actual Android malware datasets shows the validity and effectiveness of our family tree. By creating a family tree using our method, we can obtain a significant amount of information regarding new malware and its trends. We hope that our method can be applied for a more efficient analysis of ever-increasing Android malware APKs and as a useful threat intelligence approach linked to historical information.

Acknowledgments This work was supported in part by the Japan Society for the Promotion of Science through Grants-in-Aid for Scientific Research (C) (20K11800). We thank Naoki Fukushi for the valuable discussions he provided.

References

[1] Nomura, K., Chiba, D., Akiyama, M. and Uchida, M.: Auto-creation of Android Malware Family Tree, *Proc. IEEE ICC* (2021).

[2] Zhang, Y. et al.: Familial Clustering for Weakly-Labeled Android Malware Using Hybrid Representation Learning, *IEEE Trans. Information Forensics and Security* (2020).

[3] Lu, J. et al.: Learning under Concept Drift: A Review, *IEEE Trans. Knowledge and Data Engineering* (2019).

[4] Tounsi, W. and Rais, H.: A survey on technical threat intelligence in the age of sophisticated cyber attacks, *Computers & Security* (2018).

[5] Tam, K. et al.: The evolution of android malware and android analysis techniques, *ACM CSUR* (2017).

[6] Aafer, Y. et al.: DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android, *Proc. SecureComm* (2013).

[7] Arp, D. et al.: DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket, *Proc. NDSS* (2014).

[8] Mariconti, E. et al.: MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models, *Proc. NDSS* (2017).

[9] Mirzaei, O. et al.: AndrEnsemble: Leveraging API Ensembles to Characterize Android Malware Families, *Proc. ACM AsiaCCS* (2019).

[10] Pendlebury, F. et al.: TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time, *Proc. USENIX Security* (2019).

[11] MobSF, available from ⟨https://github.com/MobSF/Mobile-Security-Framework-MobSF⟩.

[12] Arora, A. et al.: PermPair : Android Malware Detection Using Permission Pairs, *IEEE Trans. Information Forensics and Security* (2019).

[13] MobSF-android_apis.yaml, available from ⟨https://github.com/MobSF/Mobile-Security-Framework-MobSF/blob/master/StaticAnalyzer/views/android/rules/android_apis.yaml⟩.

[14] McInnes, L., Healy, J. and Melville, J.: UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction, ArXiv e-prints 1802.03426 (2018).

[15] Pelleg, D. and Moore, A.W.: X-means: Extending K-means with Efficient Estimation of the Number of Clusters, *Proc. ICML* (2000).

[16] Wei, F. et al.: Deep Ground Truth Analysis of Current Android Malware, *Proc. DIMVA* (2017).

[17] Allix, K., Bissyandé, T.F., Klein, J. and Le Traon, Y.: AndroZoo: Collecting Millions of Android Apps for the Research Community, *Proc. MSR* (2016).

[18] VirusTotal, available from ⟨https://www.virustotal.com/⟩.

[19] Sebastián, S. and Caballero, J.: AVclass2: Massive Malware Tag Extraction from AV Labels, *Proc. ACSAC* (2020).

[20] Rosenberg, A. and Hirschberg, J.: V-Measure: A Conditional Entropy-Based External Cluster Evaluation Measure, *Proc. EMNLP-CoNLL* (2007).

[21] Gözüaçık, O. et al.: Unsupervised Concept Drift Detection with a Discriminative Classifier, *Proc. ACM CIKM*, pp.2365–2368 (2019).

[22] Yasumatsu, T. et al.: Understanding the Responsiveness of Mobile App Developers to Software Library Updates, *Proc. ACM CODASPY*, pp.13–24 (2019).

[23] Oyen, D. et al.: Bayesian Networks with Prior Knowledge for Malware Phylogenetics, *Proc. AAAI Workshop: Artificial Intelligence for Cyber Security* (2016).

[24] Erd'elyi, G. and Carrera, E.: Digital genome mapping: Ad-vanced binary malware analysis, *Proc. Virus Bulletin Conference* (2004).

