This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2020.3005088, IEEE Transactions on Dependable and Secure Computing

SDAC: A Slow-Aging Solution for Android Malware Detection Using Semantic Distance Based API Clustering

Jiayun Xu, Yingjiu Li, Robert H. Deng, and Ke Xu, *Singapore Management University*

Abstract—A novel slow-aging solution named SDAC is proposed to address the model aging problem in Android malware detection, which is due to the lack of adapting to the changes in Android specifications during malware detection. Different from periodic retraining of detection models in existing solutions, SDAC evolves effectively by evaluating new APIs' contributions to malware detection according to existing API's contributions.
In SDAC, the contributions of APIs are evaluated by their contexts in the API call sequences extracted from Android apps. A neural network is applied on the sequences to assign APIs to vectors, among which the differences of API vectors are regarded as the semantic distances. SDAC then clusters all APIs based on their semantic distances to create a feature set in the training phase, and extends the feature set to include all new APIs in the detecting phase. Without being trained by any new set of real-labelled apps, SDAC can adapt to the changes in Android specifications by simply identifying new APIs appearing in the detection phase. In extensive experiments with datasets dated from 2011 to 2016, SDAC achieves a significantly higher accuracy and a significantly slower aging speed compared with MaMaDroid, a state-of-the-art Android malware detection solution which maintains resilience to API changes.

Index Terms—Android Malware Detection, Mobile Security.

1 Introduction

Most Android malware detection models age quickly. According to Zhu and Dumitras [1], an Android malware detection model generated in 2012 failed to detect any malware in the Gappusin family while a 2014 model could detect most of them. In another research of Wang from Baidu [2], the recall rate of an Android malware detection model developed at Baidu decreased by 7.6% in the first six months. Recent research has identified a main reason of model aging to be API changes over time in Android specifications [1], [3]. Apparently, malware samples making use of newly added APIs in performing malicious behaviors may evade from the detection of aged models.

The common solution to address the aging of Android malware detection models is either to update signature databases for signature-based malware detection, or to renew malware detection models using new Android apps with true labels (i.e., malware and benignware) for learning-based models. However, this process is usually time-consuming and costly, which may involve many domain experts' efforts on the sample labelling and data sharing across multiple organizations. Furthermore, the true labels of newly collected apps may not be connivently or promptly available and even be mistaken in real life.

For instance, we downloaded the reports for a set of 42808 apps from VirusTotal[1] in July 2017 and July 2018 respectively. In these apps, about 11% (4717/42808) of them which were labelled as "benign" by all the antivirus engines in July 2017 turned out to be labelled as "dangerous" by at least one antivirus engine in July 2018, indicating that the labels may be erroneous for a long time. It is thus imperative to develop a slow-aging solution that remains accurate in malware detection for longer time and can be renewed without relying on the true labels of new apps.

While Android API changes have been identified as a major problem leading to model aging in Android malware detection [1], [3], the adaptation to API changes has not been rigorously addressed in the design of slow-aging solutions. A recent solution named PikaDroid [4] addressed the aging problem by utilizing the contextual information of sensitive APIs in malware detection. However, it does not adapt to the changes of Android specifications since the feature set in PikaDroid remains unchanged in its design.

Another approach, MaMaDroid [5] proposed a detection method which is resilient to the changes in Android specifications. In particular, MaMaDroid first abstracts application programming interfaces (APIs) to their corresponding packages (or package families) in the API execution paths derived from each Android app. It then summarizes all abstracted paths to a Markov model, and converts the Markov model to a feature vector for each app in model training and testing.

By abstracting APIs to packages, MaMaDroid is resilient to the adding of new APIs to *existing packages* and performs significantly better than other solutions such as DroidAPIMiner for Android malware detection. However, MaMaDroid does not address the contribution of any *new packages* to malware detection since the transitions caused by any new packages in Markov models convert to no features in MaMaDroid.

On average, about 340 new APIs in 4 new packages were added to each API-level compared to its previous one according to the Android developer documentation [6]. These new packages and new APIs are important factors

1. VirusTotal is a website which aggregates multiple antivirus scan engines.

leading to model aging in Android malware detection. Without model updating, the performance of existing malware detection models, including MaMaDroid, may downgrade significantly over time as more and more new packages and APIs are added in Android specifications and used in Android app development.

In this paper, we develop a learning-based and slow-aging solution for Android malware detection. Our slow-aging solution is named SDAC, which stands for semantic distance based API clustering. SDAC identifies an API's contribution to malicious behaviors based on the *APIś contexts*, which refers to the APIs within a fixed-size window from the API in the API call sequences performed by apps. In particular, given a training set of apps with true labels, SDAC extracts API call sequences from the apps. Based on the extracted API sequences, SDAC applies a two-layer neural network to embed APIs into *API vectors* and arrange these vectors into a vector space. In the vector space, the APIs sharing common *API contexts* are located close to each other. A *feature set* is formed according to API vector clusters, where each feature is defined as the set of all APIs whose corresponding *API vectors* are in a same cluster.

For each app in the training set, a binary feature vector is generated by a one-to-one mapping from the feature set. An element in an app's feature vector is assigned to zero if none of the APIs in the mapped feature is used by this app, and it is one otherwise. Any classification models can be built based on the feature vectors that are derived from training apps and their corresponding labels.

To make SDAC slow-aging, it is important to identify the new APIs' contributions to malware detection without knowing the true labels of the new apps that use these APIs. The key technique of performing such identification in SDAC is *feature extension*. In this step, each new API that appears in the testing set of apps is added to the closest feature in the feature set according to the distance measured in the API vector space. A new API's contribution to malware detection is modelled to be equivalent to the contribution of the other APIs in the same feature. A trained classification model does not need to be re-trained in the test phase since the contributions of all "old" APIs have already been evaluated in the training phase.

When performing malware detection over time on a series of testing sets in which apps are developed in successive time periods, SDAC can be executed in two major modes, SDAC-FEO and SDAC-FMU. SDAC-FEO is short for "SDAC-Feature Extension Only", in which feature extension is performed each time with a new testing set, while the trained classification model keeps unchanged all the time. In SDAC-FMU, which is short for "SDAC-Feature and Model Updating", both the classification model and the feature set are updated with new testing set. Note that although the classifiers are changed in some cases, both SDAC-FEO and SDAC-FMU do not need any labelled new samples in the successive time periods while can still keep a high accuracy over time, thus the whole solution SDAC is regarded as being slow-aging.

Both SDAC-FEO and SDAC-FMU require the current testing set to be wholly available in feature extension to collect the new APIs' contexts. To relax this constraint, we design SDAC-FEO-OL and SDAC-FMU-OL for online detection of individual apps without waiting for the whole testing set being available. In particular, they use existing classification models with no feature extension for online detection of individual apps, while after the whole testing set is available, they resort to their off-line versions to update themselves.

We evaluated the performances of SDAC in different modes and versions using 70,142 Android app samples dated from 2011 to 2016. For simplicity, we refer to the scenario as our "default setting" in which the 2011 samples are used for both training and 5-fold cross validation, while the 2012-2016 samples are used for testing. The evaluation results in the default setting show that the F-score of SDAC-FEO declines by 4.81% per year on average from 2011 to 2016, while MaMaDroid declines by 7.67% per year. The average F-score of SDAC-FEO on these testing sets is 87.23%, which is higher than that of MaMaDroid in the same case(59.03%), by 28.2%. Compared to SDAC-FEO, SDAC-FMU further reduces the aging speed from 4.81% to 0.10% per year on average, and increases the average F-score from 87.23% to 97.09%. While the online versions are more efficient in classifying each app online, their accuracies are slightly lower, and their aging speeds are slightly higher than the corresponding non-online versions.

2 Basic SDAC

An Android app can be considered as a set of operation paths, where each operation path is a sequence of operations that can be executed on Android platforms under certain conditions. In the design of SDAC, we focus on API call operations by which an Android app accesses Android system services and resources. An Android app may use any API that is provided in Android specifications at the time when it is developed. While Android specifications evolve over time from API-level 1 to API-level 27, a number of new APIs are added continually. Of course, an Android app cannot use any new APIs that do not exist at the time when it is developed.

Assumptions. SDAC is trained with a set of Android apps that are associated with true labels, including malware and benignware, where the apps in the training set are developed in a same time period. After training, SDAC is used to classify all apps in testing sets in other time periods. It is assumed that no true labels are available for the apps in testing sets when SDAC is in use.

A basic SDAC is first developed to classify apps in one testing set, which is developed in a time period after the training set. It is then extended in different modes and versions to classify apps in multiple testing sets, which are developed in successive later time periods.

Structure. The structure of basic SDAC is illustrated in Fig. 1. It consists of two phases, a *training phase* in which SDAC is trained with a training set, and a *detection phase* in which SDAC is used to detect malware in a testing set. In both phases, the basic SDAC proceeds through four steps, where the first two steps, including *API path extraction* and *API vector embedding*, are shared in both phases. The last two steps are *API cluster generation* and *classification model training* in the training phase, and *API cluster extension* and *classification model testing* in the detection phase.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2020.3005088, IEEE Transactions on Dependable and Secure Computing

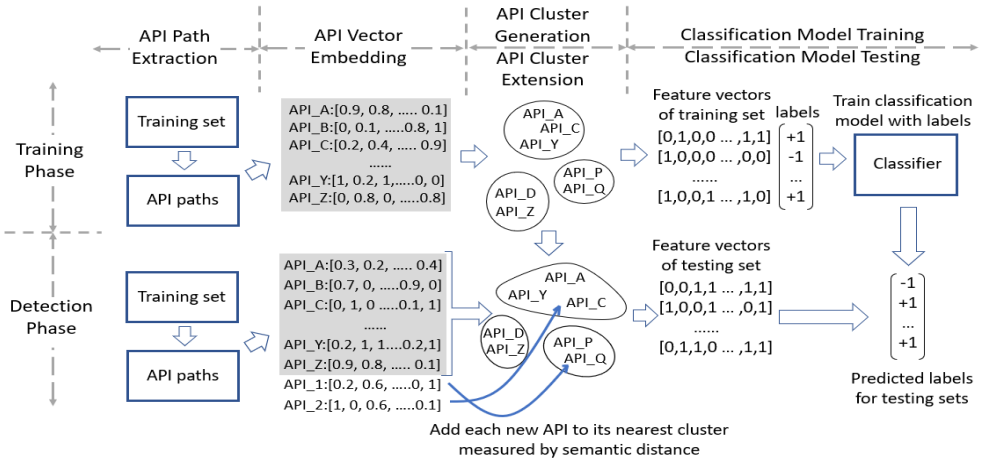


Fig. 1: Structure of Basic SDAC with One Classification Model

2.1 API Path Extraction

The first step of SDAC is to extract a set of API call paths from each app it processes. A suitable static analysis tool such as FlowDroid [7] can be exploited to transform each app from bytecode to proper representation (e.g., Jimple code), and extract a directed call graph among its program methods. From a call graph, SDAC extracts a call graph snippet for each method, and excludes any call graph snippet that is a sub-graph of another call graph snippet.

