COMPUTERS & SECURITY 80 (2019) 120–133

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

A scalable and extensible framework for android malware detection and family attribution

*Li Zhang**, *Vrizlynn L.L. Thing*, *Yao Cheng*

*Cyber Security Cluster, Institute of Infocomm Research, A*STAR, Singapore*

ARTICLE INFO

Article history:
Received 23 April 2018
Revised 28 August 2018
Accepted 3 October 2018
Available online 9 October 2018

Keywords:
Android malware detection
Malware family attribution
Online classifier
Incremental learning

ABSTRACT

The threat from the rampant Android malware has reached an alarming scale, where there are millions of new malware samples pouring into the application markets every year. In this paper, we present a new method that can efficiently detect the malware and attribute it to the corresponding malware family with a high accuracy. A multi-level fingerprint is firstly extracted from the application by using n-gram analysis and feature hashing. Each of its sub-fingerprints is then input to a dedicated online classifier. Based on the confidence scores from the classifiers and our devised combination function, the final decision will be made on whether the application is benign or malicious or in the scenario of family attribution, which malware family it belongs to. To the best of our knowledge, this is the first method developed based on the combination of n-gram analysis and online classifiers. The incremental learning enabled by the online classifiers facilitates our method to scale well even for a huge number of applications and adapt easily to different characteristics in new applications. The parallelized design not only magnifies the impact of distinguishing features in each sub-fingerprint but also allows our method to be extended, where additional application features can be added as extra sub-fingerprints. Extensive experiments were performed. The results show that our method achieved malware detection accuracy of 99.2% on a benchmark dataset with more than 10,000 samples and 86.2% on a dataset with more than 70,000 in-the-wild samples. Regarding malware family attribution, our method achieved an accuracy of 98.8% on the top 23 malware families of Drebin dataset.

1. Introduction

Smart phones have been an integral part of our daily life. We use them to make calls, send messages, check emails, take photos, and surf the Internet every day. In the year 2016 alone, a total of 1.5 billion units of smart phones were purchased by the end users, among which 84.8% of devices were powered by Android (Gartner, 2016). The popularity of Android devices has, however, made them the most attractive targets for cybercriminals. It is reported that Android developed malware accounts for 97% of all mobile malware (PulseSecure, 2014) and more than 750,000 new Android malware samples were discovered during the first quarter of 2017 (Lueg, 2017).

In fact, various security measures have been deployed in Android to harden against the installation of malware. One of the most important measures is the permission-based security model, which allows end users to manage application permissions and hence restricts the application's accessibility to sensitive resources. However, a large portion of Android users tend to blindly grant the requested permissions, thereby weakening the theoretically-sound protection. A recent massive attack is mounted by a malware dubbed FalseGuide, which was hidden in over 40 applications on Google Play from

* Corresponding author.
E-mail addresses: zhang-li@i2r.a-star.edu.sg (L. Zhang), vriz@i2r.a-star.edu.sg (V.L.L. Thing), cheng_yao@i2r.a-star.edu.sg (Y. Cheng).
https://doi.org/10.1016/j.cose.2018.10.001
0167-4048/

November 2016 to April 2017 and infected more than 2 million Android devices (The Hacker News, 2017).

The alarming threat of Android malware has driven active research for good malware detection approaches. A number of methods propose to use discriminating features from the applications, such as requested permissions (Sanz et al., 2013), API calls (Aafer et al., 2013), dynamic behaviors (Afonso et al., 2015; Burguera et al., 2011), or a combination of multiple features (Arp et al., 2014; Lindorfer et al., 2015; Yerima et al., 2015), and take advantage of similarity comparison metrics or machine learning algorithms to distinguish between the malware and benign applications. These methods typically rely on expert analysis to predetermine specific features to be processed, which may exclude other important features in the applications. Besides, with the discriminating features explicitly defined, it is easier for malware writers to update their malware accordingly so as to evade detection (Abou-Assaleh et al., 2004).

On the other hand, malware detection methods can also be based on n-gram analysis. With the captured features being implicit in the extracted n-grams, it would be harder to fool the detection algorithm. Several methods (Canfora et al., 2015; Kang et al., 2016) propose to extract n-gram opcodes from the disassembled application byte code and use machine learning algorithms to automatically distill features from the n-grams. In addition to the instruction n-grams, a recent work (Karbab et al., 2016) investigated the benefit of considering extra information in the application APK file, such as a permission vector from the XML file, byte n-grams of the DEX file, as well as byte n-grams of the whole APK file. In particular, each set of these features forms a sub-fingerprint of the application. The fingerprint of the application under test (AUT) is then compared with those of collected malware in a database by using Jaccard Similarity. Based on the similarity score and a static threshold, the application will be classified accordingly. There are several drawbacks in this method. Firstly, the AUT needs to be compared with all the entries in the malware database, which prevents this method from scaling well for a large malware database. Secondly, all the feature elements are considered with the same weight, undermining the impact of the more interesting features. Lastly, a static threshold is required to classify the application, which is hard to determine in practice.

In this paper, we firstly propose a tailored design of the application fingerprint, where n-grams of the XML strings are used instead of the permission vector. This way, besides permissions, additional important features in the XML file, such as utilized hardware components, application components, and intents, will also be considered for malware detection. Instead of using Jaccard similarity-based comparison and a static threshold as in Karbab et al. (2016), a parallel online classifier-based approach is used in our method. That is, a specific online classifier is trained for each sub-fingerprint. Apart from automatically increasing the weight of the more discriminating features, the utilized online classifier enables incremental learning, which brings in two obvious advantages. The first one is that our method can better handle the huge number of malware samples, hence ensuring its scalability. The second one is that the built model in our method can be easily updated with new malware samples. This is in contrast to methods using batch learning-based classifiers, where the classifier needs to be re-trained with all the samples.

Besides malware detection, our method can also be used for malware family attribution. Malware family attribution is an important part of threat assessment and mitigation planning. An efficient automatic family attribution of the malware will allow malware analyst to focus on new malware instead of variants of existing malware families. We highlight that the main goal of our proposed method is to detect variants of existing malware families, which make up more than 98% of new malware samples (Sun et al., 2017).

Both malware detection and family attribution belong to classification problems, where the former is a two-class problem and the latter is a multi-class one. To classify an AUT, the decision scores from each classifier will be fused by our devised combination function which will then output the final decision of the application being in a specific class. In the scenario of malware detection the specific class will be either malware or benign application, while in the scenario of malware family attribution it will be a particular malware family. As will be shown in the experimental results, for malware detection, our method achieved an accuracy of 99.2% on a benchmark dataset with more than 10,000 samples and an accuracy of 86.2% on an in-the-wild dataset with more than 70,000 samples. For malware family attribution, our method obtained an accuracy of 98.8% on the top 23 malware families (family sizes varying from 883 to 21 samples) of Drebin dataset.

To the best of our knowledge, we propose the first combination of n-gram analysis and online classifiers to tackle the problem of Android malware detection and family attribution. The primary contributions of this work are as follows:

- We propose the use of n-grams of XML strings to generate android application sub-fingerprint. As to be shown by the experimental results, the XML string-based sub-fingerprint achieves much better accuracy compared to the permission vector-based sub-fingerprint. This sub-fingerprint can be used together with other sub-fingerprints such as those derived from n-grams of the DEX file, the disassembled DEX file, and the whole APK file to further enhance the performance.
- We propose a parallel classifier-based model, where a dedicated classifier is assigned for each sub-fingerprint. There are three merits: firstly, the machine learning-based model can automatically distill the important features from the sub-fingerprints, hence avoiding being deceived by the unrelated features. Secondly, compared to the peer-matching approach in Karbab et al. (2016), it removes the need to compare the AUT with each malware in the database and hence greatly improves the efficiency. Thirdly, compared with simply concatenating all the sub-fingerprints and using one classifier, our method is featured by better accuracy and shorter training time.
- The online classifiers used in our method can incrementally learn from new samples, which not only avoids the need of a huge memory that is often required by batch learning-based methods but also enables the trained classifiers to be adaptable to population drift of malware (Singh et al., 2012) and maintain a good classification accuracy.

The paper is organized as follows: Section 2 introduces Android application basics and describes the techniques used to generate the application fingerprint. Section 3 illustrates the proposed method which is based on the multi-level sub-fingerprints and parallel incremental learning. In Section 4, experimental results and some discussion about the method are presented. A more detailed review of related works is in Section 5, and the paper is concluded in Section 6.

2. Android application basics and fingerprint generation

2.1. Android application basics

Android applications are written mainly in Java, which are then compiled along with data and resource files into an archive file called Android Package Kit (APK). The APK is the file distributed in the application market and the one used for application installation.

