Adaptive and Scalable Android Malware Detection through Online Learning

Annamalai Narayanan, Liu Yang, Lihui Chen and Liu Jinliang
Nanyang Technological University, Singapore.
annamala002@e.ntu.edu.sg, {yangliu, elhchen}@ntu.edu.sg, liuj0081@e.ntu.edu.sg

***Abstract*—It is well-known that malware constantly evolves so as to evade detection and this causes the entire malware population to be non-stationary. Contrary to this fact, prior works on machine learning based Android malware detection have assumed that the distribution of the observed malware characteristics (i.e., features) do not change over time. In this work, we address the problem of *malware population drift* and propose a novel online machine learning based framework, named DroidOL to handle it and effectively detect malware. In order to perform accurate detection, security-sensitive behavior are captured from apps in form of inter-procedural control-flow sub-graph features using a state-of-the-art graph kernel. In order to perform scalable detection and to adapt to the drift and evolution in malware population, an online passive-aggressive classifier is used.**

In a large-scale comparative analysis with more than 87,000 apps, DroidOL achieves 84.29% accuracy outperforming two state-of-the-art malware techniques by more than 20% in their typical batch learning setting and more than 3% when they are continuously re-trained. Our experimental findings strongly indicate that online learning based approaches are highly suitable for real-world malware detection.

keywords — Online Learning, Graph Kernels, Malware Detection

I. Introduction

Recently, malware detection for mobile platforms such Android has evolved as one of the challenging problems in the field of cyber-security. The number of new Android malware applications (apps for short) and their capabilities have grown tremendously in recent years. For instance, Kaspersky reports [1] detecting 4 million malware infections in 2015 which is a 216% increase over 2014. The sheer volume and growth rate of Android malware highlights an imperative need for developing sound and scalable automated malware detection process [2]–[7, 12].

Malware detection using Machine Learning (ML) techniques is predominant in various platforms (such as Windows, Android and the web) for more than a decade [2]–[7, 12]. This is because, these methods automatically learn the characteristics that distinguish malware, when trained using a collection of malware and benign samples making them amenable for automated detection. ML based approaches extract features from an apps' behaviors and apply standard ML algorithms to perform binary classification. These approaches typically use semantic features such as system calls/Application Programming Interfaces (APIs) invoked, resources and privileges used, control- and data-flows inside apps' execution to detect malicious behavior patterns [2]–[7, 12].

Most of these malware detection techniques are based on batch-learning classifiers. Meaning, the detection model is built using a batch of labelled benign and malware samples and is subsequently used to predict whether a given new sample is benign or malicious. These batch-learning based methods are typically plagued by two challenges that make them unsuitable for real-world large-scale malware detection:

- **Population drift.** Though batch-learning based solutions are promising, their success is predicated on an important assumption that may not hold for the malware detection problem. That is, they assume that the malware population (i.e., training data) used to build the detection model does not change over time. However, malware does not fit this profile. The entire population of malware is constantly evolving due to various reasons such as exploiting new vulnerabilities, and evading novel detection techniques. This evolution has a profound impact on malware characteristics and thereby on malware features. This makes the collection of malware identified today not a representative of the ones generated in the future. This phenomenon is an epitome of *population drift* [17]. Since new malware features emerge and importances of features change over time, this population drift leads to *concept drift* [17, 18].
- **Volume.** As noted before, malware grows at an alarming rate and hence a scalable classifier is of paramount importance to practical large-scale malware detection. In order to keep abreast with drifting population, batch learners have to be frequently re-trained with huge volumes of data. Hence they pose severe scalability issues when used in the Android malware detection context where we have millions of samples already and thousands streaming in every day. Retraining frequently with such a volume renders them computationally impractical.

Our Approach. We take these two challenges into consideration and propose DroidOL, an accurate, adaptive and scalable malware detection framework based on online learning, where we continuously retrain the model upon receiving each labeled sample and make prediction of a new sample using the updated model. We demonstrate that online learning based solutions are better suited for practical large-scale automated malware detection for two reasons:

- The detection model needs to adapt to changes in malware features over time, automatically.
- Online learning based solution can process large numbers of apps more efficiently than batch methods.

DroidOL's achieves superior accuracy through extracting high quality features from inter-procedural control-flow graphs (ICFGs) of apps, which are known to be robust against evasion

and obfuscation techniques adopted by malware [2, 5]. To this end, we use the Weisfeiler-Lehman (WL) graph kernel [13] that supports explicit feature vector representation of graphs to extract semantic features from ICFGs. DroidOL's adaptiveness and scalability are achieved through use of online learning.

Hence, the primary contribution of our paper is the successful application of online learning algorithms to the problem of Android malware detection. To the best of our knowledge, we are the first to propose such a framework and demonstrate its capabilities to handle *population drift*.

Experiments. DroidOL is evaluated through large-scale experiments on a recent real-world dataset of more than 87,000 apps. It is compared against two state-of-the-art batch-learning based Android malware detection techniques. DroidOL achieves 84.29% accuracy outperforming state-of-the-art techniques by more than 24% in their typical batch-learning and more than 3% when they are re-trained. Subsequently, we show that continuous retraining over newly emerging features is crucial for adapting the detection model to detect new or evolving malware.

In summary, our paper's contributions are as follows:

- We propose and develop DroidOL, an accurate, scalable and adaptive Android malware detection framework which is based on online learning, where we do not assume that the malware population is stationary (in §III).
- We conduct and report a large-scale comparative analysis of our framework against several re-trained variants of two state-of-the-art malware detection solutions on a sizable dataset of more than 87,000 apps (in §VI and VII).

