2019 IEEE European Symposium on Security and Privacy (EuroS&P)

DroidEvolver: Self-Evolving Android Malware Detection System

Ke Xu*, Yingjiu Li*, Robert H. Deng*, Kai Chen†‡ and Jiayun Xu*
*School of Information Systems, Singapore Management University
†SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences
‡School of Cyber Security, University of Chinese Academy of Sciences
{kexu, yjli, robertdeng, jyxu.2015}@smu.edu.sg, chenkai@iie.ac.cn

***Abstract*—Given the frequent changes in the Android framework and the continuous evolution of Android malware, it is challenging to detect malware over time in an effective and scalable manner. To address this challenge, we propose DroidEvolver, an Android malware detection system that can automatically and continually update itself during malware detection without any human involvement. While most existing malware detection systems can be updated by retraining on new applications with true labels, DroidEvolver requires neither retraining nor true labels to update itself, mainly due to the insight that DroidEvolver makes necessary and lightweight update using online learning techniques with evolving feature set and pseudo labels. The detection performance of DroidEvolver is evaluated on a dataset of 33,294 benign applications and 34,722 malicious applications developed over a period of six years. Using 6,286 applications dated in 2011 as the initial training set, DroidEvolver achieves high detection F-measure (95.27%), which only declines by 1.06% on average per year over the next five years for classifying 57,539 newly appeared applications. Note that such new applications could use new techniques and new APIs, which are not known to DroidEvolver when initialized with 2011 applications. Compared with the state-of-the-art overtime malware detection system MAMADROID, the F-measure of DroidEvolver is 2.19 times higher on average (10.21 times higher for the fifth year), and the efficiency of DroidEvolver is 28.58 times higher than MAMADROID during malware detection. DroidEvolver is also shown robust against typical code obfuscation techniques.**

I. Introduction

Both Android applications and Android framework evolve continually over time for various reasons such as capability enhancements and bug fixes [34]. As a result, it has been increasingly difficult to build Android malware detection systems that are trained with old Android applications and are effective and scalable in detecting malware from new applications after operating for a period of time. The rapid-aging of the existing Android detection systems raises an acute concern in both industry and academia. As it was reported in BlackHat 2016 [21], the recall rate of a malware detection system developed in Baidu drops by 7.6% in six months. In the research literature, a recent effort was made to make malware detection resilient to API changes through API abstraction [24]; however, the aging of malware detection has not been fully addressed.

To make malware detection accurate overtime, most malware detection systems need to be retrained timely and repetitively with new applications. Such solutions, however, face several challenges. First, it is difficult to determine when to retrain a malware detection system. If the system is retrained too frequently, it results in a waste of resources for retraining without providing novel information to enrich the detection system [19]; otherwise, the detection system cannot capture some new malware in a timely manner. Second, a retraining process requires manual labeling of all processed new applications, which is constrained by available resources [19]. The high cost of manual labeling usually induces a loose retraining frequency [31]; consequently, the detection performance is compromised. Lastly, most existing malware detection systems are retrained with cumulative datasets, including both original training dataset and newly labeled applications. Such retraining process is expensive and unscalable, especially in the scenario where the number of new applications grows rapidly over time.

Addressing these challenges, we propose a novel self-evolving Android malware detection system, named DroidEvolver, to make malware detection accurate over time by making necessary update to its detection models with evolving feature set. DroidEvolver maintains a model pool of different detection models that are initialized with a set of labeled applications using various online learning algorithms. The intuition of maintaining a model pool is that different detection models are less likely to be aging at same pace in malware detection even though they are initialized with the same dataset. In the detection phase, DroidEvolver makes a weighted voting among "young" detection models to classify each application based on its Android API calls. DroidEvolver extracts Android API calls as detection features because they naturally reflect the evolvement of both Android framework and applications and they can be easily extracted from bytecode for efficient malware detection.

A "young" model for a detected application is determined according to a *Juvenilization Indicator* (JI) that is calculated according to the similarity between the detected application and a batch of applications that have been classified by the detection model to the same prediction label. If a detection model is aging with respect to a detected application, DroidEvolver updates the model using the detected application and its classification result (i.e., pseudo label) generated by the model

DOI 10.1109/EuroSP.2019.00014

pool. DroidEvolver also updates its feature set to adapt to API changes discovered from the application.

DroidEvolver uses JI to determine when to update its feature set and each detection model. A JI can be calculated for each detection model and each new application being detected. If a JI falls out of a certain range, the corresponding detection model is deemed as *aging model*. A detected application is identified as *drifting application* if any model in the model pool is aging for detecting this application. An aging model has limitations of classifying a drifting application, which may include new API calls or new API usage patterns. DroidEvolver thus updates its feature set and all aging models once a drifting application is identified.

DroidEvolver requires no true labels of processed applications for updating its model pool in malware detection. This evades the necessity of manual labeling of any applications after the initialization of DroidEvolver, and thus reduces resource and cost constraints for the evolvement of DroidEvolver. In the case where a drifting application is identified, DroidEvolver generates a pseudo label for the drifting application, and updates all aging models according to the drifting application and its pseudo label before proceeding to the next application. In the case where the current application is not a drifting application (thus no aging model is identified), all models in the model pool contribute to the classification result, and no model update is performed.

DroidEvolver is efficient for malware detection over time. It does not require any retraining with cumulative datasets periodically after initialization; instead, it evolves efficiently whenever a single drifting application is identified and processed unless all models are aging in such case. To this end, all detection models in the model pool are initialized using online learning algorithms [3], which can perform incremental learning over streaming data. In contrast to batch learning algorithms, online learning algorithms are more efficient and scalable, as they avoid not only batch processing with the original training dataset in the initialization phase but also periodic retraining with cumulative datasets in the detection phase. While existing online learning algorithms work with labeled data only, DroidEvolver makes them work with applications that are associated with pseudo labels in the detection phase. Unlike existing online learning-based approaches which update detection models for each application with true label, DroidEvolver updates aging models only for each drifting application. In the update process, DroidEvolver does not require any true labels to be associated with the drifting applications; in this sense, DroidEvolver is more practical than the existing online learning-based approaches. Consequently, aging models are juvenilized promptly whenever it is necessary. This further contributes to the efficiency of DroidEvolver.

DroidEvolver is evaluated rigorously with a series of datasets, including 34,722 malicious applications and 33,294 benign applications dated from 2011 to 2016. The efficacy and efficiency of DroidEvolver are compared with MAMADROID, which is a state-of-the-art malware detection system [24] that is resilient to API changes over time. In the case where DroidEvolver and MAMADROID are trained and tested with same applications developed in the same time period, DroidEvolver significantly outperforms MAMADROID with 15.80% higher F-measure, 12.97% higher precision, and 17.57% higher recall on average in our experiments. When evaluated on testing sets that are newer than training sets by one to five years, the average F-measure of DroidEvolver is 92.32%, 89.30%, 87.17%, 87.46% and 89.97%, respectively. In comparison, the average F-measure of MAMADROID in the corresponding cases is 68.01%, 56.09%, 45.88%, 32.85% and 8.81%, respectively. The overall F-measure of DroidEvolver is 2.11 times higher than MAMADROID on average for malware detection over time. The F-measure of DroidEvolver declines by 1.06% on average per year over five years, while MAMADROID declines by 13.52% in the same case. In addition, DroidEvolver maintains its F-measure at a high level if it is updated by only a small amount of data with true labels, while MAMADROID's F-measure declines every year in such case.

We then evaluate the efficiency of DroidEvolver, and compare it with MAMADROID. The initialization of DroidEvolver requires linear time, which varies from 3s to 27s as the original training dataset increases from 10,000 to 50,000 applications, while MAMADROID takes non-linear time that varies from 26s and 1,207s. In the detection phase, DroidEvolver takes 1.37s on average to process an unknown application, while MAMADROID requires 39.15s on average in such case.

We also analyse the aging models and drifting applications identified during malware detection over time. DroidEvolver identifies 11.23% of new applications as drifting, while each detection model shows signs of aging for classifying about 30.13% of drifting applications on average. These percentages remain stable when evaluated with applications developed in later time periods. In addition, more than 50.00% of detection models are identified as aging for classifying 49.08% of drifting applications. These drifting applications are major sources of misclassifications, and the updates to aging models make DroidEvolver slow aging in malware detection.

The contributions of the paper are summarized as follows. We proposed a novel self-evolving and efficient Android malware detection system DroidEvolver. DroidEvolver is accurate in malware detection not only on the applications that developed in same time period as the training applications, but also on the new applications that are developed after the training applications with new techniques and new APIs. DroidEvolver is efficient since DroidEvolver leverages online learning algorithms to update its aging models from individual drifting applications during malware detection instead of retraining periodically from a collection of cumulative applications in a batch manner. Compared with the state-of-the-art malware detection system MAMADROID, DroidEvolver achieves significantly higher accuracy and higher efficiency in our experiments.