[25] Ardimento, P. et al.: Malware Phylogeny Analysis using Data-Aware Declarative Process Mining, *Proc. IEEE EAIS* (2020).

[26] Jang, J. et al.: Towards Automatic Software Lineage Inference, *Proc. USENIX Security*, pp.81–96 (2013).

[27] Haq, I.U. et al.: Malware lineage in the wild, *Comput. Secur.*, Vol.78, pp.347–363 (online), DOI: 10.1016/j.cose.2018.07.012 (2018).

[28] Suarez-Tangil, G. et al.: DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware, *Proc. ACM CODASPY*, pp.309–320 (2017).

[29] McLaughlin, N. et al.: Deep Android Malware Detection, *Proc. ACM CODASPY*, pp.301–308 (2017).

[30] Kim, T. et al.: A Multimodal Deep Learning Method for Android Malware Detection Using Various Features, *IEEE Trans. Inf. Forensics Secur.*, Vol.14, No.3, pp.773–788 (2019).

[31] Yang, C. et al.: DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications, *Proc. ESORICS*, Vol.8712, pp.163–182 (2014).

[32] Cai, H. et al.: DroidCat: Effective Android Malware Detection and Categorization via App-Level Profiling, *IEEE Trans. Inf. Forensics Secur.*, Vol.14, No.6, pp.1455–1470 (2019).

[33] Jordaney, R. et al.: Transcend: Detecting Concept Drift in Malware Classification Models, *Proc. USENIX Security* (2017).

Editor's Recommendation

In this paper, the authors propose a new method for automatically creating a "family tree" of Android malware that can represent how the newly detected Android malware is related to existing Android malware and its families and how they have changed over time. In recent years, Android malware continues to increase in number and type or family over time. This paper gives a great impact to readers in this research field and thus is selected as a recommended paper.

(Program Chair of anti Malware engineering WorkShop 2020,
Masatsugu Ichino)

Kazuya Nomura received his B.E. and M.E. degrees in Computer Science from Waseda University in 2018 and 2021. His research interest covers Cyber Security. He is now with NTT Security (Japan) KK, Tokyo, Japan.

Daiki Chiba is currently a manager at NTT Security (Japan) KK, Tokyo, Japan. He received his B.E., M.E., and Ph.D. degrees in computer science from Waseda University in 2011, 2013, and 2017. Since joining Nippon Telegraph and Telephone Corporation (NTT) in 2013, he has been engaged in research on cyber security through data analysis. He is a member of IEEE and IEICE.

Mitsuaki Akiyama received his M.E. and Ph.D. degrees in information science from Nara Institute of Science and Technology, Japan in 2007 and 2013. Since joining Nippon Telegraph and Telephone Corporation (NTT) in 2007, he has been engaged in research and development on cybersecurity. He is currently a Senior Distinguished Researcher at NTT Social Informatics Laboratories. His research interests include cybersecurity measurement, offensive security, and usable security and privacy. He is a member of the IEEE, IPSJ, and IEICE.

Masato Uchida is currently a Professor in the Department of Computer Science and Engineering, School of Fundamental Science and Engineering, Waseda University, Tokyo Japan. He received his B.E., M.E. and Ph.D. degrees from Hokkaido University, Hokkaido, Japan, in 1999, 2001, and 2005, respectively. In April 2001, he joined NTT Service Integration Laboratories, Tokyo, Japan. From August 2005 to March 2012, he has been an Associate Professor in the Network Design Research Center, Kyushu Institute of Technology, Fukuoka, Japan. From April 2012 to March 2015, he has been an Associate Professor in the Department of Electrical, Electronics and Computer Engineering, Faculty of Engineering, Chiba Institute of Technology, Chiba, Japan. From April 2015 to March 2016, he has been a Professor in the same department. From April 2016 to March 2017, he has been a Professor in the Department of Information and Communication Systems Engineering of the same university. He has been engaged in research on data science and machine learning with application to various fields in computer science including information networking and information security. He is a member of the IEEE, ACM, IPSJ, and IEICE.