The paper is organized as follows. We begin by discussing the related works on malware detection and also present our motivations in §II. The DroidOL framework is introduced in §III. Implementation details are discussed in §IV. DroidOL's evaluation, comparative analysis and relevant discussions are presented in sections V, VI and VII. Conclusions are presented in section VIII.

II. Related Work & Motivation

ML based approaches are popular over the past decade for malware detection. Many existing works have successfully applied ML techniques for malware detection on various platforms such as Windows, Android and the web.

A. Related Work - Android Malware Detection

1) Primitive Approaches: In the case of Android, Crowdroid [11], Drebin [4] and DroidAPIMiner [8] are noticeable among the early approaches on ML based malware detection. These methods were designed to detect malware operating on the initial versions of Android, performing simpler attacks such as making premium-rated calls/SMS. Hence, they leveraged on primitive features such as system calls, Android APIs and permissions. These techniques detect malware through identifying suspicious usage patterns of the aforesaid features. In particular, Crowdroid [11] uses Linux system call sequences as features. Drebin [4] uses APIs, permissions, components, accessed URLs and Intent filters as features. DroidAPIMiner [8] considers sensitive APIs along with parameters and package level information as features. Even though these features are good enough for detecting simpler malware, they are easily evaded by modern malware that perform sophisticated attacks [7, 9].

2) Robust Approaches: In order to detect stealthy malware, recent approaches leverage on two type of detection: (1) information-flow based detection and (2) graph based structural detection.

Information-flow based detection. These methods track the flow of sensitive information inside the apps' execution and detect malware by spotting suspicious flows. Even though these methods are highly precise, they exhibit poor scalability due to the expensive data-flow analysis they leverage on. Hence such methods are not suitable for practical large-scale malware detection [3, 4]. Mudflow [10], is a prototypical example of these types of detection methods.

Graph based structural detection. Graphs offer a natural way to model the sequence of activities that take place in a program. Hence they serve as amenable data-structures for detecting malware through identifying suspicious activity sequences. For this reason, graph representations such as call-graphs, control- and data-flow graphs, control-, data- and program-dependency graphs have been widely used for malware detection in conjunction with graph mining techniques [2, 3, 5, 7, 9, 12]. In the case of Android, DroidMiner [9] and Allix et al. [5] proposed to use control-flow graph based features to perform structural malware detection. DroidSIFT [7] and AppContext [2] proposed a more robust approach by including the contextual information of security-sensitive activities (i.e., whether or not the user is aware of such an activity) along with structural information captured through graphs.

These solutions are plagued by two limitations:
(1) *Loss of expressiveness:* These solutions follow a naïve approach to vectorise the graphs such as taking only individual nodes into consideration without their neighborhood. This leads to loosing the expressiveness of graphs.
(2) *Poor efficiency:* Many classic graph mining based approaches (e.g., [12]) are NP hard and have severe scalability issues, making them impractical for real-world malware detection [3].

Graph Kernels for malware detection. Recently, efficient and expressive graph kernels such as WL kernel [13] have been proposed and widely adopted in many application areas (e.g, bio- and chemo-informatics, computer vision, etc.). Some of these kernels also support building explicit feature vector representations of graphs. Taking notice of such a development, two approaches, Adagio [3] and Sahs et al. [6] used graph kernels to perform structural detection of Android malware.

B. Motivation

While all the above mentioned works focus on engineering robust features that can detect malware effectively, they do not address a key practical aspect of malware detection problem—*malware evolution*. As discussed in §I, many analytical studies such as [17, 21] have clearly highlighted that malware evolves in terms of its characteristics for various reasons. The evolution inevitably leads to profound changes in the malware features over time i.e., *concept drift*. This poses two challenges:
(C1) the detection model has to automatically adapt to the

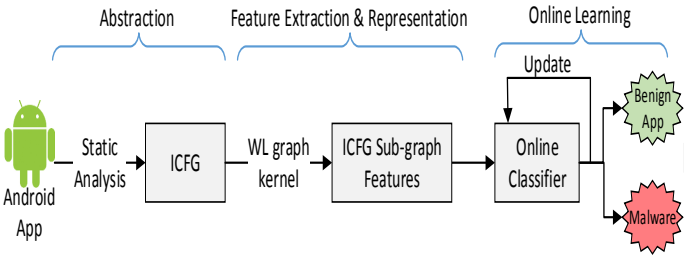

Fig. 1: DroidOL framework for performing online learning based Android malware detection

concept drift.

(C2) the detection model has to consider and account for new features that emerge over time.

Malware detection as a data-stream classification problem. While none of the existing Android malware detection techniques address this concept drift, we get a clear motivation to do so. While all the existing works consider malware detection as a *static classification problem* with a fixed training and test set, we note that this is against the real-world situation. Hence, we model it as a *data stream classification problem*. As for the two aforementioned challenges, we address the first challenge (C1) through the use of an online learning classifier which adapts itself to the drift in characteristics of samples that stream in and for the second challenge (C2), we use a *dynamic feature space*, where the number of features keeps growing over time (see §VI-C for a detailed explanation).

While we note that there a few approaches in the past that have adopted online learning to tackle concept drift in malware for other platforms such as Windows [20], we are the first to do so for Android.

