A Machine Learning Approach to Android Malware Detection

Justin Sahs and Latifur Khan
University of Texas at Dallas
Dallas TX 75080, USA
Email: {jcs074000,lkhan}@utdallas.edu

***Abstract*—With the recent emergence of mobile platforms capable of executing increasingly complex software and the rising ubiquity of using mobile platforms in sensitive applications such as banking, there is a rising danger associated with malware targeted at mobile devices. The problem of detecting such malware presents unique challenges due to the limited resources avalible and limited privileges granted to the user, but also presents unique opportunity in the required metadata attached to each application. In this article, we present a machine learning-based system for the detection of malware on Android devices. Our system extracts a number of features and trains a One-Class Support Vector Machine in an offline (off-device) manner, in order to leverage the higher computing power of a server or cluster of servers.**

***Index Terms*—Computer Security, Data Mining, Support Vector Machines**

I. Introduction

The smartphone has rapidly become an extremely prevalent computing platform, with just over 115 million devices sold in the third quarter of 2011, a 15% increase over the 100 million devices sold in the first quarter of 2011, and a 111% increase over the 54 million devices sold in the first quarter of 2010 [25], [26]. Android in particular has seen even more impressive growth, with the devices sold in the third quarter of 2011 (60.5 million) almost triple the devices sold in the third quarter of 2010 (20.5 million), and an associated doubling of market share [26]. This popularity has not gone unnoticed by malware authors. Despite the rapid growth of the Android platform, there are already well-documented cases of Android malware, such as DroidDream, which was discovered in over 50 applications on the official Android market in March 2011 [30]. Furthermore, Enck *et al.* [12] found that Android's built-in security features are largely insufficient, and that even non-malicious programs can (unintentionally) expose confidential information. A study of 204,040 Android applications conducted in 2011 found 211 malicious applications on the official Android market and alternative marketplaces [37].

The problem of using a machine learning-based classifier to detect malware presents two main challenges: first, given an application, we must extract some sort of feature representation of the application; second, we have a data set that is almost exclusively benign, so we must choose a classifier that can be trained on only one class. To address the first problem, we extract a heterogeneous feature set (described in Section IV), and process each feature independently using multiple kernels (described in Section V). To address the second problem, we use a One-Class Support Vector Machine [27], which we train using only benign applications.

This paper is organized as follows: in Section II, we discuss related work; in Section III, we present an overview of our approach; in Section IV, we present the various features that we extract from our data set; in Section V, we present the kernels we use to train our model; in Section VI, we discuss our experimental results; finally, in Section VII, we analyze our results and present possible future research.

II. Related Work

There has been significant work on the problem of detecting malware on mobile devices. Several approaches [11], [16], [18] monitor the power usage of applications, and report anomolous consumption. Others [6], [36] monitor system calls and attempt to detect unusual system call patterns. Other approaches use more traditional comparison with known malware (e.g. [4]), or other heuristics (e.g. [37]).

The more general field of malware detection is host to a wider range of approaches. Traditional static analysis approaches such as [8], [19], which focus on comparing programs to known malware based on the program code, looking for signatures or using other heuristics. Other approaches [17], [28], [32] focus on using machine learning and data mining approaches for malware detection. In [32], Tesauro *et al.* train a neural network to detect boot sector viruses, based on byte string trigrams. Schultz *et al.* [28] compare three machine learning algorithms trained on three features: DLL and system calls made by the program, strings found in the program binary, and a raw hexadecimal representation of the binary. In [17], Kolter and Maloof train several machine learning algorithms on byte string n-grams.

III. Approach

We use an open source project, Androguard [10], to extract features from packaged Android applications (APKs). We then use these extracted features to train a One-Class Support Vector Machine [27], using the Scikit-learn framework [24], which provides a convenient interface to LIBSVM [7].