The rest of paper is organized as follows. Section II details the system design of DroidEvolver. Section III introduces the experimental setting and parameter tunning used in experi-

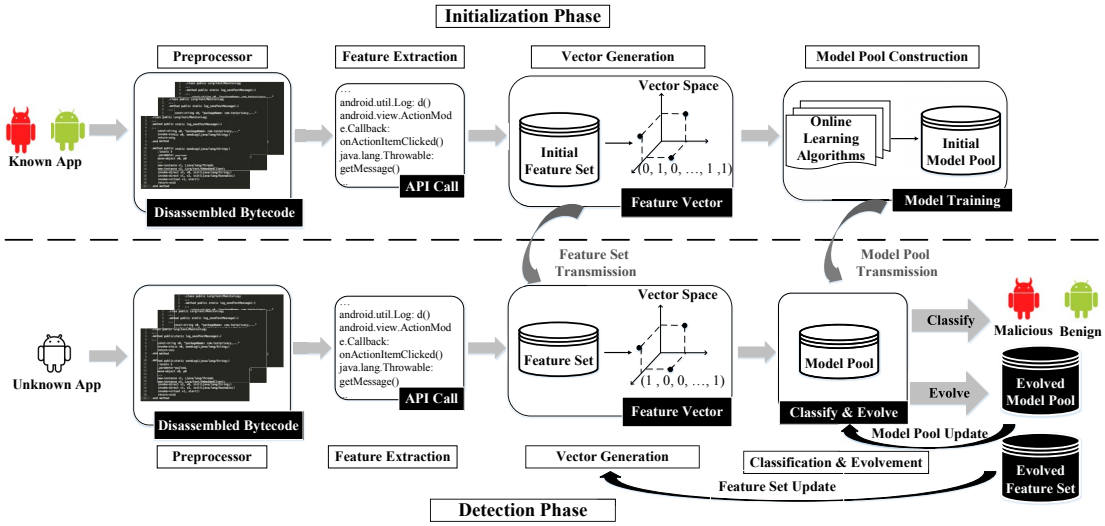


Fig. 1: Architecture of DroidEvolver

ments. Section IV evaluates DroidEvolver in different aspects, analyzes the experiment results and discusses its limitations. Section V summarizes the related work, and Section VI concludes the paper.

II. Design of DroidEvolver

The architecture of DroidEvolver[1] is shown in Fig. 1. DroidEvolver consists of two phases, including an *initialization phase* and a *detection phase*. In the initialization phase, DroidEvolver takes as input a set of known applications which are associated with true labels (i.e., "malicious" and "benign"), and outputs a set of features and a set of detection models, which are transmitted to the detection phase. In the detection phase, DroidEvolver takes as input each application which true label is unknown, and outputs a prediction label for the unknown application.

The initialization phase of DroidEvolver consists of four modules, including *preprocessor*, *feature extraction*, *vector generation*, and *model pool construction*. For each known application from the input, the preprocessor applies apktool [37] to decompile its apk file and obtain its disassembled dex bytecode, which includes API calls that are used in this application. Then, the feature extraction module is used to extract all Android APIs and record Android API binary presence for each application as the detection features of the application. An *initial feature set*, which is a total order set, is constructed by combining the detection features of all applications in the input. A feature space is constructed by a 1-to-1 mapping from all features in the initial feature set to the dimensions of the feature space. In the vector generation module, DroidEvolver generates a feature vector from each application for all detection models by mapping the detection features of the application into the feature space, where each detection feature that falls in the initial feature set is mapped to component one, while other components are set to zero.

Given the feature vectors generated from all applications in the input, the model pool construction module constructs an *initial model pool*, which consists of a set of detection models. Each detection model is initialized using a different online learning algorithm which processes all input applications according to their feature vectors and true labels. At the end of the initialization phase, DroidEvolver transmits the initial feature set and the initial model pool to the detection phase. Each detection model in the model pool is associated with a *feature set indicator*, which indicates the number of features that the detection model can process. All feature set indicators are initialized as the size of the initial feature set and may increase to larger values in the detection phase.

In the detection phase, DroidEvolver classifies each single unknown application as malicious or benign and performs necessary updates to the feature set and the detection models. The first three modules in the detection phase are similar to those in the initialization phase, except that (i) the feature set is dynamically updated to include new features, (ii) a feature space is constructed for each detection model by a 1-to-1 mapping from all features in the feature set whose ordinal numbers are less than the feature set indicator of the detection model, to the dimensions of the feature space, and (iii) DroidEvolver generates a feature vector from each application for each detection model by mapping the detection features of the application into the feature space of the detection model.

In the feature extraction module, DroidEvolver extracts Android APIs according to the existing Android API families [24], including `android`, `java`, `javax`, `junit`, `apach`, `json`, `dom`, and `xml`. While the number of API

[1]The code of DroidEvolver is released at http://github.com/DroidEvolver/DroidEvolver.

packages increases significantly from 96 at API level 1 (Android version 1.0 released in October 2008) to 196 at API level 27 (Android version 8.1 released in November 2017), the names of Android API families remain unchanged over time. In its detection phase, DroidEvolver would not miss any new Android API calls caused by Android framework evolvement as long as the API families of new API calls remain unchanged.

The last module in the detection phase is *classification and evolvement*. In this module, DroidEvolver generates a classification result (either malicious or benign) for each unknown application given in the input. If certain detection models in the model pool are aging in detecting an unknown application, DroidEvolver updates its feature set incrementally (without changing the ordinal number of any existing feature) by including all new Android API calls that are used in the unknown application, and updates the feature set indicator of each aging model to the size of the updated feature set. In addition, DroidEvolver updates each aging model by learning from the unknown application according to its classification result and updated feature vector.

The rest of this section clarifies how the model pool is constructed in the initialization phase, and how classification and evolvement are achieved in the detection phase.

A. Model Pool Construction

Given a set of known applications and their associated true labels in the initialization phase, DroidEvolver constructs a model pool with a set of online learning algorithms instead of any single detection model for malware detection. A single detection model may not always provide accurate detection results due to its limited capability [33]. The model pool can help detect and mitigate the bias of any single detection model and generate more reliable detection results in the detection phase.

Each detection model in the model pool is constructed using a different online learning algorithm that processes one application at a time. The complexity of online learning is linear to the number of applications in the input, which is different from batch learning that requires processing a set of applications at the same time. The common process of the online learning algorithms in DroidEvolver is given below.

Let the input of DroidEvolver be a total order set of N known applications. Let x_t be a d-dimensional real-valued feature vector for the t-th application in the input, where d is the size of the initial feature set derived from the input in the feature extraction module. Let y_t be the true label of the t-th application in the input, where $y_t = +1$ means "malicious" and $y_t = -1$ means "benign." The input of each online learning algorithm is a sequence of (x_t, y_t) for $t = 1, \ldots, N$. Each online learning algorithm uses a detection model, which is represented by a d-dimensional weight vector w_t, to process (x_t, y_t), where $1 \leq t \leq N$. The weight vector consists of the weights for all features in the initial feature set.

At each step t, each online learning algorithm processes x_t and generates a prediction label $\hat{y}_t = sgn(w_t \cdot x_t)$, where sgn is a function that maps any non-negative value to +1 and maps any negative value to -1. A loss value $l_t(y_t, \hat{y}_t)$ at step t is then computed from true label y_t and prediction label $\hat{y}_t$. Each online learning algorithm implements a different strategy on how to compute the loss value l_t and update its weight vector w_t to w_{t+1}.

In DroidEvolver, each detection model defines a hyperplane $\{x \in \mathbb{R}^d | w_t \cdot x = 0\}$, where w_t is the weight vector of the detection model that is generated by certain online learning algorithm. A detection model with weight vector w_t performs its classification on an application according to the distance from the feature vector x_t of the application to the hyperplane of the model: if the distance $w_t \cdot x_t$ is non-negative, the prediction label is "malicious"; otherwise, the prediction label is "benign." The absolute value of $w_t \cdot x_t$ is called as the *prediction score* of the model for x_t.

DroidEvolver constructs its model pool consisting of five linear online learning algorithms, including Passive Aggressive (PA) [9], Online Gradient Descent (OGD) [45], Adaptive Regularization of Weight Vectors (AROW) [10], Regularized Dual Averaging (RDA) [38], and Adaptive Forward-Backward Splitting (Ada-FOBOS) [11]. These algorithms are selected to cover major online learning algorithm categories, including first-order online learning (including PA and OGD), second-order online learning (including AROW), and online learning with regularization (including RDA and Ada-FOBOS) [16]. First-order online learning algorithms aim to optimize the objective functions using the first-order gradient information only. The advantage of the first-order algorithms is that their computational complexity is linear to the input size. Unlike the first-order online learning algorithms that only exploit the first-order derivative information of the gradient for the online optimization tasks, second-order online learning algorithms exploit both first-order and second-order information in order to accelerate the optimization convergence. However, second-order online learning algorithms often suffer from high computational complexity when dealing with high dimensional data. This challenge can be addressed by online learning algorithms with regularization, which aims to exploit the sparsity property of real-world high-dimensional data. In addition, the selected algorithms in each category are different in terms of update policy, learning rate, optimization method, and loss function, which enables various detection models to age differently in the model pool when DroidEvolver is applied in malware detection. The update procedures of the selected online learning algorithms are explained below.

Passive Aggressive (PA). PA [9] incrementally builds its detection model in multiple steps. At each step t, PA receives a sample x_t and predicts its label $\hat{y}_t$ using the current model w_t; it then receives the true label y_t of x_t and calculates the hinge loss $l_t = max\{0, 1 - y_t(w_t \cdot x_t)\}$. Finally, PA sets the learning rate at step t $\tau_t = \frac{l_t}{||x_t||^2}$ and updates $w_{t+1} = w_t + \tau_t y_t x_t$. In each step, the model update is passive if the hinge loss is zero (i.e., $w_{t+1} = w_t$ if $l_t = 0$). Otherwise, PA updates w_{t+1} aggressively. PA ensures that the updated w_{t+1} should stay as close as to w_t and every incoming sample should be classified

by the updated model correctly.