The call graph snippet for a target method consists of all methods and all directed links between them in the call graph. In a call graph snippet, all the methods are located at most d links away from the target one in the call graph, where the links pointing to any methods that use no APIs are not counted in the snippet. SDAC then derives all API call paths from each call graph snippet. Appendix A shows a real-world example for API call path extraction.

If d is large enough, SDAC would output all possible API call paths for each app, which is an intractable number to program size [8]. To make SDAC practical, the parameter d is chosen to be relatively small, which is equal to the window size S as mentioned in the next subsection.

2.2 API Vector Embedding

The goal of this step is to embed each API into an n-dimensional real-valued vector in $[0, 1]^K$ according to the API's context in a set of apps. An *API's context* in a set of apps is defined to be the set of all its neighboring APIs within a fix-sized window containing the API in all API call paths that are extracted from the set of apps. Different APIs with similar contexts are mapped close to each other in the vector space.

The process of API vector embedding is illustrated in Fig. 2. Based on the API call paths extracted in step "API path extraction", SDAC builds an *API vocabulary* which consists of all unique APIs in these paths. For each target API in the API vocabulary and for each extracted API path, SDAC derives all neighboring APIs which are at most S APIs away from the target API in the API path, where S is called window size. Then, SDAC pairs each target API with each of its neighboring APIs, and uses all such pairs to train a Skip-Gram model, which is a neural network with an input layer, a hidden layer, and an output layer [9]. In the model, a real-valued *input-hidden matrix* W_{V*K} is used to transform an input vector to a hidden input vector, and a real-valued *hidden-output matrix* W'_{K*V} is used to further transform a hidden input vector to an output vector, where V is the API vocabulary size, and K is the API vector size.

In the training process, a target API and all its paired APIs are regarded as the input and the target outputs of the Skip-Gram model, respectively. For a target API, the input of Skip-Gram model is a one-hot encoded vector of size V, where the index corresponding to the target API points to one and all other indexes point to zero. Similarly, a target output for a paired API is a one-hot encoded vector of size V where the index corresponding to the paired API points to one and all other indexes point to zero. For each target API, the Skip-Gram model computes an output vector from the input vector by transforming it with the input-hidden matrix, the hidden-output matrix, and the softmax function (i.e., normalized exponential function), successively. Then, the Skip-Gram model updates the elements in these two matrices using backpropagation, which is a common optimization step for supervised learning of neural networks, so as to minimize the total error between the computed output and the target outputs.

After the Skip-Gram model is trained with all API pairs, SDAC outputs an API vector for each API. In Fig. 2, the API_T's vector is the T-th row in the input-hidden matrix, where T is the index of API_T in the API vocabulary. An API's vector embeds the API's context, which represents all its neighboring APIs that are encoded in the target outputs. If two different APIs are embedded to similar API vectors, thus they have similar contexts because the computed outputs are optimized in model training to approximate their target outputs.

2.3 API Cluster Generation and Extension

The third step of basic SDAC is API cluster generation in its training phase, and API cluster extension in its detection phase. The purpose of this step is to group the APIs in the API vocabulary into a number of clusters based on the *semantic distance* between APIs, where the semantic distance is defined as the Euclidean distance in the API vector space.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2020.3005088, IEEE Transactions on Dependable and Secure Computing

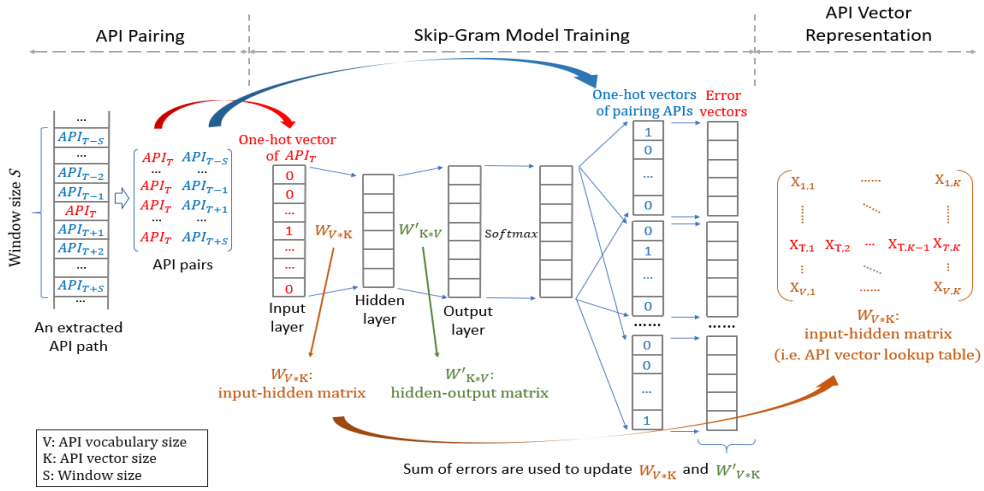


Fig. 2: API Vector Embedding

The output of this step, which is a set of API clusters, will be used as the feature set for generating a feature vector for each app in the next step.

API Cluster Generation. In the training phase, SDAC applies the same-size k-means cluster algorithm from [10] to the API vectors that are calculated from the training set. The algorithm partitions API vectors into k clusters in which each API vector belongs to the cluster with nearest mean, where the difference in size among all clusters is at most one. Using the clusters, an *initial feature set* is defined by a one-to-one mapping from the clusters, where each feature consists of all APIs whose API vectors appear in a same cluster. The APIs in a same feature share similar contexts (due to the closeness of their API vectors), and thus make similar contributions to malware detection in our model.

The same-size k-mean algorithm is chosen in SDAC because it can effectively avoid skew clustering results and thus maximize the differences between the feature vectors of benign apps and malicious ones. Please refer to Appendix B for the detail of the same-size k-means cluster algorithm which we choose.

API Cluster Extension. In the detection phase, SDAC outputs a set of API vectors derived from a testing set in the previous step. Now it extends the initial feature set to include all new APIs that appear in the API vocabulary of the testing set, but not in that of the training set. This step is also named *feature extension* because each cluster is regarded as one feature in the feature set.

In general, SDAC extends feature X to include a new API_Y if API_Y has the least average semantic distance from the APIs of feature X among all features, where semantic distance is measured by API vectors derived from **the testing set**. In a feature, if an API does not exist in any apps of the testing set, SDAC excludes it from the calculation of average semantic distance unless no API in the feature exists in the testing set, in which case corresponding API vectors derived from the training set are used for semantic distance calculation. An *extended feature set* is defined from the initial feature set after all new APIs are included in feature extensions.

A toy example is shown in Fig. 3 to illustrate the feature extension. In Fig. 3, two clusters are formed in the vector

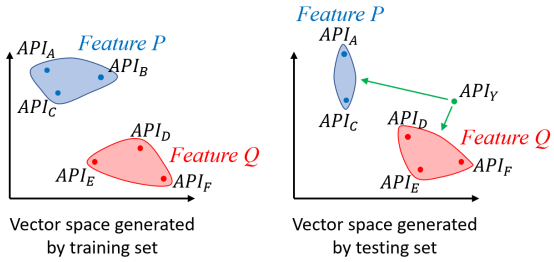


Fig. 3: API Cluster Extension

space generated from a training set, including the one for *Feature P* containing API_A, API_B and API_C, and the one for *Feature Q* containing API_D, API_E and API_F.

Now consider a new API API_Y in the vector space generated from a testing set, an average semantic distance is calculated between API_Y and each cluster of *Feature P* and *Feature Q*. Since *Feature Q* has the least average distance from API_Y, it is expanded to include API_Y for *feature extension*. After that, *Feature Q* will contain four APIs: API_D, API_E, API_F and API_Y, while *Feature P* keeps unchanged. Note that API_B does not appear in testing set and thus has no representative API vector in the vector space, so it is simply excluded in the calculation.

The purpose of generating an extended feature set is to simulate new APIs' contributions to malware detection using existing APIs', where the former cannot be directly measured by a classification model due to the lack of true labels in testing sets.

2.4 Classification Model Training and Testing

Feature Vector Generation. Given a feature set, which is either initial feature set in the training phase, or extended feature set in the detection phase, SDAC generates a binary *feature vector* for each app by a one-to-one mapping from the feature set. An element in an app's feature vector is zero if none of the APIs in its mapped feature is used by the app, and it is one otherwise.

Classification Model Training. In the training phase, SDAC generates a feature vector for each app in the training set, and associates the feature vector with the app's true label.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2020.3005088, IEEE Transactions on Dependable and Secure Computing

Then, SDAC trains a classification model with all feature vectors and associated labels.

Classification Model Testing. In the detection phase, SDAC generates a feature vector for each app in the testing set. Then, SDAC uses the trained classification model to output a predicted label for each app according to its feature vector as model input.