The main idea in [27] is to generate a classifier that will classify most of the training data as positive, and classify training or testing data as negative only if it is sufficiently different from the training data, making it ideal for our

978-0-7695-4782-4/12 $26.00 © 2012 IEEE
DOI 10.1109/EISIC.2012.34

purposes, as benign Android applications are far more readily accessible than malicious ones. As a support vector machine, the One-Class SVM is a linear classifier in a high-dimensional feature space based on the constrained quadratic optimization problem:

$$\min_{w,\vec{\xi},\rho} \quad \frac{1}{2}\|w\|^2 + \frac{1}{\nu l}\sum_i \xi_i - \rho \tag{1}$$
$$\text{such that} \quad (w \cdot \phi(x_i)) \geq \rho - \xi_i, \xi_i \geq 0\,.$$

This gives the classifier function

$$f(x) = \operatorname{sign}\left(w \cdot \phi(x) - \rho\right)\,. \tag{2}$$

Here, $\phi(x)$ is a feature space transformation aimed at mapping the features of x into a space where the classes are linearly seperable, and w is some element of that space. Then, the optimization problem (1) tries to find a linear separation such that it separates the training data from the origin, and its distance from the origin is maximal.

Rather than supply $\phi(\cdot)$ directly, we instead supply a *kernel*,

$$k(x, y) = \langle \phi(x), \phi(y) \rangle\,. \tag{3}$$

The required properties of kernels, and our particular choice of $k(\cdot,\cdot)$ are described in Section V.

IV. Feature Extraction

In order to apply any kernel, we must first extract important features from the application. Android applications are packaged as APK files, which are similar to standard Java jar files. We use the open source project Androguard [10] to process these files and extract features. Androguard provides a fairly easy-to-use interface for analyzing and reverse engineering Android applications.

A. Permissions

Every APK must include a manifest file that, among other things, requests permission to access certain restricted elements of the Android operating system. These elements include access to various hardware devices (e.g. GPS, camera), sensitive features of the operating system (e.g. contacts), and access to certain exposed parts of other applications. For example, the permission "android.permission.INTERNET" requests the right to access the Internet, and "android.permission.READ_CONTACTS" requests the right to access the users phone contacts database [13].

Once we have extracted the list of requested permissions, we divide them into two groups: standard built-in permissions and non-standard permissions. For standard permissions, we generate a binary vector where each entry corresponds to a built-in permission which is set to 1 if the application requests that permission, and 0 otherwise.

For non-standard permissions, we split the strings into three segments: the prefix (usually "com" or "org"), the organization and product section, and the permission name. We ignore any occurance of the words "android" or "permission," which are ubiquitous.

B. Control Flow Graphs

For every method in a given application, we extract a control flow graph (CFG) from the raw bytecode of the method. A CFG is an abstract representation of a program in which vertices represent atomic blocks of non-jump instructions, and edges represent the possible paths of program flow. Each vertex is labelled based on the last instruction in the block, because it is this instruction that determines how the program flow leaves the block. For example, an unconditional jump is represented as a single edge; a conditional jump is represented as two edges incident on the same starting node. However, many viruses use *metamorphic* techniques to produce modified copies with code that has the same semantics, but with different control flow [33]. To combat these mutations, we apply the graph rewriting rules of [3]:

- Merge consecutive instruction blocks
- Merge unconditional jumps with the instruction block they jump to
- Merge consecutive conditional jumps

These rewrites, or *reductions*, have the added advantage of reducing the size of the extracted graph without destroying important semantic information about program flow.

Once we have extracted and processed these graphs, we discard those graphs with five or fewer nodes, as there are only relatively few possible CFGs of five or fewer nodes (and so they encode little semantic information), and discarding them allows for a sizeable speedup in processing. We then combine these graphs into a large disconnected graph (sometimes such a graph is called a *forest*).

V. Kernels

This section is devoted to a description of the kernels used in our approach. For each kernel we use, we wish to guarantee a property called the *Mercer condition*, or positive-definiteness. To improve readability, an overview of the theory of Mercer kernels, as well as the proof of Theorem 5.2 and a more efficient formulation of the string kernel is given in an appendix.

A. A Kernel over Binary Vectors

Given a feature space of binary vectors (i.e. $\{0,1\}^n$ for some n), there are a number of possible kernels, as discussed in [23]. In particular, the standard boolean operations of conjunction, negated disjunction and if-and-only-if lend themselves to straightforward Mercer kernels. We use the the if-and-only-if kernel:

Definition. The *if-and-only-if kernel*, $k_{\leftrightarrow}$ of two binary vectors is given by

$$k_{\leftrightarrow}(A, B) = \sum_{i=1}^{n} a_i \leftrightarrow b_i, \tag{4}$$

where $x \leftrightarrow y$ is one if x and y have the same value. That is, the if-and-only-if kernel returns the number of equivalent bits.