Online Gradient Descent (OGD). OGD [45] has similar update policy as PA, except that OGD employs predefined learning rate scheme while PA chooses the optimal learning rate at each step. At each step t, OGD receives a sample x_t and predicts its label $\hat{y}_t$ using the current model w_t; it then receives the true label y_t of x_t and calculates the hinge loss $l_t = max\{0, 1 - y_t(w_t \cdot x_t)\}$. After that, OGD updates $w_{t+1} = \prod_S(w_t - \eta_t \nabla l_t(w_t))$, where η_t is a predefined learning rate, S is a convex set initialized at $t = 0$, and $\prod_S$ is the projection function to constrain the updated model to lie in the feasible domain. To be specific, if $w_t - \eta_t \nabla l_t(w_t) \notin S$, $\prod_S$ projects $w_t - \eta_t \nabla l_t(w_t)$ to a vector which is the closest vector to $w_t - \eta_t \nabla l_t(w_t)$ within S. The projected vector is w_{t+1}.

Adaptive Regularization of Weight Vectors (AROW). AROW [10] takes the frequency of occurrence of different features into consideration for model update. In AROW, the frequent features receive more updates and are estimated more accurately compared to rare features.

AROW maintains a Gaussian distribution over weight vectors with mean μ and covariance Σ, i.e., $w \sim \mathcal{N}(\mu, \Sigma)$. AROW initializes $\mu_0 = 0$ and $\Sigma_0 =$ Identity Matrix. Given a sample x_t at each step t, AROW computes a margin $m_t = \mu_{t-1} \cdot x_t$ and a confidence $\nu_t = x_t{}^\intercal \Sigma_{t-1} x_t$. After receiving true lable y_t of x_t, AROW suffers loss $l_t = 1$ if $sgn(m_t) \neq y_t$. If $m_t y_t < 1$, AROW upates μ:

$$\mu_t = \mu_{t-1} + \frac{\max(0, 1 - y_t x_t{}^\intercal \mu_{t-1})}{x_t{}^\intercal \Sigma_{t-1} x_t + r} \Sigma_{t-1} y_t x_t$$

where r is an input parameter that is set by parameter tunning. If $\mu_t \neq \mu_{t-1}$, AROW updates the covariance Σ:

$$\Sigma_t = \Sigma_{t-1} - \frac{\Sigma_{t-1} x_t x_t{}^\intercal \Sigma_{t-1}}{x_t{}^\intercal \Sigma_{t-1} x_t + r}$$

After that, AROW outputs the updated mean μ_t and covariance Σ_t, which are then used to calculate the updated weight vector.

Regularized Dual Averaging (RDA). RDA [38] adjusts its parameters by solving a minimization problem for each sample. At each step t, RDA computes the subgradient $g_t \in \partial f_t(w_t)$, where f_t is the cost function revealed at step t. The subgradient g_t is used to compute the average subgradient. RDA then updates the average subgradient $\bar{g}_t = \frac{t-1}{t} \bar{g}_{t-1} + \frac{1}{t} g_t$. With the average subgradient, RDA updates the current weight vector by solving a minimization problem:

$$w_{t+1} = \arg\min_w \left\{ \bar{g}_t{}^\intercal w + \Psi(w) + \frac{\beta_t}{t} h(w) \right\}$$

where β_t is a non-negative sequence, $\Psi(w)$ is the original sparsity-inducing regularizer (i.e., $\Psi(w) = \lambda \| w \|_1$), $h(w)$ is an auxiliary strongly convex function (i.e., $h(w) = \frac{1}{2} \| w \|^2$), and $\bar{g}_t$ is the averaged subgradient of all previous iterations (i.e., $\bar{g} = \frac{1}{t} \sum_{\tau=1}^{t} \nabla l_\tau(w_\tau)$).

Adaptive Forward-Backward Splitting (Ada-FOBOS). One major challenge of RDA is that the geometry information of underlying data distribution may not be fully exploit by the auxiliary strongly convex function $h(w)$. To address this challenge, Ada-FOBOS [11] proposes a data-driven adaptive regularization for $h(w)$:

$$h_t(w) = \frac{1}{2} w^\intercal H_t w$$

where H_t is the diagonal matrix.

The update procedure is described as follows. At each step t, Ada-FOBOS receives x_t and predicts its label $\hat{y}_t$. It then suffers loss l_t and calculates gradient g_t with respect to w_t. Afterwards, it updates the diagonal matrix H_t of $h_t(w)$:

$$H_t = \delta + diag(\sum_{i=1}^{t} g_i g_i{}^\intercal)$$

where δ is the parameter that ensures positive-definite property of the adaptive weighting matrix. At last, Ada-FOBOS updates weight vector $w_{t+1} = w_t - \frac{\tau g_t}{H_t}$, where τ is the learning rate.

After all applications in the input have been processed, DroidEvolver outputs five detection models as its model pool, and associates each detection model with a feature set indicator, which is initialized as the size of the initial feature set. Then, DroidEvolver transmits the model pool to the detection phase for classification and evolvement.

B. Classification and Evolvement

In the detection phase, the classification and evolvement module performs classification for each unknown application and makes necessary updates to its feature set and model pool. This module consists of three steps, including *drifting application identification*, *classification and pseudo label generation*, and *aging model juvenilization*.

Drifting Application Identification - When to Evolve. Malware detection models that are trained with old applications usually produce unsatisfactory results when detecting new applications that are developed later than the training data. This phenomenon is known as *concept drift*. In order to make malware detection robust to concept drift, DroidEvolver identifies *drifting applications* which are different from old applications that have been processed before. For each drifting application, DroidEvolver identifies *aging models* which show signs of aging to classify the drifting application. The emerging of drifting applications signals the necessity of updating the corresponding aging models for DroidEvolver to maintain its effectiveness in malware detection.

DroidEvolver uses a *juvenilization indicator* (JI for short) to determine whether an unknown application is drifting or not when it is processed by a detection model. In a nutshell, JI indicates the similarity between a new application and a batch of applications measured by a detection model. To compute JI for a new application in an efficient manner, DroidEvolver uses an *app buffer* $B = (b_1, \ldots, b_K)$ of size K to store the feature vectors for a subset of the applications that have been processed, where K ($\leq N$) is a parameter in DroidEvolver, and b_t ($1 \leq t \leq K$) is a feature vector of certain application that have been processed before. At the beginning of the detection phase, the K applications in the app buffer are

randomly selected from the input given in the initialization phase. To keep the existing app buffer up-to-date, whenever a new application is taken as the input to the detection phase, DroidEvolver randomly replaces one feature vector from the app buffer with the feature vector of the new application for JI computation.

In particular, assume that DroidEvolver has completed its detection on the $(i-1)$-th unknown application A_{i-1} and received a new application A_i. Let x_i be the feature vector of A_i, M_j be the j-th detection model in current model pool, w_j be the weight vector of M_j, and B be the app buffer updated by A_i. Let σ be an indicator function, where $\sigma(true)$ equals 1 and $\sigma(false)$ equals 0. JI ξ_{ij} of model M_j for application A_i is defined as follows

$$\xi_{ij} = \begin{cases} \frac{\sum_{b_t \in B} \sigma(w_j \cdot x_i \geq w_j \cdot b_t)}{\sum_{b_t \in B} \sigma(w_j \cdot b_t \geq 0)} & \text{if } w_j \cdot x_i \geq 0 \\ \\ \frac{\sum_{b_t \in B} \sigma(w_j \cdot x_i < w_j \cdot b_t)}{\sum_{b_t \in B} \sigma(w_j \cdot b_t < 0)} & \text{else} \end{cases} \quad (1)$$

In the definition of ξ_{ij}, the prediction score of M_j for A_i is compared to the prediction scores of M_j for the applications in the app buffer that are classified by M_j to the prediction label of A_i. If the prediction label of A_i by M_j is malicious (benign, respectively), then the prediction score of M_j for A_i is in the $(100 \cdot \xi_{ij})$-th percentile ($(100 \cdot (1-\xi_{ij}))$-th percentile, respectively) among the prediction scores of M_j for the applications in the app buffer that are classified by M_j to "malicious" ("benign," respectively).

A_i is identified as a drifting application to detection model M_j if its feature vector x_i is too close or too far away from the hyperplane of M_j as compared to the other feature vectors included in the app buffer (i.e., an outlier from processed applications). DroidEvolver uses two thresholds, τ_0 and τ_1 $(0 \leq \tau_0 \leq \tau_1 \leq 1)$, to identify drifting applications according to JI values. If $\tau_0 \leq \xi_{ij} \leq \tau_1$, then ξ_{ij} is deemed valid; otherwise, ξ_{ij} is invalid, where τ_0 and τ_1 are chosen in the initialization phase and enforced in the detection phase. If ξ_{ij} is invalid, then M_j is identified as an *aging model* for detecting A_i. An unknown application is identified as a *drifting application* if any of the detection models in the model pool is an aging model for detecting it. DroidEvolver makes necessary updates to its feature set and the aging models in its model pool whenever a drifting application is identified.

Classification & Pseudo Label Generation - What to Evolve With. In the detection phase, DroidEvolver generates a classification result for each unknown application through a weighted voting. If an input application A_i is not a drifting application, the weighted voting is performed by all detection models in the model pool using $\sum_{j=1}^{M} w_j \cdot x_i$, where M is the size of the model pool, w_j is the weight vector of model M_j in the model pool, and x_i is the feature vector of A_i. If the voting result is non-negative, A_i is classified as "malicious," otherwise, its classification result is "benign."