2.5 Model Voting

In SDAC, a feature in the feature set may consist of multiple APIs. It is possible that some of these APIs are used by a benign app, and some of them are used by a malicious app. A single feature can hardly be used to distinguish between benign and malicious apps. It is thus much better to leverage on all features in the feature set for malware detection. To further improves its accuracy, SDAC exploits the distinguishing power of multiple feature sets instead of a single feature set. In particular, SDAC performs its cluster algorithm for $m \geq 1$ times in its third step and thus generates m initial feature sets and m extended feature sets. A classification model is trained on each initial feature set in the training phase and then tested on the corresponding extended feature set in the detection phase. With total m classification models, SDAC outputs a predicted label "malware" for an app if at least $\tau \leq m$ out of m classification models agree on such label, and it outputs "benignware" otherwise.

In our experiments, we discover that F-scores of SDAC increase by around 3% to 10% using multiple voting. The details on tuning τ and m are explained in section 4.

3 Two Modes of SDAC and Online Versions

While the basic SDAC focuses on processing one testing set only, it can be extended in two different modes, SDAC-FEO and SDAC-FMU, to process multiple testing sets $T_1, T_2, \ldots, T_N$ in which apps are developed in different time periods with some new APIs. The difference between SDAC-FEO and SDAC-FMU is that in the detection phase, SDAC-FEO performs feature extension only, while SDAC-FMU performs feature and model updates as well.

3.1 SDAC-FEO

In the training phase, SDAC-FEO takes a training set as input and outputs m initial feature sets and m classification models in the same way as the basic SDAC does. In the detection phase when it is applied to testing set T_1, each initial feature set is extended to an "extended feature set for T_1". Then, each app in T_1 is transformed to m feature vectors according to m "extended feature sets for T_1". Finally, the m classification models are used in model voting to predict a label for each app according to its feature vectors.

When SDAC-FEO is applied to testing set T_N ($N \geq 2$), each "extended feature set for T_{N-1}" is regarded as "initial feature set for T_N", and then it will be extended to an "extended feature set for T_N" in the same way as the "feature extension" step in the basic SDAC. After that, each app in T_N is transformed to m feature vectors according to m "extended feature sets for T_N". The same m classification models are used as before to predict a label for each app according to its feature vectors.

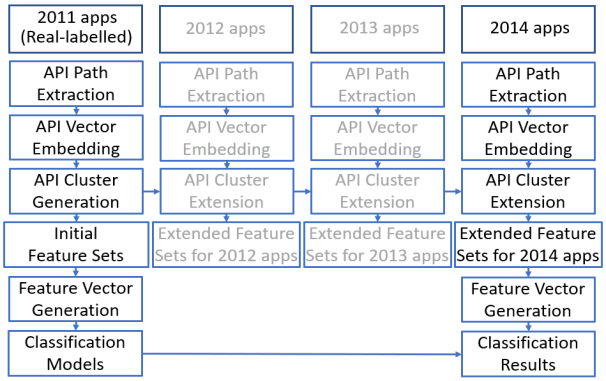


Fig. 4: Structure of SDAC-FEO with One Training Set (2011 apps) and Three Testing Sets (2012 apps, 2013 apps, and 2014 apps)

Fig. 4 shows the structure of SDAC-FEO with one training set (2011 apps), and three testing sets (2012 apps, 2013 apps, and 2014 apps). In this figure, the set of 2011 apps is used to generate initial feature sets and train classification models. The same classification models are used to classify 2014 apps after they are applied to 2012 apps and 2013 apps. The extended feature sets for 2014 apps are generated by extending the initial feature sets for 2011 apps three times in a sequence.

3.2 SDAC-FMU

SDAC-FMU is the same as SDAC-FEO in its training phase, and in the detection phase when it is applied to testing set T_1. It performs additional steps on feature and model updates when it is applied to other testing sets $T_2, \ldots, T_N$.

Now assume that SDAC-FMU has been applied to testing sets $T_1, \ldots, T_{N-1}$, producing a prediction label (i.e., pseudo-label) for each app in these sets. When SDAC-FMU is applied to testing set T_N, it first generates m "initial feature sets for T_N" (in the same way as the basic SDAC generates initial feature sets) from the union of the training set and $T_1, \ldots, T_{N-1}$. We call this process *feature update*. Note that this is different from SDAC-FEO where the initial feature sets never change.

Then, SDAC-FMU trains m classification models for T_N from scratch using (i) the apps in the training set with their true labels, and (ii) the apps in $T_1, \ldots, T_{N-1}$ with their pseudo-labels, in which each app is converted to m feature vectors according to m "initial feature sets for T_N". We call this process *model update*. Note that no true labels are available for the apps in testing sets in our assumption; thus, SDAC-FMU uses pseudo-labels for the apps in $T_1, \ldots, T_{N-1}$ for model update.

After model update, SDAC-FMU extends each "initial feature set for T_N" to an "extended feature set for T_N" (in the same way as the basic SDAC extends an initial feature set). Then, each app in T_N is converted to m feature vectors according to m "extended feature sets for T_N". Finally, the classification models for T_N which have been trained in model update are used to predict a label for each app in T_N according to its feature vectors.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2020.3005088, IEEE Transactions on Dependable and Secure Computing

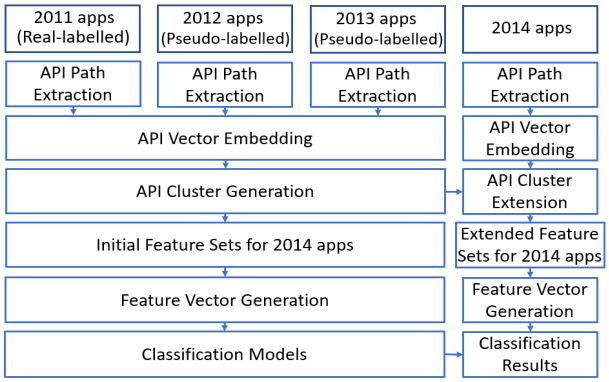


Fig. 5: Structure of SDAC-FMU with One Training Set (2011 apps) and Three Testing Sets (2012 apps, 2013 apps, and 2014 apps)

Fig. 5 shows the structure of SDAC-FMU with one training set (2011 apps), and three testing sets (2012 apps, 2013 apps, and 2014 apps). When SDAC-FMU is applied to 2014 apps after it is trained with 2011 apps and applied to 2012 apps and 2013 apps, it first generates initial feature sets for 2014 apps from the union of 2011 apps, 2012 apps, and 2013 apps. Then, SDAC-FMU generates feature vectors for each app in the union, and trains classification models for 2014 apps using (i) 2011 apps with their true labels, and (ii) 2012 and 2013 apps with their pseudo-labels, where each app is converted to its feature vectors for model training. Finally, SDAC-FMU extends the initial feature sets with 2014 apps, converts each 2014 app to feature vectors according to the extended feature sets, and uses the classification models for 2014 apps to classify each 2014 app by its feature vectors.

3.3 Online Versions

Both SDAC-FEO and SDAC-FMU require that the current testing set T_N be available for performing feature extension before they can be applied to classify each individual app in this testing set. To overcome this restriction, we develop their online versions, SDAC-FEO-OL and SDAC-FMU-OL, in which the feature extension step is skipped, for classifying individual apps in time without waiting for the whole testing set to be available.

SDAC-FEO-OL. SDAC-FEO-OL is the same as SDAC-FEO in the training phase, which generates m initial feature sets and m classification models. When SDAC-FEO-OL is used to classify each app in T_1, it converts the app to m feature vectors according to m *initial feature sets*. Then, it uses m classification models that have been trained to output a predicted label for each app according its feature vectors. After all apps in T_1 have been classified, SDAC-FEO-OL generates m extended feature sets for T_1 the same way as SDAC-FEO does from the whole set T_1.

When SDAC-FEO-OL is used to classify each app in T_N ($N \geq 2$), it converts the app to m feature vectors according to m *extended feature sets for* T_{N-1}. Then, it uses the same classification models that have been trained to output a predicted label for each app in T_N via its feature vectors.

After all apps in T_N have been processed, SDAC-FEO-OL resorts to SDAC-FEO to process T_N again, generating m extended feature sets for T_N. This is to prepare SDAC-FEO-OL for detecting apps in the next time period.

SDAC-FMU-OL. SDAC-FMU-OL is the same as SDAC-FMU in the training phase, which generates m initial feature sets for T_1 and m classification models for T_1. SDAC-FMU-OL is the same as SDAC-FEO-OL when it is applied to classify each app in T_1 according to m initial feature sets for T_1 using m classification models for T_1.

With all apps in T_1 being processed, SDAC-FMU-OL resorts to SDAC-FMU to process T_1 again, generating m initial feature sets for T_2, and m classification models for T_2.

When SDAC-FMU-OL is applied on testing set T_N ($N \geq 2$), it converts each app to m feature vectors according to m initial feature sets for T_N. Then, it uses the m classification models for T_N to predict label for each app in T_N according its feature vectors. After all apps in T_N are processed, SDAC-FMU-OL performs feature and model updates the same way as SDAC-FMU does with the whole set T_N.

Notes. When an individual app is detected online, the app is first converted to m feature vectors according to *the set of APIs* it used, and then classified by m classification models. The first three steps of SDAC (API path extraction, API vector embedding, and API cluster generation and extension) are not performed in this process, which makes the online versions much faster than the offline ones.

SDAC-FEO-OL and SDAC-FMU-OL are different from direct applications of online machine learning in malware detection [11] since our online versions do not require true labels to be used for model updates, while online machine learning does require [12].

4 Evaluation of SDAC

Dataset. SDAC is evaluated using a dataset of around 36k benignware samples and 35k malware samples randomly chosen from an open Android application collection project [13]. Table 1 shows an overview of our dataset, which consists of benignware samples and malware samples developed in six years from 2011 to 2016. The time of each app is defined as the time its APK file was packaged, which can be found in the .dex file inside its APK [14].

The labels of the samples in our dataset were decided according to the reports from VirusTotal [15] which we obtained in July 2018. Based on the reports, we labelled apps with zero positive result as "benign", and apps whose reports containing more than a threshold T_{mal} positive results as "malicious".