Theorem 5.1 ([23]): $k_{\leftrightarrow}$ is a Mercer kernel.

B. A Kernel over Strings

For our string features (Section IV-A), we use the kernel in [20], which is defined over arbitrary-length strings over some alphabet Σ. The basic idea is to count the number of common subsequences between strings, weighted by the length of the subsequences in the original string.

Definition. A *string* s is an ordered list of symbols taken from some *alphabet* Σ. We denote by Σ^n the set of all strings of length n, and by Σ^* the set of all strings (of any length). The concatenation of two strings s and t is denoted $s \circ t$. A *substring* t of s, starting at s_i and ending at s_j is denoted by $s[i : j]$.

Given a string s, a *subsequence* u is any string such that $u_j = s_{i_j}$, where $1 \leq i_j < i_{j+1} \leq |s|$. We refer to these indices as $\vec{\imath} = \{i_1, \dots, i_{|u|}\}$, and denote $u = s[\vec{\imath}]$. We denote the length of the subsequence as $l(\vec{\imath}) = i_{|u|} - i_1 + 1$; that is, the length is measured as the distance between the start and end of the subsequence in s, rather than the number of symbols in u.

Let the n^{th} subsequence kernel be defined as

$$k_n(s,t) = \sum_{u \in \Sigma^n} \sum_{\vec{\imath}:u=s[\vec{\imath}]} \sum_{\vec{\jmath}:u=t[\vec{\jmath}]} \lambda^{l(\vec{\imath})+l(\vec{\jmath})}$$

Then, $k_n(s,t)$ measures the number of common n-symbol subsequences between s and t, weighted by their lengths. If $s = t$, then they will have *every* subsequence of length n in common, thus maximizing $k_n(s,t)$ for all $n \leq \min(|s|, |t|)$. On the other hand, if s and t are maximally different (i.e. they have no symbols in common), then $k_n(s,t) = 0$ for all $n \leq N$. We can combine the $k_n(\cdot,\cdot)$ for different n, with a simple sum that is weighted by n:

$$k_{str}(s,t) = \sum_{n=0}^{N} \mu(n) k_n(s,t)$$

For simplicity, we use a simple weighting function $\mu(n) = 1$ for all $n \leq N$.

Theorem 5.2: $k_{str}(\cdot,\cdot)$ is a Mercer kernel.

C. A Kernel over Graphs

Given a feature space of labelled, directed graphs, we use the Weisfeiler-Lehman subtree kernel from [31], which is based on the Weisfeiler-Lehman test for graph isomorphism [35]. In [31], Shervashidze *et al.* propose a general framework of graph kernels where the Weisfeiler-Lehman graph isomorphism test is used to construct a series of labelled graphs, which are each compared using some other graph kernel. This approach allows for very simple (and otherwise not very useful) graph kernels to be applied in a way that improves their discriminative power.

Definition. Given a labelled graph $G = (V, E, l_0)$, where $l_0(v)$, $v \in V$ is a label taken from some alphabet Σ, the *Weisfeiler-Lehman sequence of height h* is the sequence of graphs

$$\{G_0, G_1, \dots, G_h\} = \{(V, E, l_0), (V, E, l_1), \dots, (V, E, l_h)\},$$

where $l_1, \dots, l_h$ are constructed using Algorithm 1. The function $f(\cdot)$ in the algorithm is some mapping $f : \Sigma^* \rightarrow \Sigma$ such that $f(s_i(v)) = f(s_i(u))$ if and only if $s_i(v) = s_i(u)$. The compression function $f(\cdot)$ is intended to make the algorithm more space efficient, so that labels do not grow exponentially with i, with the authors of [31] suggesting maintaining a global counter that records the number of distinct strings that $f(\cdot)$ has been called on. Because our implementation is designed to be distributed, this approach is infeasible. We instead take $f(\cdot)$ to be the identity function.