In the case that the input application is identified as a drifting application, the weighted voting is performed among the detection models in the model pool excluding all aging models for detecting it. The classification result of the drifting application is then used as its *pseudo label* for updating the feature set and all aging models. In the other case that all models are aging or no model is aging, DroidEvolver performs the weighted voting among all models in the model pool and skips the updating process.

Aging Model Juvenilization - How to Evolve. The identification of a drifting application and a proper subset of aging models in the model pool indicates that the aging models should be juvenilized according to the drifting application for reliable malware detection over time for the following reasons. Firstly, the differences between drifting applications and other processed applications may be induced by new features or new patterns included in the drifting applications. It is thus important to adapt to these new features. In addition, the detection capabilities of aging models are constrained by the "aging" feature set and model structures, which needs to be updated to make accurate classification in the future. In such case, DroidEvolver first updates its current feature set by including all new features that are extracted from the drifting application. DroidEvolver then updates the feature set indicator and model structure of each aging model individually by learning from drifting application and corresponding pseudo label. Model structure includes the dimension of weight vector (according to current feature set indicator) and the values of the weight vector (according to the features included in the processed application).

Let $\bar{y}_i$ denote the pseudo label of a drifting application A_i, and W_j denote the weight vector of aging model M_j. DroidEvolver updates each aging model M_j for A_i in four steps. First, DroidEvolver uses M_j to compute a predication label $\hat{y}_{ij}$ for A_i. Then, it reveals the pseudo label $\bar{y}_i$ to M_j and computes a loss value $l_j(\bar{y}_i, \hat{y}_{ij})$ for M_j according to the loss function l_j of M_j's online learning algorithm. Third, it updates the feature set indicator of M_j to be the size of the updated feature set, and updates the feature vector of A_i according to the updated feature set and the updated feature set indicator. Lastly, it relies on the online learning algorithm of M_j to decide when and how to update M_j according to update policy.

The necessary updates of both feature set and aging models enable DroidEvolver to adapt to the changes in Android applications and Android framework. Unlike the existing online learning-based approaches, DroidEvolver relies on a model pool instead of a single detection model to generate its detection result. In addition, DroidEvolver requires no true labels of drifting applications for updating aging models.

III. Experimental Settings and Parameter Tuning

The performance of DroidEvolver is empirically evaluated in a series of experiments. The experimental settings and parameter tuning are detailed in this section.

A. Data Collection

TABLE I: Distribution of Datasets over Years

Year	2011	2012	2013	2014	2015	2016
Benign	4,414	5,789	5,784	5,793	5,750	5,764
Malicious	6,063	5,777	5,685	5,760	5,657	5,780
Total	10,477	11,566	11,469	11,553	11,407	11,544

We collected a set of applications in July 2017 from an open Android application collection project [2]. The labels of the applications in our dataset were determined by the reports which we obtained in August 2017 from VirusTotal [36], which is an anti-virus service with 63 anti-virus scanners. An application is labeled as benign if it received no alarm from all anti-virus scanners. On the other hand, an application is labeled as malicious if it received at least fifteen alarms from 63 scanners (i.e., about 24% of scanners raised alarm). The way we labeled our applications is consistent with previous research on malware detection. For example, Arp et al. [4] labeled an application as malicious if it received alarms from at least 20% of a set of anti-virus scanners. The resulting benign dataset consists of 33,294 applications and the malicious dataset includes 34,722 applications. The time of each application is decided by the time when the apk file of the application is packaged [28], which is included in the dex file of the apk file. The dates of the applications in the collected datasets cover from 2011 to 2016. As shown in Table I, each dataset consists of a nearly balanced number of benign applications (42.1%-50.5%) and malicious applications.

B. Metrics and Measurements

The performance of DroidEvolver is assessed using standard F-measure, i.e.:

$$\text{F-measure} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

where Precision = TP/(TP+FP) and Recall = TP/(TP+FN). TP denotes the number of malicious applications being detected correctly, FP denotes the number of benign applications being mistakenly detected as malicious and FN denotes the number of malicious application being mistakenly detected as benign. The performance of DroidEvolver is measured in two cases as described below.

Performance in Same Time Period. To avoid over-fitting, the parameters of DroidEvolver are selected such that DroidEvolver achieves its best performance on a validation set after it is initialized with a training set; the performance of DroidEvolver is then evaluated on a testing set, where the training set, the validation set, and the testing set are different sets of applications developed in same time periods without overlapping.

There are six time periods in our experiments, including 2011, 2011-2012, 2011-2013, 2011-2014, 2011-2015, and 2011-2016. For each time period, applications are randomly shuffled and divided into five equal-size subsets. We randomly choose three subsets as the training set, one subset as the validation set, and the remaining subset as the testing set.

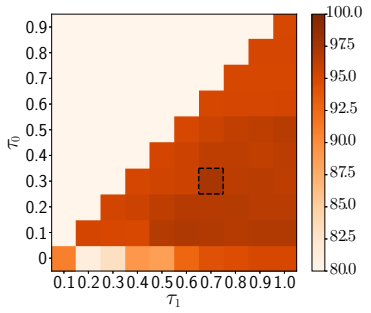


Fig. 2: Detection F-measure under Different Thresholds

Performance over Time. The performance of DroidEvolver is further evaluated over different time periods. The training set and the validation set are chosen the same way as in evaluating detection performance in same time period; the difference is that the testing set is replaced with a set of applications that are developed in later time periods.

C. Parameter Tuning

For each time period, parameters are selected such that DroidEvolver's performance is optimized with a validation set after being initialized with a training set. The selected parameters are then enforced to detect malware from a testing set (dated in same or later time periods). In the following, we show how parameter tuning is performed in an experimental setting which we refer to as the *default setting*. In this setting, 2011 dataset is used to form the training set (6,286 applications) and the validation set (2,095 applications).

Thresholds Tuning. Two thresholds, τ_0 and τ_1, are used to identify drifting applications and aging models according to JI values in the detection phase. Fig. 2 shows the effect of tuning thresholds to the performance of DroidEvolver in the default setting. Fig. 2 illustrates that the detection F-measure of DroidEvolver is stable when τ_0 changes from 0.1 to 0.3, and τ_1 increases from 0.6 to 0.8. The F-measure reaches its maximum value 96.33% when $\tau_0 = 0.3$ and $\tau_1 = 0.7$. Since the values of τ_0 and τ_1 are selected such that DroidEvolver achieves the best performance on a validation set after being initialized with a training set, DroidEvolver selects $\tau_0 = 0.3$ and $\tau_1 = 0.7$ in the default setting.

App Buffer Size Tuning. Recall that an app buffer is used in the classification and evolvement module for the calculation of JI in an efficient manner. The size K of the app buffer B has impact to the identification of drifting applications, and thus the detection F-measure of DroidEvolver and the time required to process all unknown applications. Fig. 3 shows the F-measure of DroidEvolver and the processing time of 2,095 applications in the classification and evolvement module under different app buffer sizes, where the threshold values are set to $\tau_0 = 0.3$ and $\tau_1 = 0.7$. The F-measure of DroidEvolver varies

TABLE II: Performance of DroidEvolver and MAMADROID for Detection in Same Time Period

	[F-measure, Precision, Recall]%								
Year (App #)	2011 (10,477)			2011-2012 (22,043)			2011-2013 (33,512)		
DroidEvolver	95.27	97.03	93.58	96.75	97.36	96.70	96.39	97.36	95.44
MAMADROID	76.43	91.99	65.38	81.79	89.30	75.45	81.99	84.51	79.62
Year (App #)	2011-2014 (45,065)			2011-2015 (56,472)			2011-2016 (68,016)		
DroidEvolver	96.15	97.60	94.74	96.02	95.14	96.91	96.34	98.15	94.60
MAMADROID	80.97	82.62	79.38	81.29	80.68	81.91	79.68	75.15	84.80

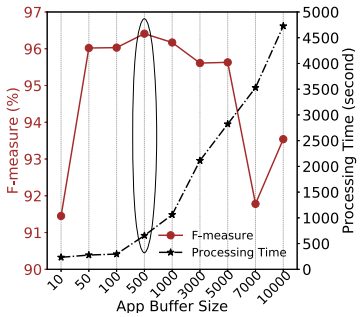


Fig. 3: App Buffer Size Tuning

from 91.45% to 96.41% when the app buffer size K increases from 10 to 10,000, indicating that the detection F-measure is not too sensitive to the change of K. On the other hand, the processing time of all applications increases significantly from 200s to 4,759s as K changes from 10 to 10,000. To balance F-measure and efficiency, DroidEvolver selects $K = 500$ in the default setting.

IV. Evaluation and Analysis

In this section, the performance of DroidEvolver is evaluated and analyzed in a series of experiments using the datasets summarized in Table I. The performance of DroidEvolver is compared with MAMADROID [24], which is a state-of-the-art malware detection system that is resilient to the changes of Android APIs over time. MAMADROID first uses Soot [35] and FlowDroid [5] to extract an API call graph from each application; it then extracts a set of API call sequences from the API call graph, and abstracts each API call to its package in package mode or to its family in family mode. From the abstracted API call sequences, MAMADROID builds a Markov Chain by evaluating the probabilities of all transitions between abstracted API calls. It then derives a feature vector from each application according to its Markov Chain, where each non-zero feature vector component is the corresponding probability in the Markov Chain. MAMADROID uses a batch learning algorithm to build a detection model and detects malware from unknown applications.

For performance comparison, we re-implemented MAMADROID using its source code [23] and operated it in its package mode (which consistently outperforms family mode [24]), where the batch learning classification algorithm is Random Forest (which achieves the best performance in [24]). We evaluated both DroidEvolver and MAMADROID in same experimental settings.