There are four different types of application components, i.e., *activity*, *service*, *broadcast receiver*, and *content provider*. Communications among these components are achieved using a messaging object called *intent*. In Android, the application must declare all its components in an XML manifest file inside the APK. *Intent filters*, which declare capabilities of the components, are often also included. Additional information declared in the XML file involve user permissions the application requires (e.g., `CALL_PHONE`, `SEND_SMS`, and `INTERNET`), the minimum API level, as well as the hardware and software features to be used by the application (e.g., GPS, camera or multi-touch screen).

Besides the XML manifest file, an APK comprises a *DEX* file (sometimes multiple DEX files) which contains all the classes and is to be executed in its own instance of virtual machine (i.e., Dalvik Virtual Machine (DVM) or Android Runtime (ART)), a *lib* folder which contains the compiled code specific to the processor software layer, a *META_INF* folder which contains application certificate, list of resources and SHA-1 digest of all resources, etc., and a *resources.arsc* file which contains pre-compiled resources, as well as a *res* folder which contains resources that are not compiled into the *resources.arsc* file.

2.2. Application fingerprint generation

The application fingerprint in our method is generated by extracting multi-level features (Karbab et al., 2016; Masud et al., 2008) from the application. It is composed of several sub-fingerprints from different contents of the application, such as the XML (manifest) file, the DEX file, the disassembled DEX file (in short, the *assembly* file), and the whole APK file. Each sub-fingerprint is created using n-gram analysis and feature hashing.

The n-gram analysis (Cavnar and Trenkle, 1994) extracts a sequence of n-item features from a given file. For the DEX and APK file, n-grams are extracted with the granularity level of per byte, while for the assembly file, the granularity level is per instruction. For the XML file, which is in the form of binary XML in the APK, we convert it to a human readable format and then extract n-gram XML strings. Unlike the method in Karbab et al. (2016), which uses only the requested permissions in the XML file to derive a sub-fingerprint, the n-gram XML strings in our method convey more information about the application, such as the application components, filtered intents, and the required hardware and software features. All these information are helpful to better identify a malware. For example, some malware will list a `BOOT_COMPLETED` intent in the XML file such that malicious activities will be triggered immediately after the smart phone is re-booted. Table 1 lists some representative XML strings extracted from a DroidKungFu3 malware sample. Note that the strings are reorganized into 8 categories just for better readability.

Table 1 – Representative XML strings extracted from a DroidKungFu3 malware sample

Meta-data	`com.livegame.metadata.COPYRIGHT`
	`com.livegame.metadata.AUTHORS`
	`com.livegame.metadata.WEBSITE_LABEL`
	`com.livegame.metadata.WEBSITE_URL`
	`com.livegame.metadata.EMAIL`

Permission	`android.permission.WRITE_EXTERNAL_STORAGE`
	`android.permission.INTERNET`
	`android.permission.READ_EXTERNAL_STORAGE`
	`android.permission.CHANGE_WIFI_STATE`
	`android.permission.RECEIVE_BOOT_COMPLETED`

Intent-filter	`com.livegame.action.PICK_FILE`
	`com.livegame.action.PICK_DIRECTORY`
	`android.intent.action.GET_CONTENT`
	`android.intent.category.OPENABLE`
	`android.intent.action.BOOT_COMPLETED`

Activity	`com.livegame.distribution.EulaActivity`
	`com.google.update.Dialog`
	`com.google.ads.AdActivity`
	`com.adwo.adsdk.AdwoSplashAdActivity`
	`com.adwo.adsdk.AdwoAdBrowserActivity`

Service	`com.google.update.UpdateService`
Content provider	`com.livegame.filemanager`
Broadcast receiver	`com.google.update.Receiver`
Feature	`android.hardware.touchscreen`

The n-gram size n is a parameter which controls the dimensions of the underlying feature space for representing an application. Its impact on classification accuracy of Andorid applications have been studied in several works (Canfora et al., 2015; Hanna et al., 2012; Kang et al., 2016). The chosen n-gram size should achieve a good balance between multi-dimensional representation of the application and reasonable number of unique features.

To generate each sub-fingerprint, we convert the corresponding sequence of n-grams to a bit-vector through feature hashing (Weinberger et al., 2009). Commonly used feature vectorization strategies such as those based on tf-idf (Larson, 2010) have to keep an in-memory mapping from n-grams to feature indexes. Besides consuming lots of memory for a large dataset, the mapping requires a full pass over the whole dataset. Hence, such strategies are not suitable for our incremental learning based model. In contrast, feature hashing

applies a hash function to the n-grams to determine their indexes directly, which is a high-speed and memory-efficient alternative and helps significantly reduce the high-dimensional feature spaces and accelerate malware triage (Jang et al., 2011). In our method, each hashed n-gram has a corresponding bit in the bit-vector to indicate whether the n-gram exists (being '1') or not (being '0'). That is, we consider only whether a specific n-gram exists or not instead of its integer counts. The bit-vector length l is another crucial parameter, which determines the approximation error (more collisions for smaller l) and the processing efficiency (inversely proportional to l).

Hanna et al. (2012) analyzed 30,000 Android applications to determine the two parameters above. It was found that for n-grams of the assembly file, the best trade-off between accuracy and processing efficiency was achieved when $n = 5$ and $l = 240,007 \approx 2^{18}$. In our implementation, we used the same values for these two parameters for all sub-fingerprints and achieved good results, as will be shown in the experiments. Nonetheless, as discussed in Section 4.4, different n-gram sizes and bit-vector lengths can be used for each sub-fingerprint to achieve the best trace-off.

3. The parallel online classifiers based approach

The application fingerprint, which consists of multiple sub-fingerprints (in the form of bit-vectors), is an abstracted representation of the application. Each bit in the bit-vector can be deemed a feature. Similar applications share similar bit-vector patterns, while the divergent ones would have very different bit-vectors. We use machine learning algorithms to automatically distill important features that facilitate the discrimination between malware and benign applications or among different malware families. The bit-vector form of the application sub-fingerprints allows them to be efficiently processed by machine learning algorithms.

Each bit-vector can be represented as $\vec{BV} \in \{0, 1\}^l$, where l is the length of the bit-vector. As $\vec{BV}$ is typically a sparse vector of fixed length with bits being either '0' or '1', in our method, instead of storing the bit-vector itself, the indexes of the (usually rare) '1' bits are stored. This helps greatly reduce the storage overhead and at the same time ensures that the bit-vector can be easily recovered.

Instead of concatenating the bit-vector of each sub-fingerprint into a single bit-vector and then use it to train one classifier, in our method, a dedicated classifier is used for each sub-fingerprint. An overview of the proposed method is shown in Fig. 1, where four sub-fingerprints (derived from the XML file, DEX file, assembly file and the whole APK file, respectively) are used. As will be shown by the experimental results in Section 4, such a design helps magnify the impact of the possibly few discriminating features and hence improve the classification accuracy.

Regarding the machine learning algorithm, we choose to use the online passive-aggressive (PA) classifier (Crammer et al., 2006). An online classifier operates in rounds, in each of which it receives a sample or a mini-batch of samples as input for prediction and then receives the true label of the sample for updating the model. As the trained model can incrementally learn from the stream of incoming samples, this type of learning is called *incremental learning*.

The PA classifier is a linear classifier, which works well for problems with a large number of features and can reach accuracy levels comparable to non-linear classifiers while taking less time to train and use (Yuan et al., 2012). Note that the sparse bit-vector form of the sub-fingerprints essentially projects the application features into a high-dimensional space. According to Cover's Theorem (Cover, 1965), these transformed features would have a high probability of being linearly separable. For the two-class malware detection problem, the PA classifier fits a decision hyperplane between malware and benign applications. Suppose on round i a sub-fingperint of the sample is $\vec{BV}_i \in \{0, 1\}^l$ and the label of the sample is $y_i \in \{-1, +1\}$, where -1 corresponds to benign application and $+1$ to malware. The built model of the PA classifier is based on a vector of feature weights denoted as $\vec{\omega} \in \mathbb{R}^l$, with the vector length l being the same as that of $\vec{BV}_i$. For round i, the existing model can be represented as $\hat{y}_i = sign(\vec{\omega}_i \cdot \vec{BV}_i)$, where the predicted label for the sample $\hat{y}_i$ is determined based on the sign of the inner product of $\vec{\omega}_i$ and $\vec{BV}_i$. The margin of the sample $(\vec{BV}_i, y_i)$ with respect to the existing model is given by $y_i(\vec{\omega}_i \cdot \vec{BV}_i)$ and the loss at the sample is defined as shown in Eq. (1). In other words, a margin not less than 1 means that a correct prediction has been made and the loss is 0.

For PA classifier, the model is only updated if the loss is not 0 (i.e., $1 - y_i(\vec{\omega}_i \cdot \vec{BV}_i)$ when the margin is less than 1). The objective is to change the feature weights as minimal as possible but with the wrongly predicted sample classified correctly in the updated model. That is, the updated weight vector $\vec{\omega}_{i+1}$ is the solution to the constrained optimization problem in Eq. (2). As detailed in Crammer et al. (2006), this optimization problem has a simple closed form solution and can be efficiently solved.