Features. As discussed in the previous subsection II-A, graph based features have been proven effective and robust to several evasion strategies followed by malware and hence we use them. While choosing a particular graph representation, we note that building data-flow and data-dependency graphs is not scalable and call-graphs are too coarse-grained to capture the program semantics perfectly. Hence, we use ICFG representation as they efficiently capture finer program semantics.

Graph Kernel. Since WL graph kernel [13] is the current state-of-the-art graph kernel, known for its expressiveness and efficiency, we use it to extract semantic features from ICFGs.

III. DroidOL - Framework Overview

The overview of DroidOL framework which performs accurate, adaptive and scalable malware detection using online learning is presented in Fig. 1. DroidOL has three phases. We begin by performing static analysis on a given set of apps to get their ICFG representations. Subsequently, ICFG subgraph features are extracted using the WL kernel and the apps are represented as feature vectors. Finally, an online PA classifier is trained with these vectors to detect malware. Each of the phases is described in detail below.

1) **Abstraction.** Our malware detection approach considers node-labeled ICFG sub-graphs as features. To extract these features, we first perform Android-specific static analysis to transform all the apps into their corresponding ICFG representations. subsequently, each ICFG's nodes are labeled with security sensitive APIs[1] that they access. Formally, $ICFG = (N, E, \lambda)$ is a directed graph where N is a set of nodes and each node $n \in N$ denotes an instruction in the disassembled format. $E \subseteq (N \times N)$ is a set of edges and each edge $e(n_1, n_2) \in E$ denotes the control-flow from n_1 to n_2. λ is the set of security-sensitive Android APIs and $\ell : N \rightarrow \lambda$, is a labeling function which assigns an API as label to each node.

Algorithm 1 WL kernel — extracting vocabulary for feature vector representation of ICFGs

Input:
$\mathcal{G} = \{ICFG_1, ICFG_2, ..., ICFG_K\}$: A set of K ICFGs (one for each of the K apps in the dataset.)
h: Degree of neighborhood to be considered for label enrichment
Output:
Σ: Vocabulary of sub-graph features present across all graphs in $\mathcal{G}$

procedure EXTRACT VOCAB($\mathcal{G}, h$)
  $\Sigma \leftarrow \phi$
  for $ICFG = (N, E, \lambda_0) \in \mathcal{G}$ **do**
    for $i = 0$ to h **do**
      for $n \in N$ **do**
        if $i = 0$ **then**
          $\lambda_0(n) \leftarrow \lambda(n)$
        else
          $\mathcal{N}(n) \leftarrow \{m \mid (n, m) \in E\}$
          $M_i(n) \leftarrow \{\lambda_{i-1}(m) \mid m \in \mathcal{N}(n)\}$
          $\lambda_i(n) \leftarrow \lambda_{i-1}(n) \oplus sort(M_i(n))$
        end if
        $\Sigma \leftarrow \Sigma \cup \lambda_i(n)$
      end for
    end for
  end for
  return Σ
end procedure

2) **Feature Extraction & Representation.** Once the ICFGs are constructed, (rooted) sub-graphs in these ICFGs that represent the security-sensitive behaviors in every app are extracted using the WL graph kernel [13]. Subsequently, ICFGs are represented as feature vectors.

WL kernel. The WL kernel works by augmenting the labels of every node n with its neighborhood (up to a certain degree) in a given graph, G. The frequency of these enriched node labels which denote sub-graphs around every node in G are used as features to facilitate an explicit feature vector representation of G.

Thus the process of obtaining feature vector representation of ICFGs in our dataset using WL kernel involves two steps: (1) Building a vocabulary, Σ of sub-graph features present across all ICFGs, (2) Transforming every ICFG into a feature vector with $|\Sigma|$ dimensions. Step (1) is presented in detail in Algorithm 1.

Algorithm Explanation. The inputs to Algorithm 1 are $\mathcal{G}$, a set of K ICFGs (one for each of the K apps in the dataset) and h, the degree of sub-graphs to be considered for feature extraction. The output is a vocabulary set Σ

[1] PScout [19], an existing research work identifies and lists the security-sensitive Android APIs. These APIs are used for labeling our ICFG nodes.

that contains the unique sub-graphs upto degree h present across all ICFGs in $\mathcal{G}$.

In each iteration i of the algorithm (lines 4 to 15), the neighborhood up to degree i around a node n is captured and condensed in the form of a neighborhood label, $\lambda_i(n)$. These neighborhood label is what we refer to as sub-graph features. For every $ICFG_k \in \mathcal{G}$, the following process is adopted to obtain these neighborhood labels.

For the initial iteration $i = 0$ no neighborhood information needs to be considered. Hence the neighborhood label $\lambda_0(n)$ is same as the original node label $\lambda(n)$ (line 7). For $i > 0$, the following procedure is used for label enrichment: Firstly, for a node $n \in N$, all of its neighboring nodes are obtained and stored in $\mathcal{N}(n)$ (line 9). For each node $m \in \mathcal{N}(n)$ the neighborhood label up to degree $i-1$ is obtained and stored in multiset $M_i(n)$ (line 10). Subsequently, $\lambda_{i-1}(n)$, neighborhood label of n till degree $i-1$ is concatenated to the sorted value of $M_i(n)$ to obtain the current neighborhood label, $\lambda_i(n)$ (line 11). Finally, the neighborhood label $\lambda_i(n)$ representing the i^{th} degree neighborhood around n is added to the vocabulary of sub-graph features Σ (line 13).