A. Detection in Same Time Period

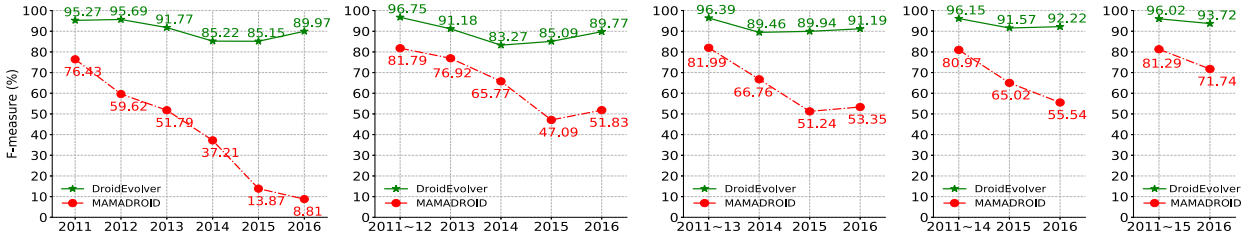
Table II compares the performance of DroidEvolver and MAMADROID in terms of F-measure, precision and recall. In each experiment, DroidEvolver and MAMADROID are initialized/trained with same training set and evaluated with same testing set. In all experiments, DroidEvolver outperforms MAMADROID consistently and significantly, achieving 15.80% higher F-measure, 12.97% higher precision, and 17.57% higher recall on average. By learning from drifting applications and adapting to new changes, DroidEvolver achieves 96.15% F-measure on average using Android APIs as detection features.

B. Detection over Time

Our next and main focus is on the detection performance of DroidEvolver and MAMADROID over time where their detection models are trained with a set of applications developed in one time period and then tested with another set of applications developed in later time periods.

As shown in Fig. 4, the overtime performance of DroidEvolver gains significant margins over MAMADROID in all experiments. When DroidEvolver is evaluated on testing sets that are newer than training sets by one to five years, the average F-measure of DroidEvolver is 92.32%, 89.30%, 87.17%, 87.46% and 89.97%, respectively. In comparison, the average F-measure of MAMADROID in the corresponding cases is 68.01%, 56.09%, 45.88%, 32.85% and 8.81%, respectively. The overall F-measure of DroidEvolver is 2.11 times higher than MAMADROID on average for malware detection over time. The F-measure of DroidEvolver declines by 1.06% on average per year over five years, while MAMADROID declines by 13.52% in the same case.

Fig. 4 also shows that, after operating for two to three years, the F-measure of DroidEvolver becomes stable. In its detection phase, DroidEvolver automatically learns from drifting applications with pseudo labels and updates its feature set and model pool. In comparison, the detection model in MAMADROID makes no evolvement after training. The



(a) (b) (c) (d) (e)

Fig. 4: Detection Performances of DroidEvolver and MAMADROID. (a), (b), (c), (d), and (e) show results of initializing/training with training set of 2011, 2011∼2012, 2011∼2013, 2011∼2014, and 2011∼2015, respectively.

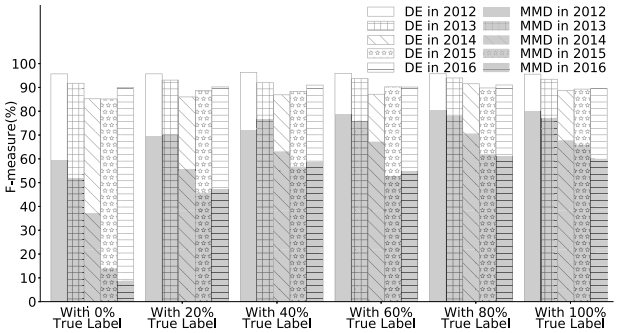


Fig. 5: Detection Performance of DroidEvolver(DE) and MAMADROID(MMD) When Updated with Different Percentage of Drifting Applications with True Labels, where Drifting Applications Take about 11.23% of All Testing Applications on Average.

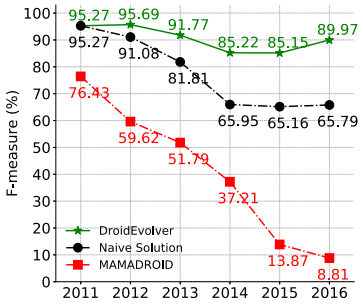


Fig. 6: Detection Performance of DroidEvolver, Naive Solution and MAMADROID in Default Setting

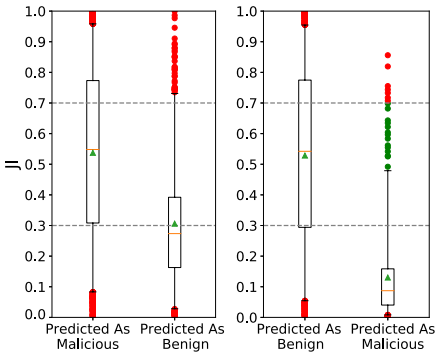
advantage of DroidEvolver is more obvious in comparison with MAMADROID in the default setting in which the training set is relatively old and small as shown in Fig. 4a.

C. Impact of Model Updates

DroidEvolver performs necessary model updates in detection phase by learning from drifting applications without true labels. Nonetheless, if true labels are available to certain applications, DroidEvolver can perform lightweight model updates on such applications. After initializing with 2011 training set, DroidEvolver and MAMADROID are further tested by updating their models with a fixed percentage of randomly selected drifting applications with true labels during malware detection, where drifting applications take about 11.23% of all testing applications on average. As shown in Fig. 5, DroidEvolver significantly and consistently outperforms MAMADROID in all settings. The overall F-measure of DroidEvolver is 1.44 times higher than MAMADROID on average for malware detection over time. In addition, DroidEvolver maintains its average F-measure above 92% if it is updated by 60% (or above) of drifting applications with true labels. Since around 11.23% of testing applications in each year are identified as drifting applications, DroidEvolver can maintain its detection performance at a high level if it is updated by only about 6.74% of testing applications with true labels in each year. Learning from a small amount of applications with true label helps DroidEvolver update itself with higher quality and achieve better detection performance.

As shown in the white bars in Fig. 5, the average F-measure of DroidEvolver slightly varies from 90.51% to 92.89% when the percentage of drifting applications with true labels increases from 0% to 100%. In addition, the F-measure remains stable when the percentage increases from 40% to 100%. In contrast, the average F-measure of MAMADROID increases significantly from 41.29% to 71.25% as the percentage changes from 0% to 100% (as shown in the grey bars in Fig. 5). Compared with MAMADROID, DroidEvolver is less sensitive to model updates with true labels and achieves much better performance. One reason is that DroidEvolver is able to learn from pseudo labels of drifting applications when true labels are not available. Another reason is that the pseudo label of each drifting application is generated through a weighted voting by excluding the aging models from the model pool, which can help mitigate the bias of any single detection model and generate more reliable pseudo labels.

When the true labels of certain applications are available for model updates, DroidEvolver outperforms MAMADROID not only in terms of efficacy, but also in efficiency. While DroidEvolver can perform instant model updates by individual



(a) True Malicious (b) True Benign

Fig. 7: Distribution of JI Values for One Detection Model

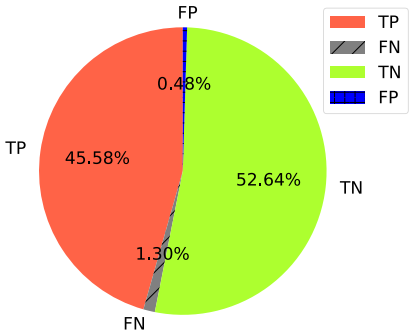


Fig. 8: Distribution of TPs, FNs, TNs and FPs in Drifting Applications Developed in 2012

testing applications with true labels, MAMADROID requires repetitively retraining by including all available testing applications with true labels into the training set.

The importance of updating aging models in DroidEvolver by drifting applications can be further justified by testing a *naive solution* in which no drifting application is identified and no update is performed; instead, the classification result of each application is generated by all detection models through weighted voting. Fig. 6 compares the performance of DroidEvolver with the naive solution in the default setting. It shows that the F-measure of the naive solution is significantly lower than DroidEvolver after operating for several years. The naive solution is less effective in malware detection overtime due to more and more detection models are aging during malware detection as no update is performed.

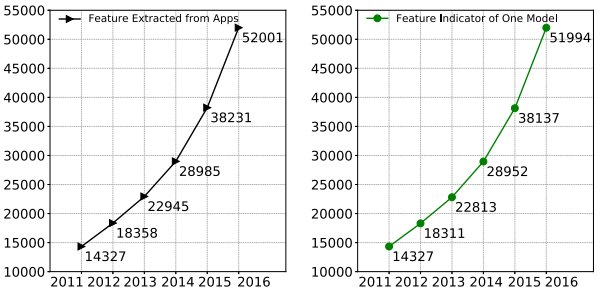
Although the naive solution is less effective than DroidEvolver, it still outperforms MAMADROID considerably. Its average F-measure is 1.88 times higher than MAMADROID in same experimental setting. The reason is that the weighted voting performed in the naive solution helps mitigate the bias of single detection model and thus generate more reliable detection results even without model updates.