For the multi-class malware family attribution problem, the one-versus-all (OvA) strategy can be used, where a single binary classifier described above is trained for each class, with members of the specific class labeled as $+1$ and all other samples as -1. The label of the sample is then predicted based on the classifier with the highest confidence score.

$$\ell(\vec{\omega}_i; (\vec{BV}_i, y_i)) = max\{0, 1 - y_i(\vec{\omega}_i \cdot \vec{BV}_i)\} \tag{1}$$

$$\vec{\omega}_{i+1} = \underset{\vec{\omega} \in \mathbb{R}^l}{\operatorname{argmin}} \frac{1}{2} \| \vec{\omega} - \vec{\omega}_i \|^2 \quad \text{subject to } \ell(\vec{\omega}; (\vec{BV}_i, y_i)) = 0 \tag{2}$$

Details about how the PA classifier is used in our method for training and classifying an AUT are presented below.

3.1. *The training stage*

The PA classifiers will be trained in parallel by the corresponding sub-fingerprints of the applications. In the scenario of malware detection, each sub-fingerprint is associated with a label of either *malware* or *beingn_app*, while in the scenario of malware family attribution, each sub-fingerprint is

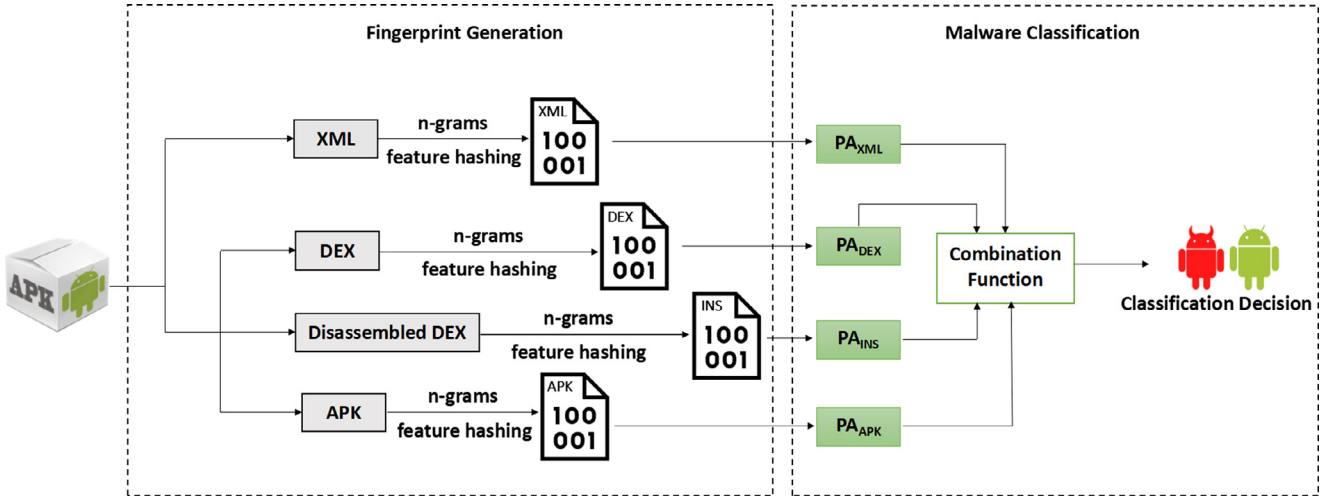


Fig. 1 – Android malware detection and family attribution with parallel PA classifiers.

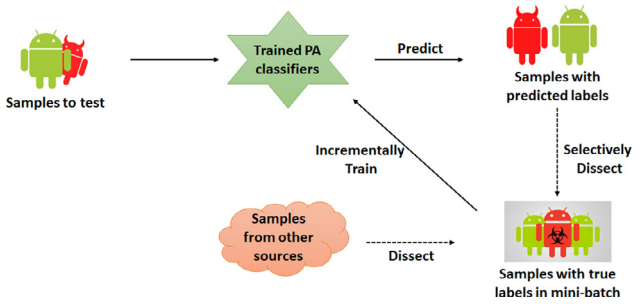


Fig. 2 – PA classifiers incrementally trained with new samples.

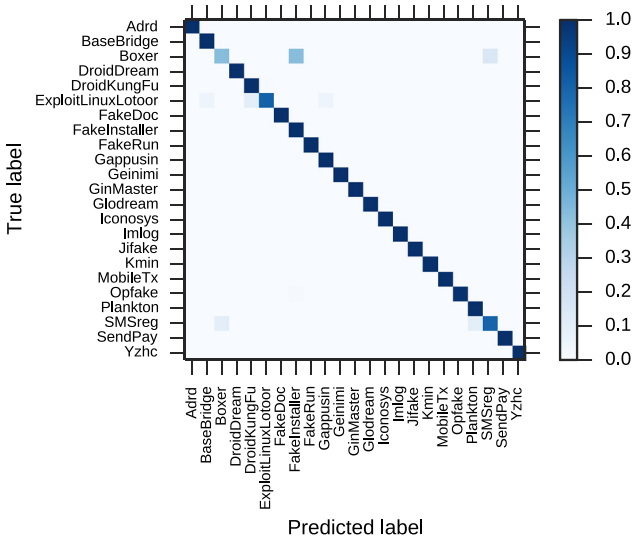


Fig. 3 – Normalized confusion matrix of the top 23 Drebin malware families using the assembly and xml-based sub-fingerpints and the combination function of average of scores.

associated with the corresponding malware family name, such as *AnserverBot*, *DroidKungFu3*, and *GoldDream*.

In our method, the training dataset of possibly massive size is divided into mini-batches (a.k.a. chunks), which are then fed to the model for training one by one. This way, it becomes feasible to learn from the huge number of applications that usually can not fit into system memory. Naturally, the mini-batch size should be determined based on the available memory space.

On the other hand, in practice, newly assessed malware or benign applications (i.e., with true labels) become available just gradually over time. These new applications can be accumulated and form new mini-batches before they are fed to the trained model. As depicted in Fig. 3, the model can continuously adapt itself to accommodate new application characteristics and hence maintain desired classification accuracy.

3.2. *The decision stage*

The decision function of each trained PA classifier will output a confidence score of the AUT being in a specific class. In the scenario of two-class malware detection, only the score for being a malware (i.e., class '1') will be generated, while in the scenario of multi-class malware family attribution, a confidence score is generated for each malware family. In both scenarios, a positive number means the AUT is predicted to be a member of the specific class, while a negative number means not.

Suppose a total of K classifiers have been trained for malware family attribution, denoted as PA_k, $k \in [1, K]$. Given an AUT, its sub-fingerprints are firstly generated, which are K bit-vectors represented as $\vec{BV}_k$. Each sub-fingerprint is then used as input to the corresponding classifier. The classifiers will output a list of scores indicating the confidence of the application being a member of the corresponding family. We denote the list of confidence scores from classifier PA_k as $\vec{S_k}$. If there are M different malware families, denoted as $C_1, \ldots C_M$, then cardinality of the list $|\vec{S_k}| = M$. The element in the score list $\vec{S_k}$ is denoted as $s_k(C_i)$, where $s_k(C_i) \in \mathbb{R}$ and $i \in [1, M]$. For example, if the sub-fingerprints are based on the XML file, the DEX file, the assembly file, and the whole APK file, respectively, the four

lists of confidence scores from the classifiers will be:

$$\begin{aligned}&[s_1(C_1), s_1(C_2), \ldots, s_1(C_M)]_{\mathbf{PA}_{XML}}\\&[s_2(C_1), s_2(C_2), \ldots, s_2(C_M)]_{\mathbf{PA}_{DEX}}\\&[s_3(C_1), s_3(C_2), \ldots, s_3(C_M)]_{\mathbf{PA}_{INS}}\\&[s_4(C_1), s_4(C_2), \ldots, s_4(C_M)]_{\mathbf{PA}_{APK}}\end{aligned} \tag{3}$$

The K list of confidence scores can be consolidated through different combination functions so as to make the final classification decision. The most intuitive way is *majority voting*, where the maximum value within each list of scores is picked up to make an intermediate decision (i.e., the malware family corresponding to the maximum value) and the K intermediate decisions will vote for the final decision. That is, for each classifier PA_k, the intermediate decision is the family C_j with $s_k(C_j) = max(\vec{S_k})$. If we represent the intermediate decision by classifier PA_k as a binary characteristic function shown in Eq. (4), the final decision made with the majority voting rule will be in the form shown in Eq. (5). In case of equal votes, we use the following way to remove uncertainty: the confidence scores supporting each equal vote will be summed and the vote with the largest sum will be the final decision. As the confidence scores are in the field of $\mathbb{R}$, in practice, there will be always a vote with a larger sum than other votes.