Algorithm 1 Weisfeiler-Lehman relabelling

```
for $i \leftarrow 0$ to $h$ do
    for all $v \in V$ do
        if $i = 0$ then
            $M_i(v) \leftarrow l_0(v)$
        else
            $M_i(v) \leftarrow \{l_{i-1}(u) \mid u \in \mathcal{N}(v)\}$
                        ▷ $\mathcal{N}(v) = \{u \mid (v,u) \in E\}$ is the set
                        ▷ of *neighbors* of $v$.
            $s_i(v) \leftarrow l_{i-1}(v) \circ sort(M_i(v))$
                        ▷ Sort $M_i(v)$ and concatenate its
                        ▷ elements to the previous label
            $l_i(v) \leftarrow f(s_i(v))$
        end if
    end for
end for
```

Definition. Given a graph kernel $k_1(\cdot,\cdot)$, the *Weisfeiler-Lehman kernel* with base kernel $k_1(\cdot,\cdot)$ and h iterations is defined by

$$k^{(h)}_{WL(k_1)}(G, G') = k_1(G_0, G'_0) + \cdots + k_1(G_h, G'_h), \quad (5)$$

where $\{G_0, \dots, G_h\}$ and $\{G'_0, \dots, G'_h\}$ are the Weisfeiler-Lehman sequences for G and G'.

Theorem 5.3 ([31], Theorem 3): If $k_1(\cdot,\cdot)$ is a Mercer kernel over graphs, then $k^{(h)}_{WL(k_1)}(\cdot,\cdot)$ is also a Mercer kernel.

Definition. The *Weisfeiler-Lehman subtree kernel* with h iterations is a Weisfeiler-Lehman kernel with a base kernel that counts the number of matching node labels between G and G':

$$k_g(G, G') = \sum_{v \in V} \sum_{v' \in V'} \delta(l(v), l(v')), \quad (6)$$

where $\delta(a, b)$ is the Dirac kernel, which is 1 when $a = b$, and 0 otherwise, and $l(v)$ is the label of node v.

D. A Kernel over Sets

Consider a feature space of unordered sets of arbitrary cardinality with elements drawn from some set X for which we have a Mercer kernel $k_0(\cdot,\cdot)$; that is, features $x \in \mathcal{P}(X)$ for some X, together with $k_0 : X \times X \rightarrow \mathbb{R}$. We wish to construct a Mercer kernel $k : \mathcal{P}(X) \times \mathcal{P}(X) \rightarrow \mathbb{R}$ from $k_0(\cdot,\cdot)$.

Ideally, to find $k(x,y)$, we should find the best *matching* of x and y, i.e. a collection of pairs (x_i, y_j) such that $k_0(x_i, y_j)$ is maximal. In [34], a simple method was proposed:

$$k_{match}(x,y) = \frac{1}{2}\frac{1}{|x|}\sum_{i=1}^{|x|}\max_{j} k_0(x_i, y_j) + \frac{1}{2}\frac{1}{|y|}\sum_{j=1}^{|y|}\max_{i} k_0(x_i, y_j). \tag{7}$$

The idea is to find the best match $y_j \in y$ for each $x_i \in x$, and then the best x_i for each y_i, and to sum the $k_0(x_i, y_j)$ for each of these matches. Unfortunately, (7) has been shown not to be a Mercer kernel. In particular, the use of the $\max$ operation does not preserve positive definiteness. As an alternative, we use the exponent match kernel of [21]:

$$k_{set}^{k_0}(x,y) = \frac{1}{|x|}\frac{1}{|y|}\sum_{i=1}^{|x|}\sum_{j=1}^{|y|} k_0(x_i, y_j)^p. \tag{8}$$

By raising the $k_0(x_i, y_j)$ to an exponent, we effectively give larger values more weight. Indeed $\lim_{p\to\infty} k(\cdot,\cdot) = k_{match}(\cdot,\cdot)$. As a sum of exponents of Mercer kernels, $k(\cdot,\cdot)$ is itself a Mercer kernel.

E. A Kernel over Non-standard Permissions

In Section IV-A, we noted that we separate the non-standard permission strings into three components: a prefix, a middle section that contains the organization and/or product names, and a suffix that contains the actual permission name. We treat these three sections as independent sets of strings, and apply the set kernel of Section V-D, and then we take the average of these three values as our kernel value (note that the average of a set of Mercer kernels is a positive linear combination of those kernels, and is therefore a Mercer kernel itself):

$$k_p(x,y) = \frac{k_{set}^{k_{str}}(x_0, y_0) + k_{set}^{k_{str}}(x_1, y_1) + k_{set}^{k_{str}}(x_2, y_2)}{3}, \tag{9}$$

where x_0 is the set of prefixes, x_1 is the set of middle strings, and x_2 is the set of permission names associated with the permissions of x.