Once the vocabulary Σ is obtained, we transform every $ICFG \in \mathcal{G}$ into its corresponding vector representation by counting the frequency of every feature from Σ in $ICFG_k$. This procedure falls under the well-known Bag-of-Features (BoF) representation model [13], where every ICFG is considered as a bag of sub-graphs.

3) **Online Learning.** Once the feature vectors of all the apps in the training-set are built, we train an online PA classifier with these vectors to detect malware. PA classifier's training and update procedures are as explained below with relevant notations.

Denote the features of an app (both benign and malware) as a vector $x = [x^{(1)}, x^{(2)}, ..., x^{(d)}]^T$, and its label as $y \in \{-1, +1\}$, where -1 indicates benign and $+1$ indicates malicious apps. The PA classifier receives a number of samples, x_i, and their labels, y_i, and trains using this labeled data. Given a new unseen sample, x, the goal of PA classifier is to predict the label, y, of this new sample based on its trained model.

PA being a linear classifier fits a linear decision boundary (i.e., hyperplane) between the positive and negative class samples. That is, the model is a weight vector, $w = [w^{(1)}, w^{(2)}, ..., w^{(d)}]^T$ which indicates the weight (i.e. relative importance) of each of the features used to predict the output label y. The predicted label, $\hat{y}$, is the sign of the inner product between x and w:

$$\hat{y} = sign(x \cdot w) \tag{1}$$

PA incrementally build the models in rounds. In round t, PA receives a sample, x_t and predicts its label $\hat{y_t}$ using the current model; it then receives y_t, the true label of x_t and updates its model based on the sample-label pair: (x_t, y_t), if it makes a wrong prediction. That is, w is updated if the predicted label, $\hat{y_t}$ and the true label, y_t of the sample x_t are not the same. The goal of the PA algorithm is to update the model w as minimal as possible to correct for any mistakes it commits. PA solves the following optimization with each given sample:

$$w_{t+1} \leftarrow \underset{w}{\text{argmin}} \frac{1}{2}||w_t - w||^2 \quad \text{subject to } y_i(w \cdot x_t) \geq 1 \tag{2}$$

Updates occur only when $y_t(w_t \cdot x_t) < 1$., The closed-form update for all samples is as follows:

$$w_{t+1} \leftarrow w_t + \alpha_t y_t x_t \tag{3}$$

where $\alpha_t = max\{\frac{1-y_t(w_t \cdot x_t)}{||x_t||^2}, 0\}$ (we refer the reader to the original work at [15] for this derivation and further details on PA algorithm).

Once the PA classifier in DroidOL is trained with all these samples it is ready to perform malware detection at scale. It is important to note that since DroidOL is trained in an online fashion, it performs malware detection and simultaneously adapts to the changing trends in malware features by retraining on every sample it misclassifies.

Alternatives. As practitioners involved in building an online malware detection framework, we do not have any vested interest in a particular algorithm for online learning. Ultimately, we wish to determine the algorithm that scales well to problems of our size and yields the best performance. To that end, we experimented with other well-known online learning algorithms such as Online Perceptron (OP) and Stochastic Gradient Descent learning based Logistic Regression (LR_{SGD}) along with PA. Since PA offered the best results in terms of accuracy, it is used in DroidOL. We believe this is because OP and LR_{SGD} do not imbibe the notion of classification confidence and treat all misclassifications equally, unlike PA. PA, on the other hand, updates more aggressively when the margin of error is large and less aggressively in when it is small. While we note that evolving classifiers such as pClass [22] could be used for handling concept drift, we prefer PA over such methods, as it offers better efficiency.

IV. Implementation & Comparative Analysis

We implemented the DroidOL framework in approximately 15,600 lines of Python and Java code. Soot[2], a popular Android static analysis workbench is used for constructing the ICFGs of the apps in our dataset. Scikit-learn[3] toolbox is used for all our ML functionalities.

We compare our online learning based detection against two state-of-the-art Android malware detection solutions, namely Drebin [4] and Allix et. al.'s [5]. It is noted that both these methods use batch learning classifiers.

Drebin [4] is well-known for its scalable and explainable detection. It extracts light-weight features such as APIs, permissions, URLs accessed, names of components from apps and subsequently, trains a linear SVM classifier to distinguish malware from benign apps.

Allix et al. [5] recently proposed another scalable approach using structural features, namely CFG signatures. Therefore, we refer to this technique as CFG-Signature Based Detection (CSBD) in the reminder of the paper. CSBD constructs CFGs of individual methods and encodes them as text-signatures.

[2]http://sable.github.io/soot

[3]http://scikit-learn.org

TABLE I: Dataset with apps dated from Jan 1, 2014 to Aug 13, 2014

Market Name	# of Benign Apps	# of Malware
Google Play	39156	26178
Anzhi	2957	12260
AppChina	1845	4154
SlideMe	289	132
HiApk	65	157
FDroid	29	2
Angeeks	6	27

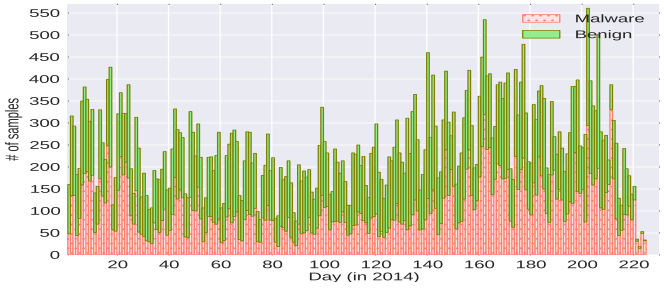

Fig. 2: Timeline based distribution of apps in our large-scale dataset (with malware and benign apps proportions).