$$T_k(C_j) = \begin{cases}1, & \text{when } s_k(C_j) = max(\vec{S_k})\\0, & \text{otherwise}\end{cases} \tag{4}$$

$$\begin{aligned}&PA_{vote}(\vec{BV}) = C_j, \text{ if } T(C_j) = max_{i\in[1,M]}(T(C_i))\\&\text{where } T(C_i) = \sum_{k=1}^{K} T_k(C_i),\ i \in [1, M]\end{aligned} \tag{5}$$

Alternative combination functions such as the *average of scores* and *maximum of scores* can also be used, where the K scores for each family C_i are firstly combined (using *mean* or *max*) and the family corresponding to the maximum combined score will be the final decision of the classifiers. The former can be represented with Eq. (6), while the latter is in the form shown in Eq. (7). As the confidence scores are signed values, the combination function of *product of scores* is not as effective and hence not used in our method.

$$\begin{aligned}&PA_{avg}(\vec{BV}) = C_j, \text{ if } \bar{S}(C_j) = max_{i\in[1,M]}(\bar{S}(C_i))\\&\text{where } \bar{S}(C_i) = \frac{1}{K}\sum_{k=1}^{K} s_k(C_i),\ i \in [1, M]\end{aligned} \tag{6}$$

$$\begin{aligned}&PA_{max}(\vec{BV}) = C_j, \text{ if } \widetilde{S}(C_j) = max_{i\in[1,M]}(\widetilde{S}(C_i))\\&\text{where } \widetilde{S}(C_i) = max_{k\in[1,K]} s_k(C_i),\ i \in [1, M]\end{aligned} \tag{7}$$

In the scenario of malware detection, each score list $\vec{S_k}$ just contains one instead of M scores. The three combination function described above are still applicable after some minor changes. The *majority voting* method now votes for whether there are more positive (indicating malicious) or negative (indicating benign) scores; if there are equal votes, the positive and negative confidence scores are each summed up and the final decision is made based on the sign of the sum with a larger absolute value. The *average of scores* method now simply averages the scores and the final decision is made based on the sign of the averaged value, while the *maximum of scores* method simply chooses the value with the maximum absolute value and the final decision is also made based on the sign of the chosen value. It may be noted that for malware detection the *majority voting* method and *average of scores* method are essentially the same and would always make the same decision.

Table 2 – The datasets used in our experiments

Dataset Category		Name	Size	Time of creation
Benchmark	Malware	AMGP	928	Aug. 2010–Oct. 2011
		Drebin	5560	Aug. 2010–Oct. 2012
In-the-wild	Malware	-	33,259	Jan. 2014–Aug. 2014
	Benign	-	37,224	Jan. 2014–Aug. 2014

4. Evaluation and discussion

4.1. Experimental setup and dataset

To generate the application fingerprint, the same open-source tools as in Karbab et al. (2016) were used. The *xxd* tool was used to create hex dumps of the APK and DEX file. The disassembler tool *dexdump*, provided by Android SDK, was used to create the assembly file, while the *aapt* tool, which is also from Android SDK, was used to dump out the XML strings (i.e., using the command `aapt dump xmlstrings`). The above tools were used together with data extraction tools such as *awk* and *grep* to generate the *n*-grams of each sub-fingerprint. The sequences of *n*-grams were then hashed to bit-vectors of length 2^{18} based on the occurrence of each *n*-gram by using the hashingVectorizer utility of scikit-learn toolkit (Pedregosa et al., 2011). The classifiers used in our experiments were also built from this toolkit. All the experiments were conducted on a desktop with Intel Core i7-6700 CPU @ 3.4 GHz (8 cores), 16 GB RAM, and 64-bit Ubuntu Linux 14.04 LTS.

Several sets of experiments were performed to evaluate the effectiveness and efficiency of the proposed method. The datasets used in our experiments are listed in Table 2. The benchmark datasets include the malware samples from Android Malware Genome Project (AMGP) (Zhou and Jiang, 2012) and Drebin (Arp et al., 2014), both of which are commonly used in the literature. Samples of the in-the-wild dataset[1] were collected by Narayanan et al. (2017, 2016) and created in a span of 224 days from 1 Jan. 2014 to 13 Aug. 2014. The reason for also using the in-the-wild dataset is that high-performance malware detection methods validated by the benchmark dataset are often not as effective in the wild (Allix et al., 2016). We would like to exhibit the use of our method in a real-world setting and examine its effectiveness in such a setting.

4.2. Evaluation on the AMGP dataset

As the multi-level fingerprint used in our method is inspired by the method proposed in Karbab et al. (2016), we firstly adopt the same experimental setup so as to facilitate comparison

[1] based on dataset at https://sites.google.com/view/casandrantu/dataset.

Table 3 – Top 8 malware families of android malware genome project used in evaluation

Malware Family name	Family size
AnserverBot	187
BaseBridge	122
DroidDreamLight	46
DroidKungFu3	309
DroidKungFu4	96
Geinimi	69
GoldDream	47
KMin	52

with the method. The malware samples used are from the AMGP dataset. A balanced malware dataset was created by randomly choosing 46 samples from each of the 8 largest families in the malware database (see Table 3), where 46 is the size of the 8*th* largest family *DroidDreamLight*. The balanced malware dataset is then concatenated with 46 randomly selected benign applications (labeled as *Benign_Apps*) to form the evaluation dataset to be used. The benign applications were downloaded from Google Play and created in the same period as the malware samples (i.e., from 2010 to 2011). To ensure they are indeed benign, the downloaded samples were checked using the VirusTotal service[2] and only those not flagged as malicious by any anti-virus scanner were kept.

In each evaluation, 70% of samples in each family of the evaluation dataset were randomly selected as the training dataset, while the remaining 30% samples were used for testing. The evaluation was repeated for three times and the final result was obtained by averaging the results from the three evaluations.

The performance metrics used in the evaluation are precision, recall, and F_1-score. A higher precision indicates fewer false positives, while a higher recall indicates fewer false negatives. The F_1-score considers both the precision and recall, and is a commonly used parameter to measure the overall accuracy.

We replicated the experiments using the peer-matching approach proposed in Karbab et al. (2016). For ease of comparison, the results obtained using the APK, DEX, assembly, and permission-based sub-fingerprints are listed in Table 4. The underlying combination function is majority voting. It can be seen that among different compositions of the sub-fingerprints, the best result was achieved when combining the assembly and permission-based sub-fingerprints, where the F_1-score is 93.6%.

The results for our proposed XML string-based sub-fingerprint are also listed in Table 4. It is interesting to find that the XML string-based sub-fingerprint achieved much better result (with F_1-score being 94.5%) than that of the permission-based sub-fingerprint (with F_1-score being 80.6%) and even better than that of the best combination of Karbab et al. (2016) described above. This highlights the merit of considering additional information in the XML file besides the requested permissions. When combined with assembly-based sub-fingerprint or assembly-based and DEX-based sub-fingerprints, the F_1-score both can be further improved to 95.9%. Nonetheless, the former takes advantage of two instead of three sub-fingerprints, and hence is more resource efficient.

[2] https://www.virustotal.com.

Table 4 – Performance results of the peer-matching approach proposed in Karbab et al. (2016) using different sub-fingerprints and the combination function of majority voting

Fingerprint subset	Performance metrics		
	Precision (%)	Recall (%)	F_1-score (%)
Apk	16.3	47.2	13.4
Assembly	91.3	92.5	91.8
Dex	57.4	83.3	59.1
Permission	82.4	84.2	80.6
Assembly, Permission, Dex, Apk	88.7	92.1	88.5
Assembly, Permission, Dex	91.2	92.7	91.5
Assembly, Permission	93.5	93.7	93.6
Xml	95.4	95.1	94.5
Assembly, Xml, Dex, Apk	93.2	95.6	93.2
Assembly, Xml, Dex	95.4	96.6	95.9
Assembly, Xml	95.4	96.6	95.9

Table 5 – Performance results of the proposed parallel PA-approach using different sub-fingerprints and combination functions, where *avg, mv, max* denotes average, majority voting and maximum, respectively

Fingerprint subset	Performance metrics		
	Precision (%)	Recall (%)	F_1-score (%)
Apk	67.5	30.2	29.4
Assembly	96.8	96.5	96.5
Dex	95.0	93.1	93.4
Xml	98.5	98.1	98.1
{Assembly, Xml, Dex, Apk}$_{avg}$	98.7	98.4	98.4
{Assembly, Xml, Dex}$_{avg}$	98.5	98.1	98.1
{Assembly, Xml}$_{avg}$	98.7	98.4	98.4
{Assembly, Xml, Dex, Apk}$_{mv}$	97.4	96.8	96.8
{Assembly, Xml, Dex}$_{mv}$	97.2	96.5	96.6
{Assembly, Xml}$_{mv}$	98.5	98.1	98.1
{Assembly, Xml, Dex, Apk}$_{max}$	96.7	95.5	95.4
{Assembly, Xml, Dex}$_{max}$	98.5	98.1	98.1
{Assembly, Xml}$_{max}$	98.5	98.1	98.1