F. A Kernel over Applications

An analysis of each of the preceding kernels reveals that they tend to be biased towards longer or larger inputs: for example, the maximum value of $k_{\leftrightarrow}(\cdot,\cdot)$ is n, the length of the vectors being compared. Similarly, the string and graph kernels both perform sums dependent on the size on the input. Therefore, we should normalize them:

$$\widehat{k}(s,t) = \frac{k(s,t)}{\sqrt{k(s,s)k(t,t)}}$$

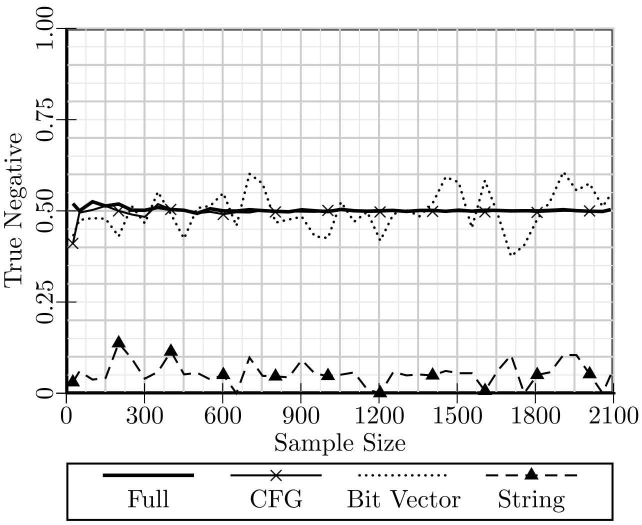

Fig. 1. True Negative vs Sample Size

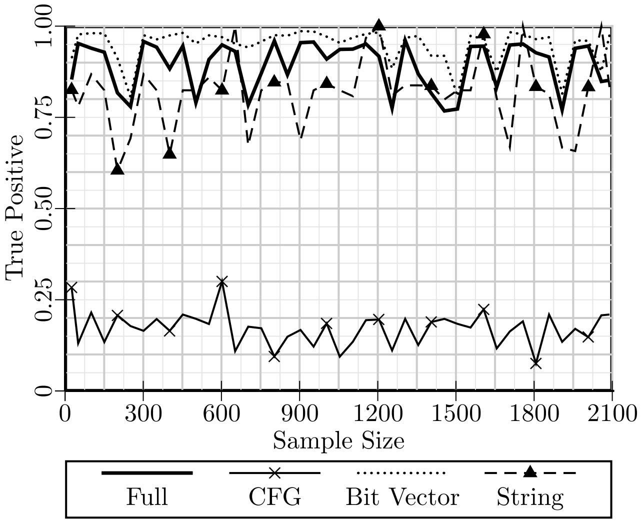

Fig. 2. True Positive vs Sample Size

Then, we have the final kernel, $k : X \times X \to \mathbb{R}$, where X is the space of possible Android applications. Let v_x, p_x and g_x be the extracted permissions bit-vector (Section IV-A), permissions string set array (Section IV-A), and control flow graph (Section IV-B) of the application x, respectively. Then, we have

$$k(x,y) = \widehat{k}_{\leftrightarrow}(v_x, v_y) + \widehat{k}_p(p_x, p_y) + \widehat{k}_g(g_x, g_y), \tag{10}$$

which is a positve linear combination of Mercer kernels, and therefore a Mercer kernel.