Subsequently, a Random Forest classifier is trained with these signatures to detect malware.

V. Data collection

We evaluate our approach on a recent large-scale real-world dataset of 87,257 apps collected in-the-wild. These apps are collected from seven different Android markets[4], namely, *Google Play, Anzhi, AppChina, SlideMe, HiApk, FDroid* and *Angeeks*, in 2014. Following the common practice in software security research, we use the *Virus Total*[5] web portal which hosts malware detection services from more than 40 Anti-virus scanners to determine the ground-truth labels of the apps. We infer that dataset contained 44,347 benign and 42,910 malware apps. The composition of the dataset is presented in Table I. It is noted that this is a subset of a large collection of apps used in [23].

The date of creation of these apps fall in a span of 224 days starting from 1 Jan'14 to 13 Aug'14. We intend to sort these apps according to their date of creation and emulate a live feed of apps to the malware detectors considered in our experiments to examine how they cope up with the drift in the malware characteristics over time. To this end, we divide all the apps in the dataset (both benign and malware) into batches according to their date of creation. Hence we have 224 batches, one for each day. The resulting time-line based distribution of the two datasets are presented in Fig. 2.

VI. Evaluation

In this section, we evaluate DroidOL's accuracy and adaptiveness using the emulated live feed of apps. To this end, we address the following questions:
(1) Does DroidOL's online learning provide any benefit over batch learning for malware detection?
(2) How does DroidOL's accuracy compare to light-weight state-of-the-art malware detection techniques?
(3) Is there a particular training regimen that fully realizes the potential of online classifier?

A. Advantages of Online Learning

We start by evaluating the benefit of using online over batch learning for the problem of malware detection in terms of detection accuracy — in particular, whether the benefit of DroidOL's efficiency comes at the expense of accuracy. Specifically, we compare DroidOL's PA based classification against four different training configurations of a canonical batch learning classifier. We used Linear SVM as our canonical batch learner. It is noted that evaluations with other batch algorithms such as logistic regression yielded similar results.

Batch Learning Configurations. For SVM, we experiment with the following variants: SVM-Once, SVM-Daily, SVM-MultiOnce, and SVM-MultiDaily. For SVM-Once, SVM classifier is trained only once on the apps from Day 1 (1 Jan'14). This model is tested on all other days without retaining. For SVM-Daily, the classifier is retrained after every day; however, only one previous day's samples are used for every retraining — e.g., 11 Jan'14 results reflect training on the apps created on 10 Jan'14, and testing on 11 Jan'14 apps. SVM-MultiOnce is similar to SVM-Once and SVM-MultiDaily is similar to SVM-Daily, however, with the size of the batch for training and retraining covers 10 days instead of 1. In summary, for Once and MultiOnce variants, the model is never re-trained and for Daily and Multi-Daily, the model is re-trained in a sliding window fashion over the batches of data. The size of MultiDaily training sets is determined to be 10-day batches based on the data that our evaluation machine with 32 GB RAM can handle.

Figs. 3 (a) shows the cumulative error rates of DroidOL in comparison to the aforementioned variants of SVM. The following observations are made from Fig. 3 (a):

- Updating the detection models over time is essential to detect new malware as shown by SVM-MultiDaily and SVM-Daily outperforming SVM-MultiOnce and SVM-Once, respectively.
- Training on significantly more data improves the performance, as illustrated by SVM-MultiDaily and SVM-MultiOnce outperforming SVM-Daily and SVM-Once, respectively. However, it is noted that there is a fundamental limit on the amount of data a batch-learning technique could train on because of the storage, memory and time requirements.
- Third, DroidOL consistently outperforms all the batch learnt variants. In particular, it outperforms the best retrained variant of SVM by more than 4% cumulative error rate. This is because DroidOL is able to adapt to the changes in the malware characteristics instantaneously as well as retain significantly useful information from the past.

[4]Google Play: https://play.google.com/store, Anzhi: www.anzhi.com, AppChina: www.appchina.com, SlideMe: www.SlideME.org, HiApk: www.hiapk.com, FDroid: www.fdroid.org and Angeeks: http://apk.angeeks.com