Then we tested our parallel PA-based method under different combination functions, i.e., majority voting, average of scores, and maximum of scores. The evaluation dataset was also split in a stratified fashion with 70:30 ratio and the evaluation was repeated for three times. Sice the size of the training dataset is relatively small, all the training samples of each sub-fingerprint were fed to the corresponding classifier in one mini-batch. That is, the PA classifier is trained in the manner of a batch learner. As shown in Table 5, the averaged F_1-score of the XML string based sub-fingerprint was further improved to 98.1%. The obvious improvement compared to 94.5% of the same sub-fingerprint using the peer-matching approach shown in Table 4 should be due to the classifier assigning more weights to the discriminating features, which in turn

Table 6 – Performance results for concatenating different sub-fingerprints and using a single PA classifier

Concatenated sub-fingerprints	Performance metrics		
	Precision (%)	Recall (%)	F_1-score (%)
Assembly, Dex	95.4	93.6	93.9
Assembly, Xml	97.9	97.6	97.6
Dex, Xml	95.4	93.6	93.9
Assembly, Xml, Dex	95.4	93.6	93.9
Assembly, Xml, Dex, Apk	93.7	93.0	92.9

improved the classification accuracy. Besides, from Table 5, it can also be observed that the combination of the assembly-based and XML string-based sub-fingerprints consistently achieve the best classification accuracy under each of the three combination functions. When the combination function of average of scores is used, the averaged F_1-score is the best at 98.4%, which is much higher than the best result obtained in Karbab et al. (2016) (F_1-score at 93.6%).

Regarding the runtime performance, the averaged training time for the classifiers assigned to the XML, assembly, DEX, and APK-based sub-fingerprint is 0.03 s, 0.41 s, 3.43 s and 8.02 s, respectively. The averaged testing time of the whole testing dataset for the four classifiers are all below 0.01 s except for the classifier for APK-based sub-fingerprint (at 0.30 s). Although these sub-fingerprints are of the same bit-vector length, they have varying sparsity. It seems that the more sparse the bit-vector is, the more efficient the PA classifier can process.

We also experimented with concatenating several sub-fingerprints into a large bit-vector and using a single PA classifier for the family attribution. The averaged precision, recall, and F_1-score for concatenation of different sub-fingerprints are listed in Table 6. The parallel PA-based approach obviously outperformed the single PA-based one, which confirms that assigning a dedicated classifier for each sub-fingerprint helps magnify the impact of distinguishing features.

4.3. *Evaluation on the Drebin and in-the-wild dataset*

The remaining experiments performed for method evaluation are to be presented in two sub-sections, where the first one is for evaluating the proposed method on malware family attribution using the Drebin malware dataset and the second one is for evaluating it on malware detection using both the Drebin dataset and the in-the-wild dataset. As the assembly and XML-based sub-fingerprint achieved much better F_1-score and efficiency than other sub-fingerprints, in the following experiments, we will only use these two sub-fingerprints. Besides, we will use *accuracy* as an additional evaluation metric so as to facilitate comparison with existing methods.

4.3.1. *Malware family attribution*

The 5560 malware samples contained in the Drebin dataset belong to 179 families. As the utilized aapt tool can not properly extract XML strings from some samples, the final number of workable Drebin samples in our experiments is 5375. We follow the setup of a recent notable family attribution method (Dash et al., 2016), where the performance is evaluated on the largest 23 families and the achieved classification accuracy is 94%. An overview of the largest 23 families is presented in Table 7. All the other families contain malware less than 20 samples.

Table 7 – Top 23 malware families of Drebin dataset used in evaluation

Family name	Size	Family name	Size
FakeInstaller	883	Glodream	69
DroidKungFu	653	ExploitLinuxLotoor	68
Plankton	625	FakeRun	61
Opfake	607	SendPay	59
GinMaster	331	Gappusin	58
BaseBridge	329	Imlog	43
Iconosys	152	Yzhc	37
Kmin	147	SMSreg	34
FakeDoc	130	Jifake	27
Adrd	91	Boxer	24
Geinimi	87	MobileTx	21
DroidDream	75		

Table 8 – Performance results of the proposed method for family attribution on the top 23 families of Drebin malware

Fingerprint subset	Performance metrics			
	Precision (%)	Recall (%)	F_1-score (%)	Accuracy (%)
Assembly	96.4	96.1	96.0	98.3
Xml	96.4	95.1	95.5	98.1
$\{\text{Assembly, Xml}\}_{avg}$	97.2	95.9	96.4	98.8
$\{\text{Assembly, Xml}\}_{mv}$	97.2	95.7	96.2	98.6
$\{\text{Assembly, Xml}\}_{max}$	97.2	95.7	96.2	98.6

The evaluation dataset was also split in a stratified fashion with 70:30 ratio. The three-time averaged evaluation results on the 23 malware families, which consists of 4611 malware samples, are listed in Table 8. It can be seen that the combination of the assembly and XML string-based sub-fingerprints with average of scores achieved the best accuracy, at 98.8%.

To further investigate the family attribution performance of our method for each malware family, we used the confusion matrix, which provides a quick graphical overview. Fig. 3 shows the normalized confusion matrix of one round using the assembly and XML string-based sub-fingerprints and the combination function of average of scores. Each matrix element is normalized by dividing it with the sum of the corresponding row. It can be observed that for many malware families all the testing samples were correctly classified, such as the largest 9 malware families. Nonetheless, a large portion of malware samples in the *Boxer* family were misclassified into the *FakeInstaller* family. There are two reasons for this. One is that there is a limited number of malware samples (only 24 samples) in the *Boxer* family, thereby refraining the classifiers from distilling enough distinguishing features for this family. The second one is that samples in the *Boxer* family have

Table 9 – Performance results of the proposed method for malware detection on 5,375 Drebin malware and 5,000 benign applications

Fingerprint subset	Performance metrics			
	Precision (%)	Recall (%)	F_1-score (%)	Accuracy (%)
Assembly	98.4	98.4	98.4	98.4
Xml	98.8	98.8	98.8	98.8
$\{\text{Assembly, Xml}\}_{avg}$	99.2	99.1	99.2	99.2
$\{\text{Assembly, Xml}\}_{max}$	98.5	98.4	98.4	98.4

very similar features as those in the *FakeInstaller* family. For example, after looking into these malware, we found one of their main activities is to send SMS messages to Premium-rate numbers, which is a typical feature of *FakeInstaller* malware.

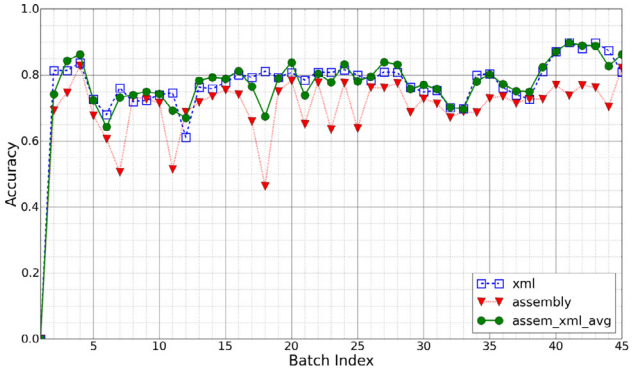
Regarding the runtime performance, the averaged time of learning from all the training samples for the classifiers assigned to the assembly and XML string-based sub-fingerprints is 7.86 s and 0.24 s, respectively. This shows that the utilized PA classifier, which is of linear model, can efficiently process a training dataset of up to 5000 samples. An accompanying benefit is more freedom of choosing the size of the mini-batch when incremental learning is used for a large dataset.

4.3.2. *Malware detection*

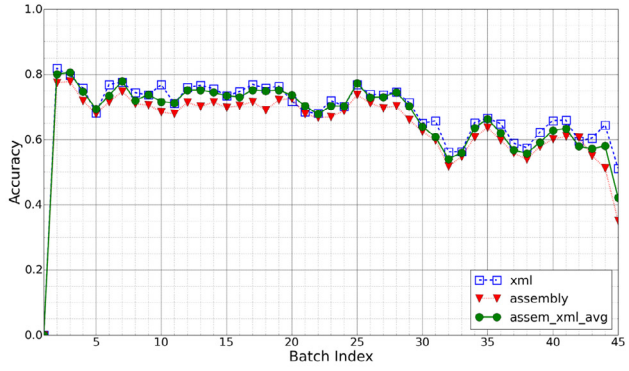
We further performed two sets of experiments to evaluate the effectiveness of our method on malware detection, which is compared with two recent incremental learning-based malware detection methods (Narayanan et al., 2017; 2016) by using the same dataset. The method proposed in Narayanan et al. (2016) extracts features from inter-procedural control-flow graphs and continuously retrain a PA-based model upon receiving each new sample, while the method proposed in Narayanan et al. (2017) leverages a specifically designed kernel to capture both structural and contextual information from API dependency graphs as features and continuously retrain an online CW classifier (Dredze et al., 2008) upon receiving each new sample.