D. Impact of Identifying Drifting Applications and Aging Models

The distribution of JI values is analyzed to show the importance of identifying drifting applications and aging models during detection. Our analysis is performed in the default setting, where DroidEvolver is evaluated with 2012 applications with fine-tuned parameters $\tau_0 = 0.3$, $\tau_1 = 0.7$, and $K = 500$. The distribution of JI values is shown for one model only (i.e., Passive Agressive) since the analysis to other models leads to similar results. The chosen detection model shows signs of aging to classify 3.55% of new applications developed in 2012. This percentage remains stable (without showing more signs of aging) when evaluated with applications developed in later time periods, which indicates the update performed by DroidEvolver make the detection model slow-aging.

Fig. 7 shows box and whisker plots for the distribution of JI values. The boxes extend from the lower quartiles to the upper quartiles of JI values, where the means and medians are marked with solid triangles and lines, respectively. The whiskers extend from the boxes to show the $5th$ and the $95th$ percentiles of JI values. The grey baselines mark τ_0 and τ_1, and each red filter point denotes the invalid JI value of a drifting application. Fig. 7a and Fig. 7b show JI distributions for true malicious applications and true benign applications, respectively. Most correct predictions (i.e., first columns in these figures) are associated with valid JI values, while a majority of false predictions are associated with invalid JI values. In most cases, if an application is not drifting, the detection model can make a correct prediction; otherwise, the detection model is aging and may misclassify the drifting application. For this reason, DroidEvolver generates a classification result for each drifting application by excluding all aging models from weighted voting; it then updates all aging models using online learning algorithms according to each drifting application and its pseudo label. In the extreme case where all detection models are aging in classifying a drifting application, the weighted voting is performed by all models, and no update is performed on any model.

We then analyse the identified drifting applications to demonstrate the impact of drifting applications. The experiment is performed in the default setting, where DroidEvolver identifies 1459, 1308, 1242, 1267, and 1183 drifting applications from testing applications dated in 2012, 2013, 2014, 2015 and 2016, respectively. Since all drifting applications and corresponding pseudo labels are then used to update identified aging models, the correctness of pseudo labels is important to the detection performance of DroidEvolver. Drifting applications with incorrect pseudo labels might mistakenly update aging models. Fig. 8 shows the distribution of TPs, FNs, TNs and FPs in drifting applications dated in 2012. For all identified drifting applications, DroidEvolver correctly classifies 98.22% of them into benign or malicious. The rest 1.78% of drifting applications with incorrect pseudo labels may mislead DroidEvolver to generate incorrect classification



(a) Num of Extracted Features (b) Feature Set Indicator

Fig. 9: Feature Evolvement

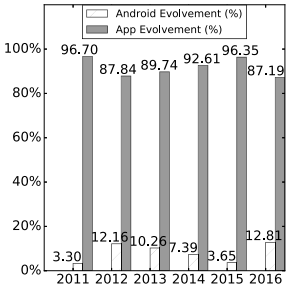


Fig. 10: Distribution of New Features

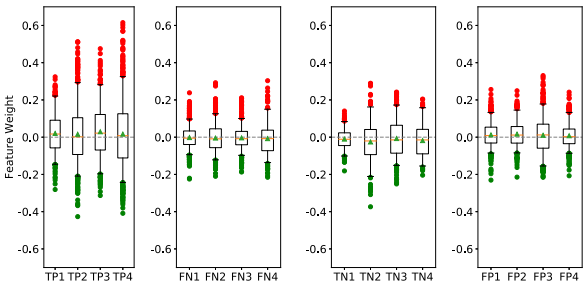
results and lead to the decline of F-measure for malware detection overtime.

E. Feature Evolvement

A major advantage of DroidEvolver is that its feature set is not fixed. It grows by taking in new features that are extracted from drifting applications during detection. This advantage helps DroidEvolver adapt to the new features that are introduced due to application evolvement and Android framework evolvement.

Fig. 9 shows feature evolvement in the default setting where DroidEvolver is initialized with 2011 training set and tested with applications developed from 2012 to 2016. In particular, Fig. 9a shows that the real number of extracted features increases from 14,327 to 52,001 in six years. A similar growth pattern is observed for the feature set indicator of each detection model in the model pool; for example, Fig. 9b shows the growth of the feature set indicator for one detection model (i.e., Passive Aggressive). The similar growth pattern indicates that (i) DroidEvolver can update its feature set to include new features discovered from drifting applications, and (ii) detection models of DroidEvolver are updated to adapt to new features for malware detection over time.

The new features that are added to the feature set in each year can be considered as coming from two sources, including application evolvement and Android framework evolvement. The application evolvement means that the new features are the existing APIs that have not been used in any applications



(a) TPs (b) FNs (c) TNs (d) FPs

Fig. 11: Feature Weight Distribution for TPs, FNs, TNs, and FPs

in the training set or any drifting applications that have been processed before. The Android framework evolvement means that the new features are the new APIs that are added into Android specifications. Fig. 10 shows that a majority of the new features comes from application evolvement, while the contribution from Android framework evolvement cannot be ignored. Note that the distribution of new features may change in different experimental settings. Nonetheless, a slow-aging malware deteciton system should be designed to accommodate new features and makes a good use of them in malware detection over time.

F. False Positives and False Negatives

To explain why false positives and false negatives are misclassified, DroidEvolver outputs a weight value for each feature of an application being detected. The weight value of a feature indicates how significant the feature contributes to the classification result. In particular, the weight of a feature for an application is computed as $(\sum_{j \in S} w_j \cdot x)$, where S is the set of models in the model pool that participate in the weighted voting, w_j is the weight vector of each model in S at the time when the weighted voting is performed for the application, and x is one-hot vector for the feature. Note that the sum of all features' weights for an application is exactly the result of the weighted voting, which is used to derive the classification result for the application. For a malicious (benign, respectively) application, a non-negative (negative, respectively) weight shows that the corresponding feature contributes positively to its detection, and the absolute value of the weight indicates how significant is the contribution.

We then present the detailed analysis of false positives and false negatives, where DroidEvolver is evaluated on 11,566 applications dated in 2012 (including 5,789 benign applications and 5,777 malicious applications). In such case, DroidEvolver achieves F-measure 95.69%, accuracy 95.91%, TPR 93.39%, and FPR 1.70%.

Feature Weight Distribution. We first analyse the feature weight distribution for true positives (TPs), false negatives (FNs), true negatives (TNs), and false positives (FPs). Fig. 11 shows box and whisker plots for 16 randomly chosen ap-

TABLE III: Distribution of Non-Negative Weight Features (NNWF) for FPs

(a) NNWF% among All Features

NNWF (%)	(40,50)	(50,60)	(60,70)	(70,80)
FPs (%)	9.09%	90.91%	0%	0%

(b) NNWF% among Top 100 Significant Features

NNWF (%)	(40,50)	(50,60)	(60,70)	(70,80)
FPs (%)	0%	36.36%	45.46%	18.18%

TABLE IV: Distribution of Negative Weight Features (NWF) for FNs

(a) NWF% among All Features

NWF (%)	(40,50)	(50,60)	(60,70)	(70,80)
FNs (%)	6.67%	81.67%	11.66%	0%

(b) NWF% among Top 100 Significant Features

NWF (%)	(40,50)	(50,60)	(60,70)	(70,80)
FNs (%)	15.00%	48.33%	35.00%	1.67%

plications (including four TPs, four FNs, four TNs and four FPs), where each box extends from the lower quartile to the upper quartile of the feature weights of certain application, the whiskers extend from the boxes to show the $5th$ and the $95th$ percentiles of the feature weights, the mean (median, respectively) of the feature values is marked with a solid triangle (line, respectively) inside the box, a grey baseline insides the box marks value zero, and each red filter point (green filter point, respectively) denotes a non-negative weight feature (a negative weight feature, respectively).

Fig. 11 shows that the feature weight distribution for FPs (see Fig. 11d) is more similar to that for TPs (see Fig. 11a) than that for TNs (see Fig. 11c) as most feature weights are non-negative. On the other hand, the distribution of feature weights for FNs (see Fig. 11b) is more similar to that for TNs (see Fig. 11c) than to that for TPs (see Fig. 11a) in a sense that most feature weights fall below zero.

False Positives. From 5,789 benign application, DroidEvolver produces 98 FPs (i.e., FPR = 1.70%). Table III shows the distribution of non-negative weight features for these FPs. As shown in Table IIIa, all of these FPs possess more than 40% of non-negative weight features among all features extracted from their apk files. Although their true labels are benign, 90.91% of them have more non-negative weight features than negative weight features. Among the top 100 significant features, Table IIIb further shows that all of these FPs have more non-negative weight features than negative weight features. These explain why these application are predicted wrongly by DroidEvolver. Among the top 100 significant features extracted from these FPs, we find some common non-negative weight features, such as *android.telephony.TelephonyManager:getDeviceID*, *android.telephony.TelephonyManager:getSimSerialNumber*, and *android.content.pm.PackageManager:getApplicationInfo*. These APIs are often used by malware to get personal information about users' devices and check installed applications.

False Negatives. DroidEvolver generates 382 FNs from 5,777 malicious applications in its detection phase (i.e., FNR = 6.61%). Table IV shows the distribution of negative weight features for these FNs. As shown in Table IVa, 93.33% of these FNs have more negative weight features than non-negative weight ones. Among the top 100 significant features, Table IVb shows that 85% these FNs contains more negative weight features than non-negative weight features. Since a majority of features are associated with negative weights, it is difficult for DroidEvolver to rectify its detection on these FNs unless more features than API calls are examined.

G. Runtime Performance

The runtime performance of DroidEvolver is evaluated in its detection phase and compared to MAMADROID on a machine with 4×3.2 GHZ Intel-Cores and 12 GB of RAM. For runtime performance evaluation, both DroidEvolver and MAMADROID are initialized/trained on 2011 training set, and tested on 2012 dataset.