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

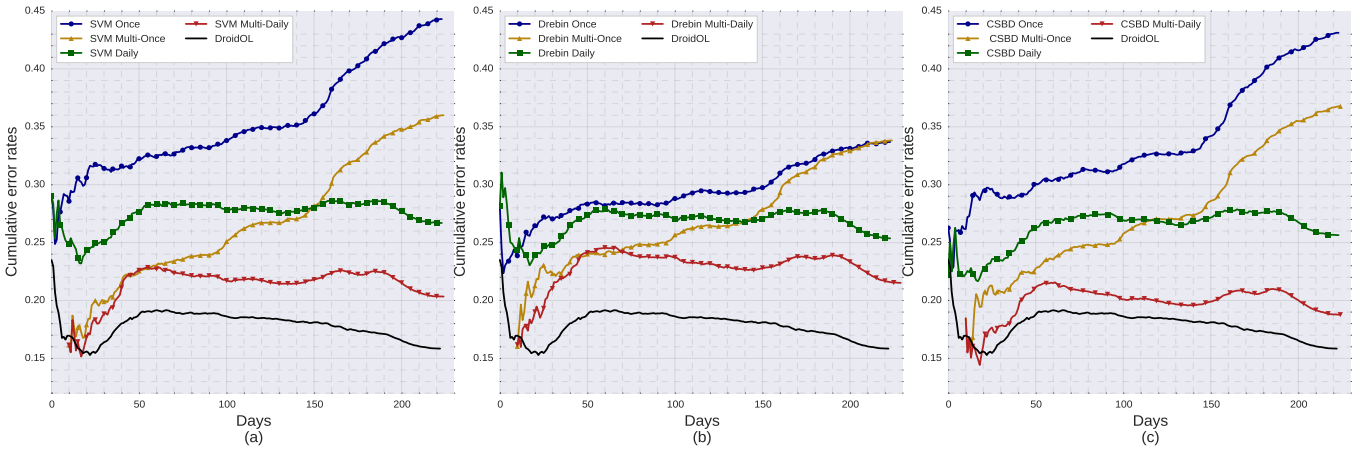

Fig. 3: Cumulative error rates for DroidOL vs. batch learning algorithms (a) DroidOL Vs. four variants of SVM (b) DroidOL Vs. four variants of Drebin (c) DroidOL Vs. four variants of CSBD.

B. Comparison with state-of-the-art Malware Detectors

The above-mentioned SVM variants use the same features as DroidOL and hence it is sufficient to compare online and batch learning paradigms. However, this does not reflect the significance of DroidOL as a practical malware detector in the context of current state-of-the-art malware detection techniques. In order to study this, we compare DroidOL with two state-of-the-art malware detectors, namely, Drebin [4] and CSBD [5].

For this comparison, we follow the same batch learning configurations described in §VI-A to arrive at the four variants of these techniques: Drebin/CSBD-Once, Drebin/CSBD-Daily, Drebin/CSBD-MultiOnce and Drebin/CSBD-MultiDaily.

The results of this comparison are presented in Fig 3 (b) and (c). From these figures we make the following inferences:

- For both these methods, the trends in performance of all four variants are similar to those of the SVM variants. Hence the observations made in §VI-A on frequently updating the models and training with more data, hold. The error rates of best-performing variants of these methods are comparable to that of SVM-MultiDaily.
- DroidOL consistently outperforms the best performing variants of both the methods. Particularly, it outperforms Drebin-MultiDaily by more than 5% and CSBD-MultiDaily by more than 3%. This reaffirms the suitability of online learning for practical large-scale malware detection.

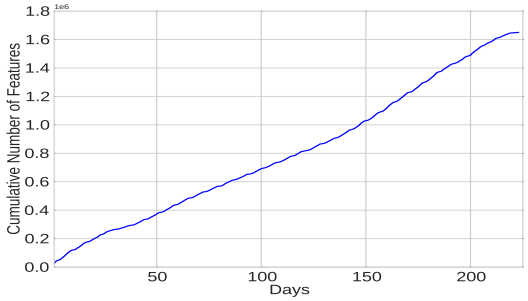

Fig. 4: Cumulative number of features observed over time for our large-scale dataset.

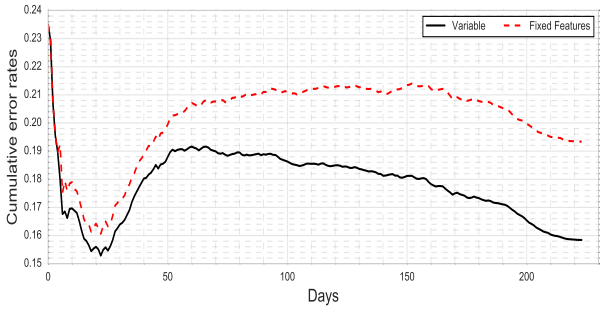

Fig. 5: Benefits of using variable-feature sets over fixed-feature sets.

C. Training Regimen

Since DroidOL's feature extraction using WL kernel is based on BoF model, our number of features grows as the samples stream in. Fig. 4 shows the cumulative number of features for each day of the evaluations in our dataset, representing the feature space growth. Each day's total includes new features introduced that day and all the old features from previous days. We obtained a total of 13,655 features from the malware and benign samples encountered on Day 1 (1 Jan'14). The dimensionality grows quickly as we extract new subgraph features from samples encountered every day and while reaching the final day (13 Aug'14), we have accumulated 1,653,496 features. This phenomenon of feature space growth is common across many techniques including Drebin and CSBD. This is because Android apps evolve over time for various reasons such as capability enhancements, bug fixes, using newly introduced Android functionalities and adapting to changes in Android framework APIs [2, 7]. This evolution results in newly observed characteristics which translate into

new features from a ML view point.

Now, we are posed with the question: *should we consider these new features that emerge every day?* To address this, we devise two types of *training regimen*, namely, *variable feature-set training* and *fixed feature-set training*.

Fixed feature-set training regimen. Under the *fixed feature-set* regimen, we train using a pre-determined set of features for all evaluation days. That is, we fix the features to those encountered up to Day 1, then we use those 13,655 features for the whole experiment.

Variable feature-set training regimen. Under the *variable feature-set* regimen, we allow the dimensionality of our PA classifier to grow with the number of new features encountered; on the last day, for instance, we classify with more than 1.6 million features. Implicitly, examples that were introduced before a feature *i* was first encountered will have value 0 for feature *i*.