In the literature of malware detection, different values of T_{mal} are used for labelling "malicious" apps. According to Roy et al., malware samples that received one positive report only from VirusTotal were considered to be of "low quality," and those received more than ten positives out of 54 scanners were considered "high quality" [3]. Arp et al. labelled an app as malicious if it received at least 20% positive results from a set of selected scanners [16]. Alex et al. performed a large-scale study on aggregating the results of scanners from Virustotal and deemed a malicious label to be trustful if it came from 4 or more positive scanning results out of 34 different scanners [17].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2020.3005088, IEEE Transactions on Dependable and Secure Computing

TABLE 1: Overview of Dataset($T_{mal} = 15$)

Year	2011	2012	2013	2014	2015	2016
Benign	6072	5887	5920	5934	5929	5903
Malware	4961	5953	5877	5902	5925	5879
# total APIs	14842	16213	17519	17714	17933	19933
# new APIs	n/a	3369	3407	2701	2902	4009
% new APIs	n/a	20.78	19.45	15.645	16.18	20.11

In our experiments, we evaluate SDAC on 3 different datasets labelled with $T_{mal} = 4, 9\ and\ 15$, respectively, out of total 63 scanners in Virustotal[1]. We set $T_{mal} = 15$ in the default case and show our results for $T_{mal} = 4$ and $T_{mal} = 9$ in section 4.5.

Table 1 shows the number of unique new APIs and the number of all APIs in different years from the dataset labelled with $T_{mal} = 15$. An API is considered to be new in a year if it is not used by any app that was developed before that year in our dataset.

Tools and Parameters. We choose the following tools and parameters for the evaluation of SDAC. In the API path extraction step, we choose FlowDroid to extract a directed call graph among program classes from each app [7]. The parameter d used for API path extraction is chosen to be the same as the window size for API vector embedding. In the API vector embedding step, we rely on the gensim toolkit [18] to implement the Skip-Gram model and derive API vectors from a set of apps, where we choose window size $S = 5$ and API vector size $K = 200$ (i.e., the dimension of API vector space).

In the API cluster generation and extension step, we choose the same-size k-means cluster algorithm from open-source data mining framework ELKI [10], and set the number of clusters $k = 1000$. In the classification model training and testing step, we choose linear SVM models as our classification models, and set the number of classification models $m = 9$, and the threshold $\tau = 3$ in model voting.

We tune these parameters, as well as other parameters (e.g., iteration times and learning rate for API vector embedding), to produce the best results when SDAC is trained and tested in cross validation using the same training set (2011 apps) under the constraint of our computing resources (a desktop computer with 3.3 GHz CPU and 12GB memory). These parameters are used across all experiments for the evaluation of SDAC on various testing sets.

Selection of Parameters: k, m and τ. The parameters k (as in k-means clustering algorithm), m (i.e. the number of classification models used by SDAC) and τ (i.e. the threshold used in model voting) are tuned for the best performance of SDAC in the cross-validation on the training set, which is the 2011 dataset in our experiments.

Fig. 6 shows the performance of SDAC in cross-validation with different k values. The F-score of SDAC increases rapidly from $k = 50$ to $k = 500$, and remains stable after $k = 1000$. Since a higher k costs more time on model training and testing, we choose $k = 1000$ in our experiments.

Fig. 7 shows how the numbers m of classification models and threshold τ are decided. In the cross-validation experiments, SDAC reaches its highest F-score (above 97%) when $m > 7$ and τ/m is around 30% to 40%. Since the overhead of SDAC is proportional to m, we choose $m = 9$ and $\tau = 3$, which reaches the highest F-score with the smallest m.

1. Lists of .apk file hashes in these datasets can be found at http://dx.doi.org/10.21227/rasc-k457.

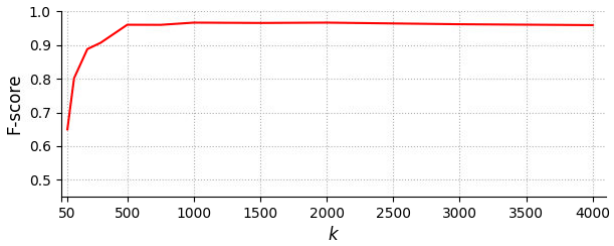


Fig. 6: F-score of SDAC in Cross Validation on 2011 Training Set with Different k

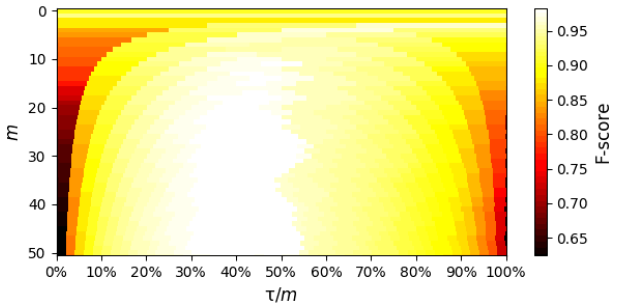


Fig. 7: F-score of SDAC in Cross Validation on 2011 Training Set with Different m and τ

4.1 Evaluation of SDAC-FEO

Three sets of experiments are conducted to evaluate SDAC-FEO. The first set is conducted to evaluate the accuracy of SDAC-FEO when it is trained and tested in cross validation using a set of samples developed in the same time period. The second set of experiments is conducted to evaluate the aging speed of SDAC-FEO when it is trained on a set of samples developed in one time period, and tested on other sets of samples developed in later time periods. The third set of experiments is conducted to compare the accuracy and aging speed of SDAC-FEO with MaMaDroid [5].

The accuracy of a malware detection model can be measured in F-score on a set of malware and on a set of benignware. F-score is the harmonic mean of $precision$ and $recall$, where $precision = |TP|/(|TP| + |FP|)$ and $recall = |TP|/(|TP| + |FN|)$. We use TP (i.e., true positives) to denote the set of malware that is correctly detected as malware, FP (i.e., false positives) the set of benignware that is incorrectly detected as malware, FN (i.e., false negatives) the set of malware that is incorrectly detected as benignware, and TN (i.e., true negatives) the set of benignware that is correctly detected as benign.

The accuracy of SDAC-FEO is first evaluated in 5-fold cross validation using samples developed in the same time period. Table 2 shows the accuracy of SDAC-FEO in terms of precision, recall, and F-score on different set of apps, which is denoted by the time period in which the apps were developed. The average F-score of SDAC-FEO is 98.25%, which serves as a good starting point for evaluating SDAC-FEO over time.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2020.3005088, IEEE Transactions on Dependable and Secure Computing

TABLE 2: F-score of SDAC-FEO in Cross Validation

App set	Precision	Recall	F-Score
2011	0.9885	0.9672	0.9778
2011~12	0.9918	0.9842	0.9880
2011~13	0.9906	0.9848	0.9877
2011~14	0.9915	0.9829	0.9872
2011~15	0.9905	0.9812	0.9858

SDAC-FEO Performance Over Time. The aging property of SDAC-FEO is evaluated in a series of experiments in which SDAC-FEO is trained on a set of samples developed in one time period and tested on other sets of samples developed in later time periods. Fig. 9 shows the F-score of SDAC-FEO in detection over time. When SDAC-FEO is evaluated on a testing set that is newer than the training set by one year, its average F-score is 97.49%, which declines by 1.03% from its average F-score in cross validation (98.52%). It declines further to 95.02%, 88.48%, 78.22%, 73.72% when SDAC-FEO is evaluated on testing sets that are newer than the training sets by two, three, four, and five years, respectively. The average aging speed of SDAC-FEO is 4.96% in F-score per year in these experiments.

Analysis on API Cluster Extension. API cluster extension is a critical step in SDAC. In this step initial feature sets are extended with new APIs to create the extended feature set. This enables SDAC to evaluate new APIs' contributions to malware detection with the existing classification models, which have been trained by a set of labelled apps in which none of the new APIs are used. Several case studies of new APIs being added to existing API clusters are described in Appendix C.

To further understand how API cluster extension contributes to slow aging of SDAC-FEO, in this section, we calculate the changes on feature vectors of testing set apps with and without feature extension, and then extract each feature's weight in the linear SVM model which helps to figure out how such change will affect the detection results.

In detail, in the SVM model used by SDAC-FEO, the inner product between an app's feature vector and the linear SVM model's weight vector is the *output score* of the app's feature vector in the SVM model [16], which represents the confidence of the SVM model in classifying the app as either malware if the output score is positive, or benignware if the output score is negative. The confidence of an SVM model is proportional to the absolute value of an output score.

For each app in a testing set, we transform it to two feature vectors according to an initial feature set and an extended feature set for each SVM model, respectively. The *output score difference* of an app is defined as the output score of its feature vector derived from the extended feature set subtracted by the output score of its feature vector derived from the initial feature set. The output score difference of an app in an SVM model represents the change in confidence on the app caused by API cluster extension. A positive (negative, respectively) output score difference means more confidence in classifying an app as malware (benignware, respectively).

Then, we examine the average output score difference for all malware samples in a testing set and for all classification models used by SDAC-FEO. We also examine the

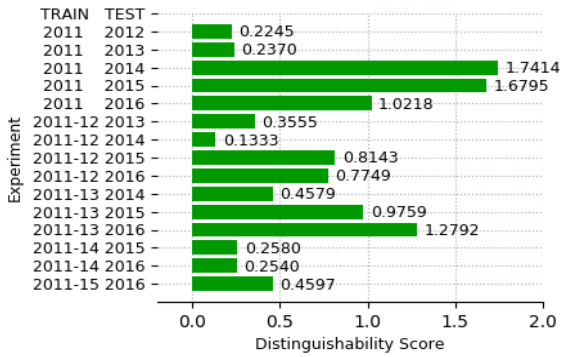


Fig. 8: Distinguishability of API Cluster Extension

average output score difference for all benignware samples in the same testing set. The *distinguishability of API cluster extension* in each experiment (with a training set and a testing set) is defined as the average output score difference for all malware samples subtracted by the average output score difference for all benignware samples. If the distinguishability of API cluster extension is positive, then the API cluster extension makes a positive contribution to malware detection, and thus contributes to slow aging of SDAC-FEO.