Table V summarizes the runtime performance of DroidEvolver and MAMADROID in the average case. As shown in Table Va, the performance bottleneck of DroidEvolver is preprocessor, which decompiles apk files to obtain bytecode. DroidEvolver takes only about 0.05s on average to perform classification and evolvement, including drifting application identification, application classification, and necessary aging model juvenilization. By using API calls that can be easily extracted as detection features and applying online learning algorithms to quickly update detection models, DroidEvolver takes 1.37s on average in total to process each unknown application in its detection phase.

Table Vb shows the runtime performance of MAMADROID in the same experimental setting. The first step of MAMADROID is "call sequence abstraction," which extracts API call graphs using Soot and FlowDroid and abstracts the extracted API calls to their packages. This step takes 37.29s on average for each application, which is the bottleneck of MAMADROID. The second step is to build a Markov chain model and construct a feature vector, which takes about 0.43s on average. Finally, MAMADROID takes about 0.0036s on average to classify a feature vector to either benign or malicious. In total, MAMAROID requires 39.15s on average to process each unknown application in its detection phase, which is 28.58 times slower than DroidEvolver.

As shown in Table V, DroidEvolver is significantly faster than MAMADROID in all modules except in classification and evolvement. Although classifying the feature vector of unknown application and updating aging model (i.e., 3.22×10^{-3}s on average) are lightweight, it is dominant to identify drifting applications and corresponding aging models. Although DroidEvolver leverages app buffer to strike a balance between effectiveness and efficiency, it still takes more time than other steps to calculate JI value for each unknown

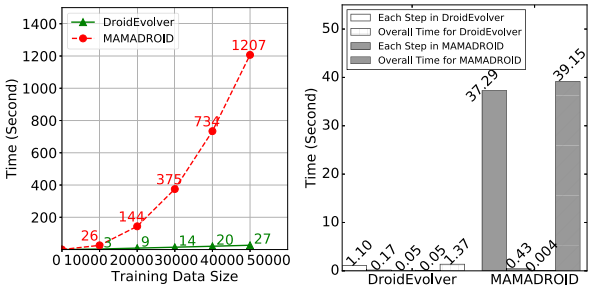
TABLE V: Average Time of Processing An Unknown Application in Detection Phase

(a) DroidEvolver

DroidEvolver	Preprocessor	Feature Extraction	Vector Generation	Classification& Evolvement	Overall
	1.1s	0.17s	0.05s	0.05s	1.37s

(b) MAMADROID

MAMADROID	Call Sequence Abstraction	Feature Vector Extraction	Classification	Overall
	37.29s	0.43s	0.0036s	39.15s



(a) Initialization Phase (b) Detection Phase

Fig. 12: Runtime Evaluation

application, which requires comparison with all applications included in the app buffer.

The runtime performance of DroidEvolver and MAMADROID is then evaluated separately in their initialization/training phase and detection phase in Fig. 12. As shown in Fig. 12a, the time required by the initialization phase of DroidEvolver increases from 3s to 27s as the size of training set increases from 10,000 to 50,000. In the same setting, the training phase of MAMADROID increases from 26s to 1,207s, which ranges from 8.67 to 44.70 times slower than DroidEvolver.

In their detection phases, both DroidEvolver and MAMADROID are linear in runtime performance to the number of applications that are processed by them. Fig. 12b shows the average performances of them for processing each application in their detection phases, indicating that MAMADROID is significantly slower than DroidEvolver.

A high efficiency of DroidEvolver is achieved in several aspects. First, DroidEvolver leverages online learning algorithms [17] to update its aging models from individual drifting applications instead of learning from a collection of applications as in all batch learning-based malware detection systems. This update process makes DroidEvolver efficient to process a stream of applications sequentially. DroidEvolver does not require any retraining with cumulative datasets periodically to keep up with the trends in application evolvement and Android framework evolvement. In addition, DroidEvolver requires no true labels to be available for updating its model pool in the detection phase. In comparison, most existing malware detection systems depend on periodic model retraining to update their detection models, and such model retraining requires manual labeling of a set of new applications beyond the original training set, which is constrained by available resources. Besides, DroidEvolver applies extracted Android API calls as detection features, which can be easily retrieved from decompiled bytecode without complicated process, such as those performed in Soot [35], FlowDroid [5], and TaintDroid [12]. Last but not least, DroidEvolver applies app buffer to speed up the process of JI calculation without declining detection performance.

H. Robustness

DroidEvolver is robust against common code obfuscations, including identifier renaming, junk code insertion, code reordering, and data encryption. Such code obfuscations may evade many existing commercial anti-malware tools [30].

DroidEvolver is resilient to identifier renaming type of obfuscations such as resigning, repackaging, and changing class/field/method names because DroidEvolver does not rely on specific application signatures or class/field/method names to detect malware. DroidEvolver is also resilient to junk code insertion, which inserts junk code segments into application source code. If the inserted junk code segments include no Android API calls, such junk code segments will not be extracted by DroidEvolver and thus have no impact to DroidEvolver's performance. In addition, DroidEvolver is robust to code reordering type of obfuscations, which change the control-flow logics of obfuscated applications. DroidEvolver is robust because it exploits no control-flow logic for malware detection. DroidEvolver is also robust to data encryption type of code obfuscations since they encrypt strings and/or arrays without modifying original API calls in application source code.

We test the robustness of DroidEvolver by applying DroidChameleon [30] to obfuscate 100 malicious applications that are randomly selected from the malicious applications in 2012 dataset. DroidChameleon is a framework of eleven typical obfuscation techniques, including (1) Identifier renaming: Disassembling and Reassembling, Class Renaming, Method Renaming, and Field Renaming; (2) Junk code insertion: Junk Code Insertion, and Nop Instruction; (3) Code reordering: Code Reordering, Order Reversing, and Function Indirection

Insertion; and (4) Data encryption: String Encryption, and Array Encryption. We apply each of the 11 obfuscation techniques to the 100 selected malicious applications, generating 1,100 obfuscated applications. After initializing on 2011 dataset, DroidEvolver can successfully detect 96% of the obfuscated applications. For each of the 11 obfuscation techniques, DroidEvolver missed four out of 100 obfuscated applications. After manually checked these 44 missed applications, we found that all of them were obfuscated from the same four malicious applications. We further tested DroidEvolver on these four malicious applications and found that DroidEvolver missed them too. Therefore, DroidEvolver missed these 44 obfuscated applications not because they are obfuscated but because they are missed even without obfuscations.

A strong attacker may carefully craft malware in adversarial sampling [15] so as to evade malware detection and mislead detection models. One example of adversarial sample crafting is to introduce selected perturbations to the feature vectors of certain malicious applications so that the detection results on the perturbed feature vectors are benign [29], [14]. Consider a strong attacker who knows DroidEvolver mechanisms and status, including all detection models in the model pool, and all feature vectors in the app buffer, at any time. Such attacker may craft a feature vector at certain time to evade malware detection, or create a set of feature vectors to mislead the evolvement of DroidEvolver. However, it remains challenging for such attacker to build a real-world malicious application from a crafted feature vector since it is non-trivial to achieve certain purpose-driven malicious function from a fixed list of Android APIs that is defined by a crafted feature vector. It remains a future research direction on how such attacks can be performed in reality in a large scale.

I. Limitations and Extensions

DroidEvolver is a static analysis system that detects malware according to a set of Android API calls included in its bytecode. It cannot detect malware that can only be detected based on more complicated features such as API call graphs and bytecode semantics. DroidEvolver can be evaded if the API calls of malware are not visible in static analysis, such as dynamically loaded malicious code and runtime malicious behaviors. Another limitation of DroidEvolver that is inherited from online learning is that it is vulnerable to poisoning attacks [6][20]. In poisoning attacks, attackers may deliberately craft the initialization dataset for DroidEvolver such that it is not be able to detect certain malware effectively.

To address these limitations, DroidEvolver can be extended to work with any other detection features instead of a set of Android API calls. Such extension is feasible because DroidEvolver's core modules, including model pool construction and classification and evolvement, are independent of how feature vectors are generated in other modules. By plugging in more complicated features such as API call graphs and control flow graphs in static analysis, DroidEvolver may achieve higher accuracies with certain tradeoffs on its performance overheads. DroidEvolver can also be extended to native code analysis and dynamic analysis as long as appropriate features can be generated and used for such analyses. While our current work focuses on designing and evaluating the evolving mechanisms in DroidEvolver with lightweight detection features, our future work will shift to extending DroidEvolver to other detection features.

To thwart poisoning attacks, we suggest that DroidEvolver be extended to include a sanitization module in its initialization phase. For each application in the initialization dataset, the sanitization module requires each detection model in the model pool to generate a classification label based on the current model structure. If a majority of the detection models in the model pool cannot reach an agreement on the classification label of an application, the sanitization module may consider the application "poisoned" and delete it from the initialization dataset. This sanitization process can be performed in several rounds (each time from scratch) with randomly reshuffled initialization dataset. After the initialization dataset is sanitized, DroidEvolver can be initialized for malware detection.

V. Related Work

Over the past few years, Android malware detection has attracted extensive attentions in both academia and industry. DroidEvolver is more related to learning-based Android malware detection systems which we review below.

Detection Over Time. MAMADROID [24] is an Android malware detection system that is resilient to Android API changes due to abstract API calls to their packages and families. The detection model in MAMADROID is not automatically updated by individual applications in the detection phase since it is built from a batch learning algorithm. To maintain its effectiveness for malware detection over time, MAMADROID may be retrained with a new set of applications; however, such retraining is constrained on the availability of true labels for the applications used in retraining. In comparison, DroidEvolver requires no true labels for model evolvement in its detection phase.