The first set of experiments was on the 5375 malware samples from Drebin dataset and 5000 benign applications created at the same time as Drebin malware (i.e., from 2010 to 2012). These benign applications were also verified with VirusTotal and not reported by any scanner as malicious. Same as in Narayanan et al. (2017), 70% of the samples in the evaluation dataset were used to train and 30% samples were used to test, with the procedure repeated 5 times and the results averaged. The experimental results are listed in Table 9. As the combination function of average of scores and majority voting are essentially the same, only the results for the former are presented. It can be seen that the combination of assembly and XML string-based sub-fingerprints with average of scores achieved the best result, with F_1-score and accuracy both being 99.2%. This result is close to the best result of Narayanan et al. (2017) which achieved F_1-score of 99.23% using the same batch-learning setup and on the same dataset.

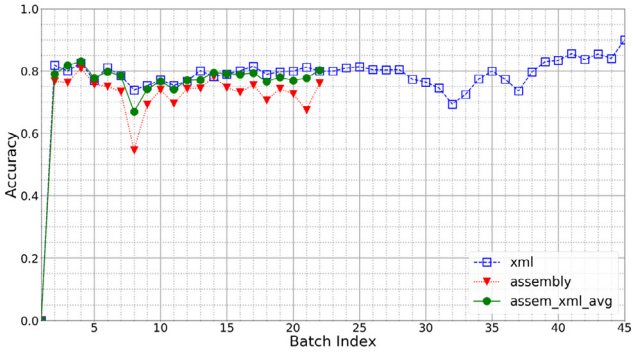
The second set of experiments was performed in the manner of incremental learning, which is on the in-the-wild



(a) The proposed method with PA incrementally trained



(b) Linear SVM trained by the 1st mini-batch



(c) Linear SVM re-trained by all previous mini-batches

Fig. 4 – Accuracy of the proposed method, the linear SVM trained once, and the linear SVM re-trained on the in-the-wild dataset.

dataset. We divided the dataset into mini-batches, each of which contains samples of 5 days. As a result, there is a total of 45 mini-batches. The first mini-batch is only used to train the initial model, while starting from the 2nd batch, before each batch is fed to the classifier for training, it will be used to test the prediction accuracy of the existing model. The experimental results for the assembly and XML string-based sub-fingerprints as well as their combination with average of scores are shown in Fig. 4(a). The results of the combination based on maximum of scores are always inferior to those based on average of scores, and hence not plotted in the figure for clarity. It can be observed that the detection accuracy of using the XML string-based sub-fingerprint was noticeably better than that of using the assembly-based

sub-fingerprint most of the time, and was very close to that of using the combined sub-fingerprints. Considering the combined sub-fingerprints, the detection accuracy for the last 7 mini-batches stays above 82.4% and for the final batch is 86.2%.

To examine the capability of the proposed method in adapting to different features of new applications, we replaced the PA classifiers used in our method with linear SVM classifiers (being the canonical batch learner), trained these classifiers with only the 1st mini-batch, and then tested the detection accuracy with the remaining 44 mini-batches. The experimental results for the assembly and XML-based sub-fingerprints as well as their combination with average of scores are shown in Fig. 4(b). It can be observed that the detection accuracy of the SVM-based model drops as the time of creation of the testing samples gets more distant. This shows that new features emerge in the later testing samples. In contrast, the PA classifiers used in our method generally exhibit an ascending trend in terms of detection accuracy, meaning that our incrementally trained PA-based model can properly adapt to those new features.

On the other hand, we also experimented on re-training the linear SVM classifiers with all previous mini-batches and testing the detection accuracy with the following mini-batch. As the memory consumption increased fast for the growing training data set, our experimental setup (with 16 GB memory) can only afford the re-training of the SVM classifier for the assembly based sub-fingerprint up to the first 22 mini-batches of samples. Nonetheless, as the XML string based sub-fingerprint is sparser than the assembly based and consumes less memory, we managed to finish its re-training for all the 45 mini-batches. The experimental results are shown in Fig. 4(c). As the accuracy of the combined sub-fingerprints is close to that of the XML string based sub-fingerprint for the first 22 mini-batches, we expect them to be also close for the remaining 23 mini-batches. It can be seen that the achieved accuracy is very similar to our incrementally trained PA, indicating that our method does not really sacrifice the accuracy to achieve better scalability.

In contrast, the incremental learning-based method in Narayanan et al. (2016), which also uses the PA-classifier and was tested on the same dataset, achieved a final detection accuracy of 84.29%. It is worth to highlight that the method in Narayanan et al. (2016) achieved better variation than our method, where its detection accuracy varies in the range of 80–85%. This may be due to their trained model being updated for each tested sample, which makes the model adapt faster. Nonetheless, we believe in a real-life scenario (e.g., thousands of new applications are submitted to Google Play every day), it would be infeasible to dissect each new sample and label it immediately. Instead, accumulating the dissected and labeled new samples into mini-batches and then feeding these mini-batches to the classifiers should be a more practical setup.

4.4. *Discussion*

Besides achieving better accuracy compared to Karbab et al. (2016), our method avoids the possibly resource prohibitive peer matching process between each AUT and the whole database, as well as the need to determine a static threshold for whether the AUT should be classified as a malware. In fact, after being trained, the classifiers in our method can provide almost immediate prediction for each AUT.

It should be noted that the *n*-gram size and bit-vector length used to construct the application sub-fingerprints can be adjusted based on actual trade-off between accuracy and processing efficiency. The PA classifier used in our method can be changed to any classifier that supports incremental learning. For example, the CW classifier proposed in Dredze et al. (2008) may be used. Compared to the PA classifier, the CW classifier maintains an additional probabilistic measure of confidence in each parameter, which allows it to update less confident parameters more aggressively than more confident ones. As reported in Dredze et al. (2008), the CW classifier achieved superior classification accuracy and faster learning over the PA classifier. In fact, by changing the PA-classifier used in Narayanan et al. (2016) with a CW classifier, significantly better malware detection accuracies were achieved (Narayanan et al., 2017). Moreover, we use the simple combination functions in our method to fuse the confidence scores from the parallel classifiers due to their efficiency and achieved good results. A more sophisticated information fusion approach, such as one based on the Dempster–Shafer theory (DST) (Shafer, 1976), can also be used. The DST is strong in combining evidence obtained from multiple sources and managing the conflicts among them. It has been successfully applied to combine results of multiple classifiers (Al-Ani and Deriche, 2002; Liu et al., 2017; Zhang and Srihari, 2002) and helped improve the detection accuracy of Android malware in several previous works (Du et al., 2015; Wang et al., 2017).

On the other hand, the parallelized nature of our approach not only allows the classifiers to be trained in parallel but also enables our method to accommodate new features as additional sub-fingerprints. For example, dynamic runtime behaviors of the application can be collected and added into our method as an additional sub-fingerprint. This will help enhance the capability of our method in detecting the malware that employs encryption, dynamic code loading or various sophisticated obfuscation techniques such as reflection and bytecode encryption (Rastogi et al., 2013).

5. Related works

5.1. *Malware detection*

A large number of malware detection methods are based on signature extraction and matching. A notable representative is DroidAnalytics (Zheng et al., 2013), which proposed a three-level signature generation scheme and used Jaccard similarity to measure whether an AUT is similar to some malware. The three-level signature is generated by building firstly the signature of a method based on API call sequences, then the signature of a class using the signatures of the underlying methods, and finally the application signature with all signatures of its classes. Although proven effective, such methods have the drawback of assigning equal weights to all signature bits (i.e., features) and practically expensive signature matching process.

As machine learning algorithms can automatically distill discriminating features and avoid the expensive signature matching process, numerous machine learning-based methods were proposed. Depending on the utilized features, these methods can be mainly divided into static analysis based, dynamic analysis based, and a hybrid type. A good survey of these methods can be found in Egele et al. (2012); Li et al. (2016); Sufatrio et al. (2015). For instance, Drebin (Arp et al., 2014) extracted features such as permissions, hardware components, filtered intents, API calls, and network addresses based on static analysis and used a linear SVM to distinguish between malware and benign applications. Crowdroid (Burguera et al., 2011) extracted dynamic features by monitoring system calls and used k-means algorithm to separate the malware from benign applications. Static analysis-based methods face challenges from code obfuscation (Rastogi et al., 2013) and dynamic code loading (Poeplau et al., 2014), while dynamic analysis-based ones usually have limited code coverage and may be bypassed by various evasion techniques (Diao et al., 2016; Vidas and Christin, 2014). As a result, methods such as Marvin (Lindorfer et al., 2015) proposed to combine the features from both static analysis and dynamic analysis so as to refine the malware detection accuracy. Apart from the inherent features extracted from applications, there have also been methods (Martn et al., 2017; Teufl et al., 2016) exploiting features based on application metadata available in the application market (e.g., Google Play) such as application description, information about the developer, and application ratings. In Teufl et al. (2016), the authors suggested that the metadata-based method should be an essential part of a complete malware detection chain which includes the static and dynamic analysis-based methods described above.