Fig. 8 shows the *distinguishability* brought by API cluster extension in our experiments. The contributions of API cluster extension to malware detection are positive in all of our experiments, which demonstrates that the feature extension will indeed contribute to the accuracy of SDAC.

Aging Slower Than MaMaDroid. The aging speed of SDAC-FEO is compared with MaMaDroid using the same dataset. MaMaDroid is the only solution which we know to be resilient to the changes in Android specifications, and has a significant better performance than other solutions such as DroidAPIMiner [5]. In particular, MaMaDroid first derives API paths from each app using FlowDroid, and abstracts APIs to their corresponding packages (or package families) in the API paths. It then summarizes all abstracted paths to a Markov model, and converts the Markov model to a feature vector for each app, where each feature in the feature vector represents a transition between two existing packages in the Markov model. After that, it trains a machine learning classification model from a training set of apps according to their feature vectors and associated true labels. By abstracting APIs to packages, MaMaDroid is resilient to the adding of new APIs to existing packages in Android specifications; however, it is not designed to be resilient to the adding of new API packages.

We re-implemented MaMaDroid using its sourcecode [19]. Both SDAC-FEO and MaMaDroid rely on FlowDroid to decompile app Apk files; they were evaluated using the same training sets, testing sets, standard SVM classification models, and F-score measurement.

We note that in the original MaMaDroid paper [5], the random forests algorithm produces the best malware detection results among four classification algorithms, including random forests, 1-NN, 3-NN, and SVM. However, in our experiments, MaMaDroid performs the best with SVM. Please refer to Appendix D for the comparison of MaMaDroid with different classification algorithms on our datasets. Therefore, we choose SVM as the classification

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2020.3005088, IEEE Transactions on Dependable and Secure Computing

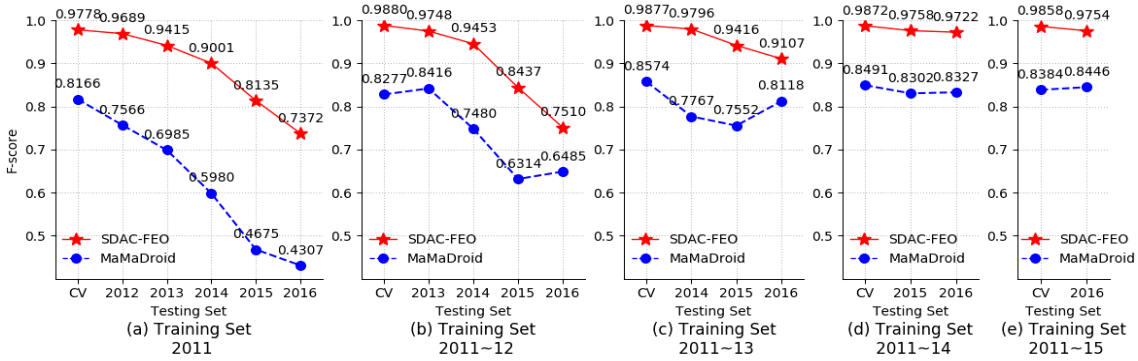


Fig. 9: Comparison between SDAC-FEO and MaMaDroid (CV: 5-fold cross validation)

model for MaMaDroid in our experiments.

Fig. 9 shows that the performance of SDAC-FEO is significantly and consistently better than MaMaDroid in all experiments. In particular, the average F-scores of SDAC-FEO when it is evaluated on testing sets that are newer than training sets by one to five years are 97.49%, 95.02%, 88.48%, 78.22%, and 73.72%, respectively. In comparison, the average F-scores of MaMaDroid in the corresponding cases are 81.00%, 75.86%, 68.05%, 55.80%, and 43.07%, respectively. The average F-score of SDAC-FEO is higher than MaMaDroid by 20.65% when they are evaluated on the same training set and testing set across all experiments. In terms of aging speed, SDAC-FEO declines by 4.96% in F-score per year on average over five years, while MaMadroid declines by 8.15% in the same case.

We notice that the performance of MaMaDroid in our experiments is not as good as what was reported in [5]. One possible reason is that overlaps exist between the training sets and the testing sets used in [5], where two benign sets were collected from PlayDrone [20] and Google Play store, and five malware sets were collected from Drebin [16] and VirusShare [21]. For one set of experiments in [5], the same "oldbenign" benign set was used to mix with various malware sets dated from 2012 to 2016 to form training sets and testing sets. For another set of experiments, the same "newbenign" benign set was used to form all training sets and testing sets. Such overlaps made it difficult to evaluate how MaMaDroid aged over time.

4.2 Evaluation of SDAC-FMU

Aging Slower Than SDAC-FEO and MaMaDroid. Compared to SDAC-FEO, SDAC-FMU takes additional steps of feature update and model update for better performance. Fig. 10 shows that its performance is significantly and consistently better than SDAC-FEO. The average F-score of SDAC-FMU (97.39%) is higher than SDAC-FEO (92.89%) by 4.50% when they are evaluated on the same training set and testing set across all experiments. In terms of aging speed, SDAC-FMU declines by 0.25% in F-score per year on average over five years, while SDAC-FEO declines by 4.96% in the same case.

Since SDAC-FMU updates its classification models with pseudo-labels, we also update MaMaDroid classification models with pseudo-labels for a fair comparison. Fig. 10 also shows that the performance of MaMaDroid updated with pseudo-labels is almost the same as before. One possible reason is that MaMaDroid neglects the APIs of new packages in generating these pseudo-labels, and the updating will then reinforce the mistakes caused by neglecting these new APIs in MaMaDroid's model.

False Positives. We examine the misclassified results of SDAC-FMU when it is trained on 2011 set and evaluated on five testing sets, dated from 2012 to 2016. To understand why false positives are misclassified, we compute a *weight for an API* by averaging the weights of the features that contain this API in all SVM models when SDAC-FMU is applied to each testing set. The weight of an API can be used to measure its contribution to the confidence of SDAC-FMU in classifying an app. We sort all APIs according to their weights for each testing set. We choose the top p and the bottom p APIs in each sorted list, which are the APIs with most contributions to the confidence of SDAC-FMU in classifying an app as malware and as benignware, respectively.

For a set of apps, we define *API ratio for an API* as the percentage of the apps in the set that use this API. We further define *top-p ratio* (*bottom-p ratio*, respectively) as the average of API ratios for the top p APIs (bottom p APIs, respectively). The top-p ratio subtracted by the bottom-p ratio for a set of apps means the confidence of SDAC-FMU in classifying the set as malware.

Fig. 11 shows typical values of top-p ratio subtracted by bottom-p ratio for true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN), where $p = 1\%$. Compared to TN, SDAC has more confidence in classifying FP as malware, and the most confidence in classifying TP as malware.

In the list of top-weighted APIs, we discovered some APIs such as *getConfiguration()* and *getDeviceId()* that were considered as "dangerous" in previous research [16], [5]. Benign apps using such APIs are more likely to be detected as malware by SDAC. For example, 40% (126/315) of false positives and more than a half of true positives (3310/5890) use API *getConfiguration()*, while only 23% (1289/5555) of true negatives makes use of it. For another example, 66% (208/315) of false positives and 96% (5633/5890) of true positives include API *getDeviceId()* in class *TelephonyManager*, while only 22% (1245/5555) of true negatives use it. Please refer to Appendix E for 1% top-weighted APIs which contribute to the false positive results.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2020.3005088, IEEE Transactions on Dependable and Secure Computing

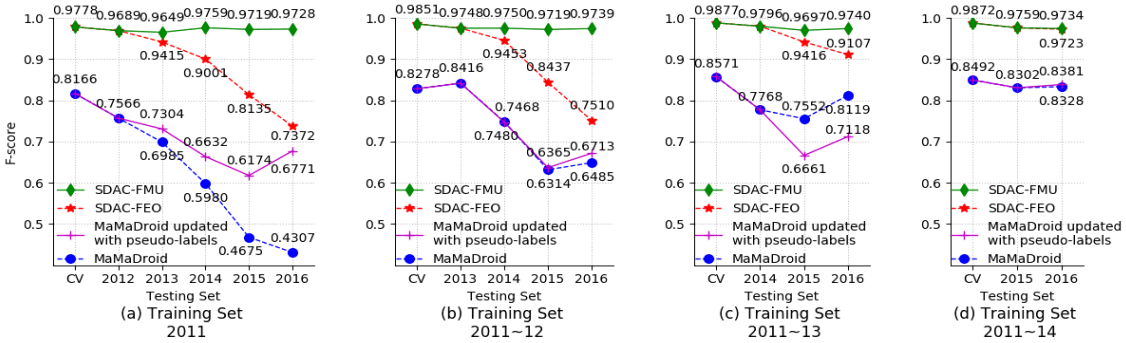


Fig. 10: Comparison between SDAC-FMU, SDAC-FEO, and MaMaDroid (CV: 5-fold cross validation)

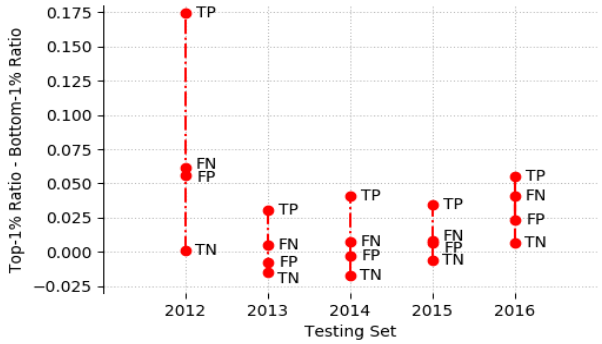


Fig. 11: Difference in API Usage among TP, FP, FN, and TN with 2011 Training Set

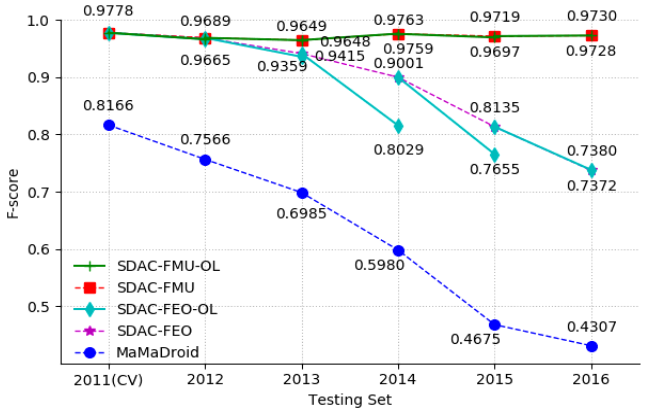


Fig. 12: Evaluation of Online Versions with 2011 Training Set (CV: 5-fold cross validation)

False Negatives. We also examine the false negatives of SDAC-FMU when it is trained on 2011 set and evaluated on five testing sets, dated from 2012 to 2016. Among 1076 false negative samples generated from all testing sets, about 69% (638/931) of them are classified as "positive: adware" by at least one VirusTotal scanner. According to TrendMicro [22], the adware apps may come from repackaging benign apps with 3rd party advertisement libraries; it is difficult for SDAC-FMU to distinguish them from true benign apps.