Fig. 5 shows the importance of using *variable feature-set training* over *fixed feature-set training*. We see that the performance for fixed feature regimen is significantly and consistently inferior to the variable feature regimen. The latter regimen has a 3.3% lesser cumulative error rate. This reveals that, continuous retraining with a *variable feature-set* allows a model to successfully adapt to new data and new features on a sub-day granularity. This adaptiveness is critical to realize the full benefits of online learning. This indicates that choosing the right training regimen can be just as important as choosing the right classifier. The aforementioned training regimens can help online algorithms stay abreast of changing trends in malware and benign apps' features.

VII. Understanding the Performance

In this section we seek to get deeper insights into the superior performances of DroidOL's online learning and to understand how characteristics of the Android malware detection task affects its performances. Specifically, we evaluate and quantify the importance of long term memory and fast model update. To this end, we pose the following question: *Why does DroidOL outperform batch-learning solutions in detecting malware, even when they are re-trained?*

Dataset. In order to provide insights into our results, we need the notion of similarity among the malware samples. More specifically, we need the malware to be grouped according to their families. Malware familial analysis and grouping is currently a manual process [2, 4]. It is impractical to group apps from our large-scale dataset according to their families. Hence we use a well-known benchmark dataset that has malware readily grouped according to their family, namely, Drebin_{5k}[6] [4]. Drebin_{5K} contains a total of 5560 malicious Android apps belonging to 179 malware families. The date of creation of these samples lie in the range: Mar'09 to Oct'12.

Familial similarity and Notations. We now introduce the notion of familial similarity between a pair of malicious apps. We consider two malware m_1 and m_2 as variants, denoted by $m_1 \sim m_2$, if they belong to the same malware family. For instance, Drebin_{5K} dataset contains 965 samples belonging to the *FakeInstaller* family. We consider each of pair of them as variants. This stems from the fact that malware belonging to the same family exhibit similar malicious characteristics, making them exhibit similar semantic features from a ML view point.

[6]In order to distinguish the dataset provided by Drebin authors from the Drebin malware detection technique discussed in §VI, we refer to the dataset as Drebin_{5K}.

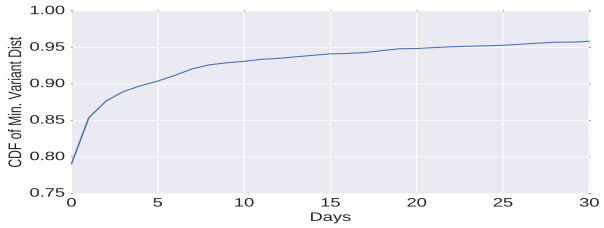

Fig. 6: The Head of CDF of minimum variant distance in Drebin_{5K} dataset

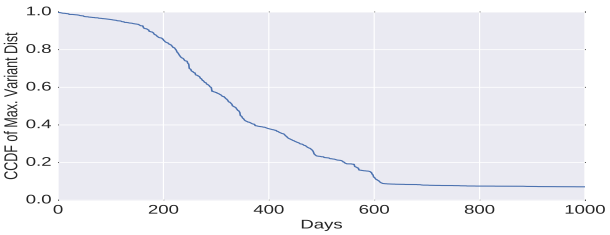

Fig. 7: The Head CCDF of maximum variant distance in Drebin_{5K} dataset

Subsequently, for each malware m, set M contains all the malware belonging to the same family as m. We calculate the time difference between the creation of m and every other malware, $m' \in M$, denoted as $\triangle(m, m')$. We call the minimum delay between another variant of the same family as m as *minimum variant delay*, denoted by $\triangle_{min}(m)$. Similarly, we call the maximum delay between another variant of the same family as m as *maximum variant delay*, denoted by $\triangle_{max}(m)$.

If there are no variants of for a malware m, its default values for $\triangle_{min}(m)$ and $\triangle_{max}(m)$ are 0 and D, respectively, where D is the difference between the date of creation of m and that of the latest app in Drebin_{5K} dataset.

Importance of fast model update. Fig. 6 depicts the head of the cumulative distribution function (CDF) of the *minimum variant delay* of the malware samples in the Drebin_{5K} dataset. It is clear that nearly 80% malware have a *minimum variant delay* of zero days. Meaning, for a given malware sample, at least one other variant of the same of family is built (and probably released) on the same day itself. This is understandable as malware authors aim to maximize their gains by launching several polymorphic variants of their malware, around the same time, leveraging on techniques such as obfuscation. Hence in order to keep abreast with quick-succession releases batch-learning solutions have to be updated at least daily (ideally, they have to be updated continuously). This explains why re-training SVM/Drebin/CSBD solutions every day yields higher classification accuracy in sections VI-A and VI-B. Therefore, unless the detector is updated continuously to reflect the most recent features present in the last malware, it will not be able

to effectively detect a majority of its variants.

Importance of long term memory. Fig. 7 depicts the head of the complementary cumulative distribution function (CCDF) of the *maximum variant delay* of all the malware samples in the Drebin_{5K} dataset. We observe that more than 40% of malware have a *maximum variant delay* of more than a year. Which means, a significant number of malware families keep evolving for a few years. This is understandable as malware authors make new variants either to evade known detection techniques or to improve/enhance their attacks for a prolonged period of time.