Instead of simply concatenating the different sets of features described above and training a single classifier, some recent methods (Du et al., 2015; Wang et al., 2018; 2017) proposed to train a dedicated classifier for each set of features. For instance, DroidEnsemble (Wang et al., 2018) extracted string features (e.g., permissions, filtered intents, and restricted API calls) and structural features (i.e., function call graph) and achieved improved malware detection accuracy with ensemble of results from two classifiers. All these methods use batch-learning based classifiers, such as SVM, Random Forest (RF), and K-nearest neighbor (KNN). In contrast, our method utilizes online classifiers, where the incremental learning feature makes our method scales better for the massive samples in practice and avoids the need of re-training the model with all previous samples. Moreover, for each set of features, it is possible to utilize heterogeneous classifiers with diverse characteristics to enhance malware detection accuracy (Yerima et al., 2014). Such a design can be easily applied to our model to further improve the detection accuracy.

The machine learning-based methods described above rely on a hand-engineered feature set. Inspired by the success of using n-gram analysis to detect computer malware (Abou-Assaleh et al., 2004; Masud et al., 2008), several works (Canfora et al., 2015; Kang et al., 2016; McLaughlin et al., 2017) proposed to leverage the merits of combining n-gram analysis and machine learning algorithms for android malware detection. To counter the problem of excessive number of unique n-grams, which will cause a huge overhead during the model training process, these methods usually use feature selection techniques such as information gain to distill and keep only the high ranked n-grams as features. Nonetheless, as Android malware keeps on evolving, the discarded lower ranked n-grams may represent discriminating features of new malware. In this regard, the feature hashing technique used in our method can better accommodate the changes of application characteristics.

There has been a rising concern about the impact of malware's rapid evolution on the accuracy of the learned model. For instance, Transcend (Jordaney et al., 2017) statistically compared samples encountered after the model is deployed with those used to train the model and measured the prediction quality so as to raise a red flag before the model starts making consistently poor decisions due to out-of-date training. An orthogonal direction is to build a sustainable model that can automatically adapt to new characteristics of future malware. Examples are the PA-based method in Narayanan et al. (2016) which uses features from inter-procedural control-flow graphs and the online CW classifier (Dredze et al., 2008) based method in Narayanan et al. (2017) which is based on features from structural and contextual information of API dependency graphs. Both method incrementally update the trained model for each new sample. To the best of our knowledge, our method is the first work which proposes to combine n-gram analysis and incremental learning so that new characteristics of future malware can be automatically distilled and utilized for malware detection.

5.2. *Malware family attribution*

Although most of the malware detection methods can be potentially used for malware family attribution, their performance in terms of accuracy and efficiency may not be satisfactory due to the higher complexity of the multi-class problem. There are several methods in literature that are tailored for this task. DroidLegacy (Deshotels et al., 2014) proposed to extract API calls used by malicious modules to create malware family signatures and then match the API called by each module of an AUT against the signatures of known malware families. If the Jaccard similarity of the API called by any of its modules against the API calls in one malware family signature exceeds some threshold, the AUT is deemed a member of the family. Dendroid (Suarez-Tangil et al., 2014b) proposed to extract control flow graph (CFG)-based code structure information to characterize each malware family and then use a 1-NN classifier to compute the predicted family for each AUT. DroidSIFT (Zhang et al., 2014) proposed to classify the AUT based on the extracted weighted contextual API dependency graphs, which considers both dependencies among the API calls and the context of a specific API call being triggered (e.g., through user interfaces or background callbacks). Despite of being also a static analysis-based method, as the constructed features are based on behavior graphs, DroidSIFT has good immunity against byte-code level transformation attacks.

To better counter with continuously improved obfuscation techniques (Balachandran et al., 2016; Suarez-Tangil et al., 2014a), DroidScribe (Dash et al., 2016) used runtime behaviors observed during dynamic analysis to classify Android malware into families, where features such as pure system

calls, decoded binder communication and abstracted behavioral patterns were fed to an SVM-based multi-class classifier. A statistical mechanism was used to evaluate the classification quality and Conformal Prediction (Vovk et al., 2005) was selectively applied to refine the classification whenever the results for SVM is statistically unreliable. As our method is easy to be extended by adding new features as extra sub-fingerprints, in practice, specific static features such as the behavior graph-based ones and dynamic features can be used as additional sub-fingerprints, which helps further improve the accuracy of malware detection and family attribution.

6. Conclusion

In this paper, we presented a parallel Passive Aggressive classifier-based approach for Android malware detection and family attribution. With the application features captured and stored in the multi-level *n*-gram-based fingerprint and the automatic feature distilling capability of the classifiers, our method achieves a high accuracy in detecting malware and attributing it to the corresponding family. The parallelized design of the classifiers not only helps magnify the impact of the malware-distinguishing features, but also allows the classifiers to be trained in parallel and hence enhances the efficiency. Besides making our method scales well and capable of handling the huge number of applications in real life, the incremental learning enabled by the utilized online classifiers also facilitates the built model to continuously adapt to new features in applications and retain classification accuracy. On the other hand, our method can be easily expanded, where other information such as dynamic runtime analysis-based features can be added as extra sub-fingerprints. This will be part of our future work.

Acknowledgment

This material is based on research work supported by the Singapore National Research Foundation under NCR Award No. NRF2014NCR-NCR001-034.

REFERENCES

Aafer Y, Du W, Yin H. Droidapiminer: mining api-level features for robust malware detection in android. In: Proceedings of the international conference on security and privacy in communication systems; 2013. p. 86–103.

Abou-Assaleh T, Cercone N, Keselj V, Sweidan R. N-gram-based detection of new malicious code, 2. IEEE; 2004. p. 41–2.

Afonso VM, de Amorim MF, Grégio ARA, Junquera GB, de Geus PL. Identifying android malware using dynamically obtained features. J Comput Virol Hacking Tech 2015;11(1):9–17.

Al-Ani A, Deriche M. A new technique for combining multiple classifiers using the dempster-shafer theory of evidence. J Artif Intell Res 2002;17:333–61.

Allix K, Bissyandé TF, Jérome Q, Klein J, Le Traon Y, et al. Empirical assessment of machine learning-based malware detectors for android. Empir Softw Eng 2016;21(1):183–211.

Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K, Siemens C. Drebin: effective and explainable detection of android malware in your pocket. Proceedings of network and distributed systems security(NDSS), 2014.

Balachandran V, Tan DJ, Thing VL, et al. Control flow obfuscation for android applications. Comput Secur 2016;61:72–93.

Burguera I, Zurutuza U, Nadjm-Tehrani S. Crowdroid: behavior-based malware detection system for android. In: Proceedings of the ACM workshop on security and privacy in smartphones and mobile devices; 2011. p. 15–26.

Canfora G, De Lorenzo A, Medvet E, Mercaldo F, Visaggio CA. Effectiveness of opcode ngrams for detection of multi family android malware. In: Proceedings of the IEEE international conference on availability, reliability and security (ARES); 2015. p. 333–40.

Cavnar WB, Trenkle JM. N-gram-based text categorization. In: Proceedings of 3rd annual symposium on document analysis and information retrieval (SDAIR); 1994. p. 161–75.

Cover TM. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. IEEE Trans Electron Comput 1965;14(3):326–34.

Crammer K, Dekel O, Keshet J, Shalev-Shwartz S, Singer Y. Online passive-aggressive algorithms. J Mach Learn Res 2006;7(Mar):551–85.

Dash SK, Suarez-Tangil G, Khan S, Tam K, Ahmadi M, Kinder J, Cavallaro L. Droidscribe: classifying android malware based on runtime behavior. In: Proceedings of the IEEE security and privacy workshops (SPW); 2016. p. 252–61.

Deshotels L, Notani V, Lakhotia A. Droidlegacy: automated familial classification of android malware. In: Proceedings of ACM SIGPLAN on program protection and reverse engineering workshop.

Diao W, Liu X, Li Z, Zhang K. Evading android runtime analysis through detecting programmed interactions. In: Proceedings of the 9th ACM conference on security & privacy in wireless and mobile networks. ACM; 2016. p. 159–64.

Dredze M, Crammer K, Pereira F. Confidence-weighted linear classification. In: Proceedings of the 25th international conference on machine learning; 2008. p. 264–71.

Du Y, Wang X, Wang J. A static android malicious code detection method based on multi-source fusion. Secur Commun Netw 2015;8(17):3238–46.

Egele M, Scholte T, Kirda E, Kruegel C. A survey on automated dynamic malware-analysis techniques and tools. ACM Comput Surv (CSUR) 2012;44(2):6.