Besides, about 8.5% (79/931) false negative samples are classified as "positive: riskware", and about 17.4% (162/931) as "positive: not-a-virus" by at least one VirusTotal scanner. According to the explanation from Kaspersky Lab [23], riskware refers to legtimate programs which are easy to be exploited by malicious attackers, and not-a-virus is associated with adware and riskware.+

Fig. 11 also shows that compared to TP, SDAC has more confidence in classifying FN as benignware, and the most confidence in classifying TN as benignware.

4.3 Evaluation of SDAC-FEO-OL & SDAC-FMU-OL

The performances of SDAC-FEO-OL and SDAC-FMU-OL are evaluated in the default case, which is formed with the smallest training set (2011 apps) and the longest time span across testing sets (2012-2016) in our experiments. Fig. 12 shows that SDAC-FMU-OL performs very closely to SDAC-FMU, while the performance gap between SDAC-FEO-OL and SDAC-FEO is more obvious. Compared to SDAC-FMU, the F-score of SDAC-FMU-OL declines by 0.41% on average, by -0.29% in minimum (for 2014 testing set), and by 0.85% in maximum (for 2012 testing set). Compared to SDAC-FEO, the F-score of SDAC-FEO-OL declines by 3.45% on average, by -0.30% in minimum (for 2013 testing set), and by 7.97% in maximum (for 2014 testing set). Nonetheless, both SDAC-FMU-OL and SDAC-FEO-OL perform significantly better than MaMaDroid.

4.4 Runtime Performance

The runtime performance of SDAC is evaluated on a desktop computer using one Intel(R) i5-4590 3.3 GHz CPU and 12 GB physical memory running on the Ubuntu 14.04 (LTS) operating system. Table 3 shows the runtime performance of SDAC in all four steps: (i) API Path Extraction, (ii) API Vector Embedding, (iii) API Cluster Generation and Extension, and (iv) Classification Model Training and Testing.

Runtime of SDAC-FEO-OL and SDAC-FMU-OL in Detection Phase. SDAC-FEO-OL can be used to detect individual apps online without waiting for a whole testing set to be available. The time cost for detecting an app online is 0.20 seconds on average. Once a whole testing set is available, SDAC-FEO-OL extends its feature sets using the whole testing set in the same way as SDAC-FEO does. This additional time cost is similar to SDAC-FEO in its detection phase.

The time cost of SDAC-FMU-OL is the same as SDAC-FEO-OL for detecting an app online. Once a whole testing set is available, SDAC-FMU-OL performs feature and model updates in the same way as SDAC-FMU, so the additional time cost is also same as in SDAC-FMU.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2020.3005088, IEEE Transactions on Dependable and Secure Computing

TABLE 3: Runtime Performance of SDAC

Step		Runtime
API Path Extraction	Call Graph Generation	Avg. 37.29 sec. per app (min. 3.12s & max. 1184.33s)
	API Path Extraction	Avg. 16.82 sec. per app (min. 8.30e-4s & max. 1350.06s)
API Vector Embedding	Transform APIs into Vectors	333.20 sec. (11033 apps) ~ 3154.48 sec. (58360 apps)
API Cluster Generation and Extension	API Cluster Generation (in training phase)	85.91 sec. (11033 apps) ~ 300.33 sec. (58360 apps)
	API Cluster Extraction (in testing phase)	48.93 sec. (11033 apps) ~ 25.77 sec. (58360 apps)
Classification Model Training and Testing	Classification Model Training (in training phase)	9.77 seconds (11033 apps) ~ 1683.75 seconds (58360 apps)
	Classification Model Testing (in testing phase)	Avg. 8.90e-4 sec. per app (min. 4.57e-4s & max. 1.59e-3s)

Runtime of MaMaDroid. MaMaDroid takes three major steps in both training phase and detection phase: (i) FlowDroid is exploited to derive a set of API paths from an app, (ii) a Markov model is formed from a set of API paths, and then used to compose a feature vector, and (iii) a classification model is trained from (in training phase) or applied to (in detection phase) a set of apps. In our implementation, the training phase of MaMaDroid takes 37.29 seconds on average in step one, 0.41 seconds on average in step two, and 16.3 seconds (785.29 seconds, respectively) for processing a set of 11,033 apps (58,360 apps, respectively) in step three. In its detection phase, MaMaDroid takes 0.0036 seconds on average for classifying a single app in step three, while the first two steps take the same time as in the training phase.

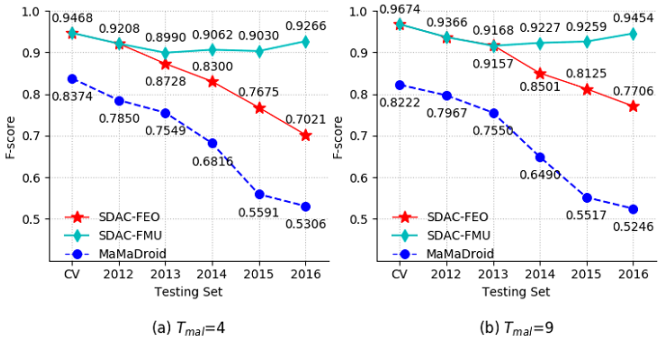
Runtime Performance Comparison. In the training phase, SDAC spends more time (54.17 seconds per app on average) than MaMaDroid (37.70 seconds per app on average) for transforming decompiled codes into feature vectors. In the next step of classification model training, which refers to model training in SDAC-FEO or model updating in SDAC-FMU, the time cost ranges from 9.77 seconds (on the smallest training set containing 11033 apps) to about 0.47 hours (on the largest training set containing 58033 apps) for training each classification model. Since 9 classification models are used in SDAC, the total time cost of SDAC for this step ranges from 87.93 seconds to about 4.2 hours. In comparison, MaMaDroid spends 16.3 seconds to about 13.1 minutes on different training sets in the model training step. In the detection phase, the average time for detecting each app is 8.90e-4 seconds for SDAC, and 3.6e-3 seconds for MaMaDroid.

Although SDAC takes longer training time than MaMaDroid, it achieves much higher accuracy and slower aging speed as shown in sections 4.1 to 4.3. The runtime performance of SDAC is acceptable in all out experiments even though they are conducted on a common desktop computers (i.e., i5-4590@3.3GHz CPU and 12GB memory).

Notes. For both SDAC-FMU and SDAC-FMU-OL, the time cost for feature and model updates increases with the size of its input data, which is the union of its training set and all past testing sets. The size of the input data keeps increasing as more testing sets are processed and accumulated over time. To address this problem, we suggest to apply a validation window to the input data which covers all past testing sets starting from the last testing set in which all apps' true labels are available[1]. The size of this validation window is limited, and so is the time cost for feature and model updates.

The time granularity in forming testing sets is mainly decided by the number of apps that were collected within each time granularity, and each testing set should be large enough to extract accurate API context information from it. We suggest to choose each testing set to be larger than 5500 apps based on our experience with SDAC.

1. Relaxing our assumption, we believe that true labels of testing apps will be finally available after a limited period of time.



(a) T_{mal}=4 (b) T_{mal}=9

Fig. 13: Comparison between SDAC-FMU, SDAC-FEO, and MaMaDroid with $T_{mal} = 4$ and $T_{mal} = 9$ (CV: 5-fold cross validation)

4.5 Evaluation of SDAC with Different T_{mal}

The performance of SDAC-FEO and SDAC-FMU are also evaluated on datasets that are labelled with different positive threshold $T_{mal} = 4$ and $T_{mal} = 9$ in VirusTotal reports. We also run MaMaDroid on these datasets for performance comparison.

Fig. 13 shows the F-measurements of SDAC when $T_{mal} = 4$ and $T_{mal} = 9$, respectively. In both cases, the 2011 app set is used as the training set and the 2012~2016 data sets are used as the testing sets. In the case of $T_{mal} = 4$, SDAC-FEO declines by 4.89% in F-score and SDAC-FMU declines by 0.40% per year, while MaMaDroid declines by 6.12% per year on average. The advantages of SDAC-FEO and SDAC-FMU over MaMaDroid are 14.85% and 22.55%, respectively, in F-score on average. When $T_{mal} = 9$, the average aging speed in F-score is 3.94% for SDAC-FEO, 0.44% for SDAC-FMU, and 5.95% for MaMaDroid. The advantages of SDAC-FEO and SDAC-FMU over MaMaDroid are 19.22% and 25.22%, respectively, in F-score on average.