Another related work is Transcend [19], which is a framework to detect concept drift in malware classification models. The importance of identifying drifting applications and aging models in DroidEvolver is motivated by Transcend; however, the statistical metrics proposed in Transcend for detecting concept drift cannot be directly applied in DroidEvolver because Transcend calculates its concept drift metrics for each testing application by comparing it to all and only training applications with true labels. The concept drift metrics do not capture how each testing application is different from any new applications that are given in the detection phase without true labels. Another difference between Transcend and DroidEvolver is that Transcend focuses on how to detect concept drift only without covering how to develop a slow-aging, scalable, and robust malware detection system for effective malware detection over time.

Online Learning in Malware Detection. Online learning algorithms have been applied to Android malware detection

in recent years. For example, DroidOL [27] and CASANDRA [26] use online learning algorithms to build malware detection models and classify Android applications according to their API call graphs. Both DroidOL and CASANDRA automatically retrain their detection models using online learning algorithms upon receiving each labeled application, and classify unlabeled applications using the updated detection models. The retraining process requires that each application be associated with its true label, which is constrained by available resources. In comparison, DroidEvolver requires no true labels for automatically updating its detection models in its detection phase, which evades the necessity of labeling any applications after the initialization phase. Unlike these existing approaches, DroidEvolver relies a model pool instead of any single online learning algorithm to generate its detection results. The model pool can help detect and mitigate the bias of any single detection model and thus generate more reliable detection results.

Other Malware Detection Methods. Different from DroidEvolver, most learning-based malware detection systems rely on frequent retraining to maintain their effectiveness in malware detection over time, while the retraining is performed on a set of labeled applications including new applications beyond the previous training set. An incomplete list of such malware detection systems is given below. 6thSense [32] utilizes three different machine learning techniques (i.e., Markov Chain, Naive Bayes, and LMT) to detect malicious behaviors associated with mobile phone sensors. StormDroid [8] uses various machine learning algorithms, such as SVM, Decision Trees, and Naive Bayes to detect malware according to used permissions, sensitive API calls, and sensitive API call sequences. MARVIN [22] applies the Linear Logistic Regression model to classify Android applications from a large number of features derived in both static analysis and dynamic analysis. DroidMiner [40] also uses various machine learning algorithms, including SVM, Decision Tree, Naive Bayes, and Random Forest to detect malware from sensitive API call graphs. Drebin [4] applies SVM to malware detection based on request permissions, app components, and suspicious API calls. Finally, DroidSIFT [43] performs both anomaly based detection and signature based detection based on feature vectors generated from contextual API dependency graphs.

Recent effort on malware detection has been made on how to use deep neural networks for better malware detection. For example, DroidDetector [42] and Droid-Sec [41] build Deep Belief Networks for Android malware detection using 192 human engineered features, including required permissions, sensitive API calls, and certain dynamic behaviors obtained from DroidBox [7]. Deep4maldroid [18] constructs weighted directed graphs from Linux kernel system calls and use them to train deep neural networks for Android malware detection. Mclaughlin et al. [25] uses a deep neural network to detect Android malware according to how 218 dex instructions are used by each application. DeepRefiner [39] uses two detection layers with different deep neural networks to detect Android malware from different perspectives. FeatureSmith [44] uses natural language processing techniques to generate malware detection features from scientific literature.

Besides learning-based malware detection systems, enormous signature-based malware detection systems have been proposed. For example, Kirin [13] detects malware according to required permissions which break certain pre-defined security rules. Since they are not very close to this work, we refer readers to a recent survey [1] for more details.

VI. Conclusion

This paper presented DroidEvolver, an effective and efficient Android malware detection system that can automatically update itself so as to catch up with the rapid evolution of both malware and Android framework. Different from most learning-based malware detection systems which rely on batch learning algorithms for generating immutable detection models with fixed feature sets, DroidEvolver applies online learning algorithms to make necessary update to its detection models with evolving feature set. While most existing malware detection systems can be updated by retraining on a new set of applications with true labels, DroidEvolver requires neither retraining nor true labels to update itself; therefore, DroidEvolver is more practical in resource-constraint settings where true labels are not promptly available to many new applications. Rigorous experiments show that the performance of DroidEvolver is consistently higher than the state of the art in malware detection over time in terms of both accuracy and efficiency. DroidEvolver is also shown robust against several typical code obfuscation techniques. In the future, we plan to extend DroidEvolver using malware detection features other than Android API calls in both static analysis and dynamic analysis. The ultimate goal is to improve the accuracy of DroidEvolver to that updated by true labels. Our current work offers a promising first step towards this ultimate goal.

Acknowledgements

Ke Xu and Robert H. Deng are supported by AXA Research Fund. Yingjiu Li's work is supported by Huawei Research Project with agreement number YBN2018085241 from Huawei International Pte. Ltd. through Singapore Management University. Kai Chen is supported in part by NSFC U1836211, U1536106, 61728209, National Top-notch Youth Talents Program of China, Youth Innovation Promotion Association CAS, Beijing Nova Program, Beijing Natural Science Foundation (No.JQ18011) and National Frontier Science and Technology Innovation Project (No. YJKYYQ20170070).

References

[1] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, "Sok: Lessons learned from android security research for appified software platforms," in *IEEE Symposium on Security and Privacy*, 2016.

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

[3] T. Anderson, *The theory and practice of online learning*. Athabasca University Press, 2008.

[4] 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 *NDSS*, 2014.

[5] 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," in *Acm Sigplan Notices*, 2014.

[6] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines," in *ICML*, 2012.

[7] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck, "Droidbox: An android application sandbox for dynamic analysis," in *Lund University Technical Report*, 2011.

[8] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "Stormdroid: A streaminglized machine learning-based system for detecting android malware," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016.

[9] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer, "Online passive-aggressive algorithms," in *Journal of Machine Learning Research*, 2006.

[10] K. Crammer, A. Kulesza, and M. Dredze, "Adaptive regularization of weight vectors," in *Advances in neural information processing systems*, 2009.

[11] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," in *Journal of Machine Learning Research*, 2011.

[12] 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," in *ACM Transactions on Computer Systems (TOCS)*, 2014.

[13] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[14] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *ICLR*, 2015.

[15] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," in *CoRR*, 2016.

[16] S. C. Hoi, D. Sahoo, J. Lu, and P. Zhao, "Online learning: A comprehensive survey," *arXiv preprint arXiv:1802.02871*, 2018.

[17] S. C. Hoi, J. Wang, and P. Zhao, "Libol: A library for online learning algorithms," in *The Journal of Machine Learning Research*, 2014.

[18] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs," in *IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*, 2016.

[19] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting concept drift in malware classification models," in *26th USENIX Security Symposium*, 2017.

[20] M. Kloft and P. Laskov, "Security analysis of online centroid anomaly detection," in *Journal of Machine Learning Research*, 2012.

[21] LeiWang, "Baidu Security Lab," https://www.blackhat.com/docs/eu-16/, 2016.

[22] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *IEEE 39th Annual Computer Software and Applications Conference*, 2015.

[23] Mariconti, "MAMADROID Project," https://bitbucket.org/gianluca_students/mamadroid_code, 2018.

[24] 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 *NDSS*, 2017.

[27] 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)*, 2016.

[25] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé *et al.*, "Deep android malware detection," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017.

[26] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "Context-aware, adaptive, and scalable android malware detection through online learning," in *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2017.

[28] P. Palumbo, L. Sayfullina, D. Komashinskiy, E. Eirola, and J. Karhunen, "A pragmatic android malware detection procedure," in *Computers & Security*, 2017.

[29] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *IEEE European Symposium on Security and Privacy*, 2016.

[30] V. Rastogi, Y. Chen, and X. Jiang, "Catch me if you can: Evaluating android anti-malware against transformation attacks," in *IEEE Transactions on Information Forensics and Security*, 2014.

[31] D. Sahoo, C. Liu, and S. C. Hoi, "Malicious url detection using machine learning: A survey," *arXiv preprint arXiv:1701.07179*, 2017.

[32] A. K. Sikder, H. Aksu, and A. S. Uluagac, "6thsense: A context-aware sensor-based attack detector for smart devices," in *26th USENIX Security Symposium*, 2017.

[33] C. Smutz and A. Stavrou, "When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors." NDSS, 2016.

[34] M. Sun, T. Wei, and J. Lui, "Taintart: A practical multi-level information-flow tracking system for android runtime," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[35] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a java bytecode optimization framework," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, 1999.

[36] VirusTotal, "Virustotal-free online virus, malware and url scanner," *Online: https://www. virustotal. com/en*, 2012.

[37] R. Winsniewski, "Android–apktool: A tool for reverse engineering android apk files," 2012.

[38] L. Xiao, "Dual averaging methods for regularized stochastic learning and online optimization," in *Journal of Machine Learning Research*, 2010.

[39] K. Xu, Y. Li, R. H. Deng, and K. Chen, "Deeprefiner: Multi-layer android malware detection system applying deep neural networks," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.

[40] 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*, 2014.

[41] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: deep learning in android malware detection," in *ACM SIGCOMM Computer Communication Review*, 2014.

[42] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," in *Tsinghua Science and Technology*, 2016.

[43] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2014.

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

[45] M. Zinkevich, "Online convex programming and generalized infinitesimal gradient ascent," in *Proceedings of the 20th International Conference on Machine Learning (ICML)*, 2003.