Gartner. Gartner says worldwide sales of smartphones grew 7 percent in the fourth quarter of 2016. http://www.gartner.com/newsroom/id/3609817.

Hanna S, Huang L, Wu E, Li S, Chen C, Song D. Juxtapp: a scalable system for detecting code reuse among android applications. In: Proceedings of the international conference on detection of intrusions and malware, and vulnerability assessment; 2012. p. 62–81.

Jang J, Brumley D, Venkataraman S. Bitshred: feature hashing malware for scalable triage and semantic analysis. In: Proceedings of the 18th ACM conference on computer and communications security (CCS); 2011. p. 309–20.

Jordaney R, Sharad K, Dash SK, Wang Z, Papini D, Nouretdinov I, Cavallaro L. Transcend: detecting concept drift in malware classification models. Proceedings of the 26th USENIX security symposium, 2017.

Kang B, Yerima SY, McLaughlin K, Sezer S. N-opcode analysis for android malware classification and categorization. In: Proceedings of the IEEE international conference On cyber security and protection of digital services; 2016. p. 1–7.

Karbab EB, Debbabi M, Mouheb D. Fingerprinting android packaging: generating dnas for malware detection. Digit Investig 2016;18:33–45.

Larson RR. Introduction to information retrieval. J Am Soc Inf Sci Technol 2010;61(4):852–3.

Li L, Bissyande TFDA, Papadakis M, Rasthofer S, Bartel A,

Octeau D, Klein J, Le Traon Y. In: Technical Report. Static analysis of android apps: a systematic literature review; 2016.
Lindorfer M, Neugschwandtner M, Platzer C. Marvin: efficient and comprehensive mobile app classification through static and dynamic analysis, 2; 2015. p. 422–33.
Liu Z, Pan Q, Dezert J, Martin A. Combination of classifiers with optimal weight based on evidential reasoning. IEEE Trans Fuzzy Syst 2017;26(3):1217–30.
Lueg C. 8,400 new android malware samples every day. https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day.
Martn I, Hernndez J, Muoz A, Guzmn A. Android malware characterization using metadata and machine learning techniques. ArXiv e-prints 2017.
Masud MM, Khan L, Thuraisingham B. A scalable multi-level feature extraction technique to detect malicious executables. Inf Syst Front 2008;10(1):33–45.
McLaughlin N, Martinez del Rincon J, Kang B, Yerima S, Miller P, Sezer S, Safaei Y, Trickel E, Zhao Z, Doupe A, et al. Deep android malware detection. In: Proceedings of the seventh ACM on conference on data and application security and privacy (CODASPY); 2017. p. 301–8.
Narayanan A, Chandramohan M, Chen L, Liu Y. Context-aware, adaptive, and scalable android malware detection through online learning. IEEE Trans Emerg Top Comput Intell 2017;1(3):157–75.
Narayanan A, Yang L, Chen L, Jinliang L. Adaptive and scalable android malware detection through online learning. In: Proceedings of the IEEE international joint conference on neural networks (IJCNN); 2016. p. 2484–91.
Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, et al. Scikit-learn: machine learning in python. J Mach Learn Res 2011;12(Oct):2825–30.
Poeplau S, Fratantonio Y, Bianchi A, Kruegel C, Vigna G. Execute this! analyzing unsafe and malicious dynamic code loading in android applications., 14; 2014. p. 23–6.
PulseSecure. Pulse secure releases first mobile threat report; finds 391 percent increase of unique malicious applications developed in 2014. https://www.pulsesecure.net/news-events/press-releases/pulse-secure-releases-first-mobile-threat-report-finds-391-percent-increase-of---11g037804-001.
Rastogi V, Chen Y, Jiang X. Droidchameleon: evaluating android anti-malware against transformation attacks. In: Proceedings of the ACM SIGSAC symposium on information, computer and communications security; 2013. p. 329–34.
Sanz B, Santos I, Laorden C, Ugarte-Pedrero X, Bringas PG, Álvarez G. Puma: permission usage to detect malware in android. In: Proceedings of the international joint conference CISIS-ICEUTE-SOCO special sessions; 2013. p. 289–98.
Shafer G, 42. Princeton University Press; 1976.
Singh A, Walenstein A, Lakhotia A. Tracking concept drift in malware families. In: Proceedings of the 5th ACM workshop on security and artificial intelligence; 2012. p. 81–92.
Suarez-Tangil G, Tapiador JE, Peris-Lopez P. Stegomalware: playing hide and seek with malicious components in smartphone apps. In: Proceedings of the international conference on information security and cryptology; 2014a. p. 496–515.
Suarez-Tangil G, Tapiador JE, Peris-Lopez P, Blasco J. Dendroid: a text mining approach to analyzing and classifying code structures in android malware families. Expert Syst Appl 2014b;41(4):1104–17.
Sufatrio, Tan DJ, Chua TW, Thing VL. Securing android: a survey, taxonomy, and challenges. ACM Comput Surv (CSUR) 2015;47(4):58.
Sun M, Li X, Lui JC, Ma RT, Liang Z. Monet: a user-oriented behavior-based malware variants detection system for android. IEEE Trans Inf Forensics Secur 2017;12(5):1103–12.
Teufl P, Ferk M, Fitzek A, Hein D, Kraxberger S, Orthacker C. Malware detection by applying knowledge discovery processes to application metadata on the android market (google play). Secur Commun Netw 2016;9(5):389–419.
The Hacker News. Beware! new android malware infected 2 million google play store users. http://thehackernews.com/2017/04/android-malware-playstore.html.
Vidas T, Christin N. Evading android runtime analysis via sandbox detection. In: Proceedings of the 9th ACM symposium on information, computer and communications security. ACM; 2014. p. 447–58.
Vovk V, Gammerman A, Shafer G. Algorithmic learning in a random world. Springer Science & Business Media; 2005.
Wang W, Gao Z, Zhao M, Li Y, Liu J, Zhang X. Droidensemble: detecting android malicious applications with ensemble of string and structural static features. IEEE Access 2018;6:31798–807.
Wang X, Zhang D, Su X, Li W. Mlifdect: android malware detection based on parallel machine learning and information fusion. Secur Commun Netw 2017;2017.
Weinberger K, Dasgupta A, Langford J, Smola A, Attenberg J. Feature hashing for large scale multitask learning. In: Proceedings of the 26th annual international conference on machine learning; 2009. p. 1113–20.
Yerima SY, Sezer S, Muttik I. Android malware detection using parallel machine learning classifiers. In: Proceedings of IEEE international conference on next generation mobile apps, services and technologies (NGMAST); 2014. p. 37–42.
Yerima SY, Sezer S, Muttik I. High accuracy android malware detection using ensemble learning. IET Inf Secur 2015;9(6):313–20.
Yuan GX, Ho CH, Lin CJ. Recent advances of large-scale linear classification. Proc IEEE 2012;100(9):2584–603.
Zhang B, Srihari SN. Class-wise multi-classifier combination based on dempster-shafer theory, 2; 2002. p. 698–703.
Zhang M, Duan Y, Yin H, Zhao Z. Semantics-aware android malware classification using weighted contextual api dependency graphs. In: Proceedings of ACM SIGSAC conference on computer and communications security; 2014. p. 1105–16.
Zheng M, Sun M, Lui JC. Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware. In: Proceedings of the IEEE international conference on trust, security and privacy in computing and communications (TrustCom); 2013. p. 163–71.
Zhou Y, Jiang X. Dissecting android malware: characterization and evolution. In: Proceedings of the IEEE symposium on security and privacy (SP); 2012. p. 95–109.

Li Zhang received the B.Eng. (Hons.) and Ph.D. degrees from Nanyang Technological University (NTU), Singapore, in 2010 and 2015, respectively. He served as a security evaluator for smart cards in UL before joining the Cyber Security and Intelligence Department at the Institute for Infocomm Research (I2R), A*STAR as a research scientist. His research interests include vulnerability detection, malware analysis and classification, and hardware security and trust.

Dr. Vrizlynn Thing is the Head of Cyber Security & Intelligence Department at the Institute for Infocomm Research, A*STAR. She is also an Adjunct Associate Professor at the National University of Singapore, and holds the appointment of Honorary Assistant Superintendent of Police (Specialist V) at the Singapore Police Force, Ministry of Home Affairs. During her career, she has taken on various roles to lead and conduct cyber security R&D that benefits our economy and society. She participates actively as the Lead Scientist of collaborative projects with industry partners and

government agencies, and takes on advisory roles at the national and international level.

Yao Cheng received her Ph.D. degree in Computer Science and Technology from University of Chinese Academy of Sciences in 2015. She is currently a scientist at Institute for Infocomm Research, A*STAR, Singapore. Her research interests are in the information security area, focusing on vulnerability analysis, privacy leakage and protection, malicious application detection, and usable security solutions.