4.6 Evaluation of SDAC with Unbalanced Datasets

On our balanced datasets, SDAC outperforms MaMaDroid significantly. However, according to a recent research

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2020.3005088, IEEE Transactions on Dependable and Secure Computing

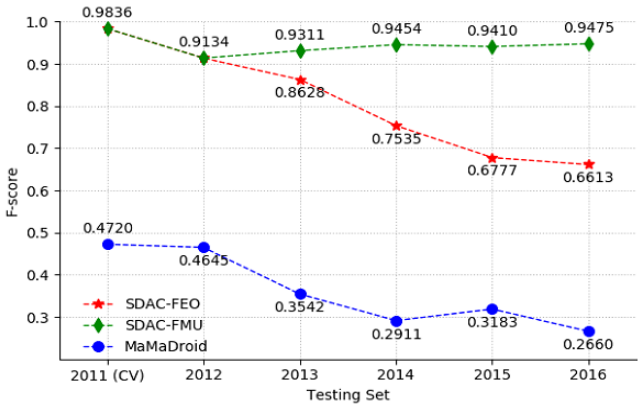


Fig. 14: Comparison between SDAC-FMU, SDAC-FEO and MaMaDroid with Datasets of Unbalanced Ratio (CV: 5-fold cross validation)

project, a balanced dataset may lead to biased results in malware detection since malware is usually the minority class (as compared to benignware) in the wild. It was reported that the ratio of malware is around 10% in a real-world setting [24]. To estimate SDAC's performances in this case, we downsample malware to make its ratio to be 10% out of all apps, and form new datasets in our evaluation. The performances of SDAC on such datasets are shown in figure 14, which demonstrate wider differences between SDAC and MaMaDroid, and a similar trend as shown in our previous evaluations.

5 Discussions

5.1 SDAC against Obfuscation

Code obfuscation tools, such as DroidChameleon [25] or [26] are often used to obfuscate malicious apps to avoid detection. Since SDAC presents its detection based on Android APIs, the obfuscation methods can be mainly classified into three categories by their impacts on the APIs used in apps: category-I: the set of APIs used by an app is not changed in obfuscation (e.g. encrypt native exploit or payload, rename identifier, identifiers or package), category-II: the set of APIs used by an app is enlarged to include new APIs in obfuscation (e.g. repackage, or insert junk code) and category-III: the set of APIs used by an app is reduced during obfuscation (e.g. method call hiding or reflection[27], [28]).

SDAC is naturally robust to category-I obfuscation methods since it detects an app solely based on the set of APIs used by it. Category-II obfuscation methods, such as "insert junk code" and "repackage malware into benignware", may enlarge the set of APIs used by an app, and thus avoid detection by SDAC. To test the robustness of SDAC against category-II methods, we generate a collection of 10,000 API sets for each testing set. Each API set in that collection is the union of the API set used by a benign sample and that by a true-positive malware sample randomly chosen from the testing set. One united API set is considered to be derived from a virtual malicious app created by "injecting" the code of a malware sample into the code of a benign sample. Then SDAC is applied to such virtual malicious apps to check its recall rates in various modes on different testing sets. Table 4 shows that the recall rates of SDAC in all modes are higher than 65% in the first two years, indicating that they can still detect a majority of obfuscated malware in such cases.

TABLE 4: Robustness of SDAC (in recall rate) against Category-II Obfuscation with 2011 Training Set

Testing set	2012	2013	2014	2015	2016
SDAC-FMU	81.41%	69.87%	59.85%	49.88%	36.54%
SDAC-FEO	81.41%	67.22%	64.23%	36.49%	37.79%
SDAC-FMU-OL	77.63%	81.57%	60.98%	48.82%	47.49%
SDAC-FEO-OL	77.63%	65.49%	62.23%	58.03%	45.17%

The category-III obfuscation methods are mostly achieved by making certain malicious codes loaded at runtime [26], which is invisible from static analysis. In general, no static analysis is robust against category-III obfuscation methods. Nonetheless, SDAC can be potentially applied in dynamic analysis since the API call sequences captured in dynamic analysis can be directly used as the input to the API embedding step in SDAC. It remains interesting to test the robustness of SDAC in dynamic analysis against category-III obfuscation methods in the future.

Obfuscation by App Packing. Packing technique is also an effective method to apply obfuscation on apps and hide their codes. Various kinds of packing techniques are widely adopted by malware developers as reported in [29]. These mechanisms protect packed malware against reverse analysis and thus thwart path extracting and feature vector generating in SDAC. However, it is still potential to combine SDAC with either dynamic analysis tools or Android unpackers [29], [30], [31] against the packed malware.

5.2 API Semantic Extraction

Besides sequential APIs from which SDAC extracts API semantics, data dependency is another way to analyze APIs' relationship and extract their semantics. This method has been used in previous research such as DarkHazard [32].

SDAC currently focuses on malware detection based on sequential API analysis, which is complementary to data dependency analysis. Some APIs may have direct data dependency with each other but do not have any sequential relationship. On the other hand, data dependency analysis may miss some API relationship in malware detection. For example, in an Android malware detection solution proposed by Wang et, al. in [33], it is found that data dependency information is lost in self-defined methods and thus may result in malicious behaviors being undetected. While in SDAC, these self-defined methods are collected together with their caller methods and callee methods, and then used in generating API sequences. It would be interesting to extend SDAC to cover data dependency relationships in semantic extraction in a future work.

5.3 Limitations

In our experiments, FlowDroid is used in the first step of SDAC for extracting API paths from Android apps. It is observed that FlowDroid fails to process 2.89% (1053/36490) benign samples and 1.74% (610/35106) malware samples among all the apps we collected originally, these samples

are excluded in our experiments. Some failures are due to exceeding memory limit in the extracting (i.e., 4 GB in our experiments for API path extraction under the Soot tool). To address this problem, one may use more powerful computers with larger memory, or rely on other static analysis tools such as Amandroid [34] and Androguard [35] for API path extraction.

Another limitation in our experiments is that FlowDroid does not cover HTML5 codes, native codes, or the codes which are loaded at runtime. It is a future direction to extend SDAC to cover such codes by performing dynamic analysis.

6 Related Work

Android Framework Evolvement. Android apps rely on Android APIs to perform their functions, and many APIs are added or deprecated in Android specifications over time. The impact of API evolution to the usability of apps has been studied recently. For example, McDonnell, Ray, and Kim investigated how Android app developers follow and adopt Android API changes over time [36]. Linares-Vásquez et al. studied the relationship between API change and fault proneness, and evaluated its threat to the success of Android apps [37]. Brito et al. studied the adoption of API deprecation messages and its impact to software evolution [38]. Recently, Wu et al. focused on inconsistency between the versions of declared Android API frameworks and the actual ones used in Android apps [39]. While most of previous research in this area focused on the usability of apps, no rigorous study has been conducted on the impact of Android framework evolvement to malware detection.

Android Malware Detection. Android malware detection can be categorized into static analysis and dynamic analysis (e.g., [40], [41], [42], [43], [44], [45]). SDAC belongs to static analysis though its approach could be extended to dynamic analysis in the future.

Static analysis detects Android malware according to the information extracted from app APK files. It can be further categorized into signature based solutions (e.g., [46], [47], [48]) and learning based solutions. We briefly summarize some learning based solutions that are more closely related to SDAC than other solutions.

A wide variety of features have been examined in developing learning based solutions. For example, Arp et al. devised Drebin to extract eight classes of features (e.g., network addresses, component names, permissions, and API calls) from manifest files and disassembled codes. Avdienko et al. examined the difference in sensitive data flows between malware and benignware [49]. Yang et al. designed DroidMiner to extract malicious behavior patterns from APIs and framework resources [50]. In another work, Ke, Li, and Deng devised ICCDetector to extract inter-component communication features from app components [51].

In addition, DroidAPIMiner proposed by Aafer, Du, and Yin extracts a set of API-level features including critical API call frequencies, framework classes, and API parameters [52]. DroidSIFT proposed by Zhang et al. extracts weighted contextual API dependency graphs from apps based on sensitive APIs [53]. MAST proposed by Chakradeo et al. examined strong relationships between declared indicators of application functionality (e.g., permissions, intent filters, and the presence of native code) [54]. Many other types of features are also used in learning based solutions (e.g., [55], [56], [57], [58], [59], [60]).

A common feature of the feature sets in most learning based solutions is that they are "static", not keeping up with the evolvement of Android frameworks. Consequently, the accuracy of such solutions may decline significantly over time (i.e., model aging), which has been observed in both industry (e.g., [2]) and academia (e.g., [1], [3]) recently. While MaMaDroid is resilient to the adding of new APIs to existing packages in Android specifications [5], the model aging problem is largely unsolved before because many new packages are continually added in Android framework.

7 Conclusion

In this paper, we designed a novel slowing aging solution named SDAC for Android malware detection. The key drivers to achieve slow aging in SDAC include (i) clustering APIs based on the semantic distances among APIs, (ii) evaluating a new API's contribution to malware detection using existing APIs' based on API clusters, and (iii) updating API clusters and classification models based on both training data with true labels and testing data with pseudo-labels. The best versions of SDAC achieve both high accuracy with average F-score 97.49%, and slow aging speed with average F-score decline 0.11% per year over five years in our experiments. The other versions have lower requirements on computing resources, but still perform better than the state of the art.

References

[1] Z. Zhu and T. Dumitras, ""Featuresmith: Automatically engineering features for malware detection by mining the security literature"," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[2] T. L. Wang, ""AI-Based Antivirus: Detecting Android malware variants with a deep learning system"," in *Blackhat Europe*, 2016.

[3] S. Roy, J. DeLoach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara, ""Experimental study with real-world data for Android app security analysis using machine learning"," in *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2015.

[4] J. Allen, M. Landen, S. Chaba, Y. Ji, S. P. H. Chung, and W. Lee, ""Improving accuracy of Android malware detection with lightweight contextual awareness"," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*. ACM, 2018.

[5] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, ""MaMaDroid: Detecting Android malware by building Markov Chains of behavioral models"," in *The Network and Distributed System Security Symposium (NDSS)*, 2017.