Therefore, if the batch size is limited to a few days or months, these 40% of malware would not have many similar variants in the batch. This leads to a significant reduction the detection accuracy because the detection model, has not yet learned enough on malware that is similar to these malware by the time it needs to classify them. In general, Fig. 7 demonstrates that classifying malware require long term memory. This explains why extending SVM/Drebin/CSBD batch sizes yielded better classification performance in sections VI-A and VI-B. However, when using batch-learning based detection solutions, the batch size is limited by the amount of available memory. Particularly, for a problem such as Android malware detection where we have millions of samples in a year and typical ML solutions extracting thousands of features, having a batch size of more than a year is not practical. On the other hand, online learning algorithms do not have this limitation. They retain significant useful information from all of the malware that they have seen. In other words, online algorithms operate with effectively infinite batch size. This explains why they have an edge over batch-learning solutions in our experiments.

Summary. From figures 6 and 7 it is clear that to perform reasonably accurate and adaptive malware detection, the batch-learning solutions should re-train every day with a batch size of at least a year. This strategy is highly expensive in terms of computation time and resources, rendering it impractical. Hence, we can confidently conclude, online learning based methods which do not have such computational limitations and inherently adaptive are better suited for Android malware detection.

VIII. Conclusions

In this paper, we present DroidOL, an accurate, adaptive and scalable Android malware detection framework. DroidOL's unique feature is its ability to handle *population drift* in Android malware through the use of online learning. DroidOL exhibits high accuracy, as it extracts effective structural features from apps using a state-of-the-art graph kernel. Further, DroidOL adapts automatically to the drift in malware characteristics over time and exhibits high scalability, making it suitable for real-world malware detection.

Our large-scale evaluations on a real-world dataset demonstrates that DroidOL outperforms state-of-the-art malware detectors. DroidOL achieves 84.29% accuracy outperforming existing techniques by more than 20% in their typical batch-learning setting. This superior performance make DroidOL, in particular, and online learning based solutions, in general, better candidates for practical large-scale malware detection.

Acknowledgment

We thank the authors of [4] and [5], for their suggestions and discussions that helped us re-implement their methods. We thank Kevin Allix for sharing the dataset used in [23].

References

[1] Kaspersky 2014 Annual Threat Report. URL. https://securelist.com/files/2015/12/Kaspersky-Security-Bulletin-2015_FINAL_EN.pdf

[2] Yang, Wei, et al. "Appcontext: Differentiating malicious and benign mobile app behaviors using context." Proc. of the International Conference on Software Engineering (ICSE). 2015.

[3] Gascon, Hugo, et al. "Structural detection of android malware using embedded call graphs." Proceedings of the 2013 ACM workshop on Artificial intelligence and security. ACM, 2013.

[4] Arp, Daniel, et al. "Drebin: Effective and explainable detection of android malware in your pocket." Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS). 2014.

[5] Allix, Kevin, et al. "Empirical assessment of machine learning-based malware detectors for Android." Empirical Software Engineering (2014): 1-29.

[6] Sahs, Justin, and Latifur Khan. "A machine learning approach to android malware detection." Intelligence and Security Informatics Conference (EISIC), 2012 European. IEEE, 2012.

[7] Zhang, Mu, et al. "Semantics-aware Android malware classification using weighted contextual API dependency graphs." Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014.

[8] Aafer, Yousra, Wenliang Du, and Heng Yin. "DroidAPIMiner: Mining API-level features for robust malware detection in android." Security and Privacy in Communication Networks. Springer International Publishing, 2013. 86-103.

[9] Yang, Chao, et al. "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications." Computer Security-ESORICS 2014. Springer International Publishing, 2014. 163-182.

[10] Avdiienko, Vitalii, et al. "Mining apps for abnormal usage of sensitive data." Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on. Vol. 1. IEEE, 2015.

[11] Burguera, Iker, Urko Zurutuza, and Simin Nadjm-Tehrani. "Crowdroid: behavior-based malware detection system for android." Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices. ACM, 2011.

[12] Fredrikson, Matt, et al. "Synthesizing near-optimal malware specifications from suspicious behaviors." Security and Privacy (SP), 2010 IEEE Symposium on. IEEE, 2010.

[13] Shervashidze, Nino, et al. "Weisfeiler-lehman graph kernels." The Journal of Machine Learning Research 12 (2011): 2539-2561.

[14] Google Play Store. URL. https://play.google.com/store

[15] Crammer, Koby, et al. "Online passive-aggressive algorithms." The Journal of Machine Learning Research 7 (2006): 551-585.

[16] Blum, Avrim. On-line algorithms in machine learning. Springer Berlin Heidelberg, 1998.

[17] Singh, Anshuman, Andrew Walenstein, and Arun Lakhotia. "Tracking concept drift in malware families." Proceedings of the 5th ACM workshop on Security and artificial intelligence. ACM, 2012.

[18] Ma, Justin, et al. "Identifying suspicious URLs: an application of large-scale online learning." Proceedings of the 26th Annual International Conference on Machine Learning. ACM, 2009.

[19] Au, Kathy Wain Yee, et al. "Pscout: analyzing the android permission specification." Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012.

[20] Masud, Mohammad M., et al. "Cloud-based malware detection for evolving data streams." ACM Transactions on Management Information Systems (TMIS) 2.3 (2011): 16.

[21] Zhou, Yajin, and Xuxian Jiang. "Dissecting android malware: Characterization and evolution." Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012.

[22] Pratama, Mahardhika, et al. "pClass: an effective classifier for streaming examples." Fuzzy Systems, IEEE Transactions on 23.2 (2015): 369-386.

[23] Allix, Kevin (2015). "Challenges and Outlook in Machine Learning-based Malware Detection for Android." (Doctoral dissertation). Retrieved from URL. http://hdl.handle.net/10993/24900