[6] "Android developer documentation," https://developer.android.com/reference/classes.

[7] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, ""Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps"," *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[8] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, ""Appintent: Analyzing sensitive data transmission in android for privacy leakage detection"," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2013.

[9] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, ""Distributed representations of words and phrases and their compositionality"," in *Advances in Neural Information Processing Systems (NIPS)*, 2013.

[10] A. Z. Erich Schubert, ""ELKI data mining"," https://elki-project.github.io/, 2016.

[11] A. Narayanan, L. Yang, L. Chen, and L. Jinliang, ""Adaptive and scalable android malware detection through online learning"," in *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2016.

[12] Y. Wu, S. C. Hoi, C. Liu, J. Lu, D. Sahoo, and N. Yu, ""Sol: A Library for Scalable Online Learning Algorithms"," *Neurocomputing*, vol. 260, pp. 9–12, 2017.

[13] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, ""Androzoo: Collecting millions of android apps for the research community"," in *IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016.

[14] P. Palumbo, L. Sayfullina, D. Komashinskiy, E. Eirola, and J. Karhunen, ""A pragmatic Android malware detection procedure"," *Computers & Security*, vol. 70, pp. 689–701, 2017.

[15] "VirusTotal," https://www.virustotal.com/, 2004.

[16] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, ""DREBIN: Effective and explainable detection of Android malware in your pocket."," in *The Network and Distributed System Security Symposium (NDSS)*, 2014.

[17] A. Kantchelian, M. C. Tschantz, S. Afroz, B. Miller, V. Shankar, R. Bachwani, A. D. Joseph, and J. D. Tygar, ""Better malware ground truth: Techniques for weighting anti-virus vendor labels"," in *8th ACM Workshop on Artificial Intelligence and Security (AISec)*. ACM, 2015.

[18] R. Řehůřek and P. Sojka, ""Software framework for topic modelling with large corpora"," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA, 2010.

[19] Mariconti, "Mamadroid project," https://bitbucket.org/gianluca_students/mamadroid_code, 2017.

[20] N. Viennot, E. Garcia, and J. Nieh, ""A measurement study of google play"," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1. ACM, 2014, pp. 221–233.

[21] "Virusshare project," https://virusshare.com/, 2011.

[22] Trendmicro, ""A Case of Misplaced Trust: How a Third-Party App Store Abuses Apple's Developer Enterprise Program to Serve Adware"," https://blog.trendmicro.com/trendlabs-security-intelligence/how-a-third-party-app-store-abuses-apples-developer-enterprise-program-to-serve-adware/, 2016.

[23] KasperskyLab, ""What is Riskware?"," https://www.kaspersky.com/resource-center/threats/riskware, 2017.

[24] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, ""TESSERACT: Eliminating experimental bias in malware classification across space and time"," *arXiv*, 2018.

[25] V. Rastogi, Y. Chen, and X. Jiang, ""Droidchameleon: evaluating android anti-malware against transformation attacks"," in *ACM Asia Conference on Computer and Communications Security (ASIACCS)*, 2013.

[26] "DexProtector," https://dexprotector.com/.

[27] A. Apvrille and R. Nigam, ""Obfuscation in Android Malware, and How to Fight Back"," *Virus Bulletin*, pp. 1–10, 2014.

[28] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer, ""Andrubis–1,000,000 Apps Later: A View on Current Android Malware Behaviors"," in *Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. IEEE, 2014.

[29] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, ""Things you may not know about Android (Un) Packers: A systematic study based on whole-system emulation"." in *The Network and Distributed System Security Symposium (NDSS)*, 2018.

[30] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, ""Understanding Android app piggybacking: A systematic study of malicious code grafting"," *IEEE Transactions on Information Forensics and Security (TIFS)*, 2017.

[31] W. Yang, Y. Zhang, J. Li, J. Shu, B. Li, W. Hu, and D. Gu, ""Appspear: Bytecode decrypting and dex reassembling for packed Android malware"," in *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2015.

[32] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, ""Dark Hazard: Learning-based, large-scale discovery of hidden sensitive operations in Android apps."," in *The Network and Distributed System Security Symposium (NDSS)*, 2017.

[33] X. Wang, S. Zhu, D. Zhou, and Y. Yang, ""Droid-AntiRM: Taming control flow anti-analysis to support automated dynamic analysis of Android malware"," in *Proceedings of the 33th Annual Computer Security Applications Conference (ACSAC)*, 2017.

[34] F. Wei, O. X. Roy, Sankardas, and Robby, ""Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps"," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2014.

[35] A. Desnos, ""Androguard: Reverse engineering, malware and goodware analysis of Android applications... and more (ninja!)"," http://code.google.com/p/androguard, 2015.

[36] T. McDonnell, B. Ray, and M. Kim, ""An empirical study of API stability and adoption in the android ecosystem"," in *IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2013.

[37] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, ""API change and fault proneness: A threat to the success of Android apps"," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2013.

[38] G. Brito, A. Hora, M. T. Valente, and R. Robbes, ""Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems"," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 360–369.

[39] D. Wu, X. Liu, J. Xu, D. Lo, and D. Gao, ""Measuring the declared SDK versions and their consistency with API calls in android apps"," in *International Conference on Wireless Algorithms, Systems, and Applications (WASA)*. Springer, 2017.

[40] L.-K. Yan and H. Yin, ""DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis."," in *USENIX security symposium*, 2012.

[41] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, ""TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones"," *ACM Transactions on Computer Systems (TOCS)*, 2014.

[42] L. Chen, M. Zhang, C.-y. Yang, and R. Sahita, ""Semi-supervised classification for dynamic Android malware detection"," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[43] P. Lantz, ""Droidbox - Android application sandbox"," https://github.com/pjlantz/droidbox, 2014.

[44] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, ""FLEXDROID: Enforcing in-app privilege separation in Android."," in *The Network and Distributed System Security Symposium (NDSS)*, 2016.

[45] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Kruegel, G. Vigna, A. Doupé, and M. Polino, ""Going Native: Using a large-scale analysis of Android apps to create a practical native-code sandboxing policy."," in *The Network and Distributed System Security Symposium (NDSS)*, 2016.

[46] S. Dai, A. Tongaonkar, X. Wang, A. Nucci, and D. Song, ""Networkprofiler: Towards automatic fingerprinting of Android apps"," in *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2013.

[47] W. Enck, M. Ongtang, and P. McDaniel, ""On lightweight mobile phone application certification"," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2009.

[48] Y. Feng, S. Anand, I. Dillig, and A. Aiken, ""Apposcopy: Semantics-based detection of Android malware through static analysis"," in *International Conference on Software engineering (ICSE)*. ACM, 2014.

[49] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, ""Mining apps for abnormal usage of sensitive data"," in *International Conference on Software engineering (ICSE)*. IEEE, 2015.

[50] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, ""DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications"," in *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2014.

[51] K. Xu, Y. Li, and R. H. Deng, ""ICCDetector: ICC-based malware detection on Android"," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 11, no. 6, pp. 1252–1264, 2016.

[52] Y. Aafer, W. Du, and H. Yin, ""DroidAPIMiner: Mining API-level features for robust malware detection in Android"," in *EAI International Conference on Security and Privacy in Communication Networks (SecureComm)*. Springer, 2013.

[53] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, ""Semantics-aware Android malware classification using weighted contextual api

dependency graphs",," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2014.

[54] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, ""Mast: Triage for market-scale mobile malware analysis",," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. ACM, 2013.

[55] S. Hou, Y. Ye, Y. Song, and M. Abdulhayoglu, ""Hindroid: An intelligent Android malware detection system based on structured heterogeneous information network",," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. ACM, 2017.

[56] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, ""Droid-sec: deep learning in Android malware detection",," in *ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2014.

[57] F. Idrees, M. Rajarajan, M. Conti, T. M. Chen, and Y. Rahulamathavan, ""PIndroid: A novel Android malware detection system using ensemble learning methods",," *Computers & Security*, vol. 68, pp. 36–46, 2017.

[58] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, ""Appcontext: Differentiating malicious and benign mobile app behaviors using context",," in *International Conference on Software engineering (ICSE)*. IEEE, 2015.

[59] M. Lindorfer, M. Neugschwandtner, and C. Platzer, ""Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis",," in *IEEE Computer Software and Applications Conference (COMPSAC)*. IEEE, 2015.

[60] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, ""Stormdroid: A streaminglized machine learning-based system for detecting Android malware",," in *ACM Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, 2016.

Robert H. Deng Robert H. Deng is AXA Chair Professor of Cybersecurity, Singapore Management University. His research interests are in the areas of data security and privacy, cloud security and IoT security. He has 26 patents and published more than 300 papers on cybersecurity. He served on many editorial boards and conference committees, including the editorial boards of IEEE Security & Privacy Magazine, IEEE Transactions on Dependable and Secure Computing, IEEE Transactions on Information Forensics and Security, Journal of Computer Science and Technology, and Steering Committee Chair of the ACM Asia Conference on Computer and Communications Security. He is an IEEE Fellow.

Jiayun Xu Jiayun Xu is currently a PhD candidate in the School of Information Systems at Singapore Management University (SMU). He received his master's and bachelor's degree at Shanghai Jiao Tong University. His research interests include Mobile Security and Malware detection on Android platform.

Ke Xu Ke Xu is currently a research scientist at Singapore Management University. She received the Ph.D degree from School of Information Systems, Singapore Management University in June 2018. Her research focuses on AI security, mobile security and application security.

Yingjiu Li Yingjiu Li is currently an Associate Professor in the School of Information Systems at Singapore Management University (SMU). His research interests include RFID Security and Privacy, Mobile and System Security, Applied Cryptography and Cloud Security, and Data Application Security and Privacy. He has published over 130 technical papers in international conferences and journals, and served in the program committees for over 80 international conferences and workshops. Yingjiu Li is a senior member of the ACM and a member of the IEEE Computer Society. The URL for his web page is http://www.mysmu.edu/faculty/yjli/