VI. Experimental Results

We tested our system against a collection of 2081 benign and 91 malicious Android applications. For each datapoint, we selected a random subset of the training (benign) applications and performed k-fold cross validation. We did this four times per datapoint, and average the results. In addition to the full kernel, we also trained against each individual kernel separately. Figures 1 and 2 show the true negative (benign

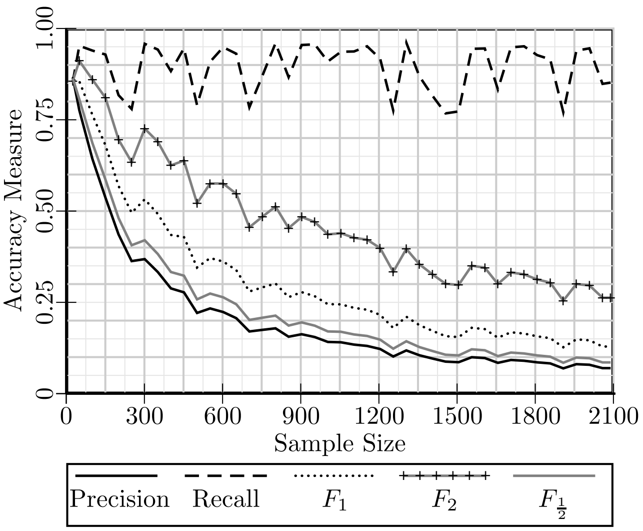

Fig. 3. Accuracy Measures vs Sample Size

classified as benign) and true positive (malware classified as malware) versus sample size, respectively.

Figure 3 shows various measures of accuracy of the system: precision, recall, F_1 measure (the harmonic mean of precision and recall), F_2 measure (like F_1, but with recall weighted twice as much as precision), and $F_{\frac{1}{2}}$ (like F_1, but with precision weighted twice as much as recall). Note that the increasing benign sample size combined with a fixed malicious sample size causes precision to decrease as the benign sample increases (and therefore also the F-measures).

These data suggest some interesting traits of our system. First, the full kernel never outperforms the bit-vector kernel that compares built-in permissions. The bit-vector kernel seems to divide the feature space into two segments: applications that request the kind of permissions that are required by malware, and those that do not. Although it correctly classifies most malware, approximately half of all benign applications request those same permissions, and are therefore classified as malware by the bit-vector kernel and frequently also the full kernel.

Second, the string kernel seems to have very little discriminative power, and generally classifies nearly everything as malware. This is likely because the non-standard permissions that are its input are, by definition, uncommon. This leads to a feature space that is extremely sparse, and often zero, so that much of the training data is at the origin, making it impossible for the 1C-SVM to separate the training data from the origin.

Finally, the graph kernel shows somewhat unusual behavior: it has a higher false negative rate than true negative rate. This suggests that the malware are further from the origin than a significant proportion of the training data, putting them on the wrong side of the separating hyperplane trained by the 1C-SVM.

VII. Discussion and Future Work

We have presented a novel machine learning-based malware detection system for the Android operating system. Our system has shown promising results in that it has a very low false negative rate, but also much room for improvement in its high false positive rate. There are a number of possible improvements that could be investigated.

A. Features

Our system is limited to just the permissions (built-in and non-standard), and CFGs of the input applications. There are many other potential sources of information-rich features. For instance, there are other metadata entries in the manifest file that contains the requested permissions. There are also many potential sources of features in the program code itself, such as constant declarations and method names.

Additionally, the current features could be improved. In particular, the way we extract CFGs abandons much of the information originally present in the code: we label nodes in the CFG based only on the last instruction of the block it represents. We could instead label based on all of the instructions in the block. We also only have a small set of labels, which could be expanded to include more detailed information about the kinds of instructions present (e.g. arithmetic operations, memory access, etc.). Such distinctions would lead to a much more robust label set, which would possibly increase the power of the graph kernel, since it is based on the graph labels.

B. Kernels

In addition to new features, and any new kernels such features would require, there may be room for improvement among the kernels we chose for our existing features. Once again, the CFG feature is the most likely candidate for improvement. The current system treats the collection of graphs extracted as a single disconnected graph. This treatment may give up discriminitive power by comparing these graphs in aggregate, rather than an all-pairs comparison. Therefore, combining the graph kernel of [31] with a set kernel such as those in [21] or even [14] could lead to a much more powerful kernel. Unfortunately, the high number of methods in each application make the quadratic time complexity of the kernel presented in [21] impractical, and the kernel in [14] is over a feature space of sets of vectors, each drawn from the same vector space. Because the graph kernel of [31] essentially converts graphs to vectors, it seems like a good match with [14]. However, two graphs will be converted to vectors drawn from two different vector spaces, and projecting these vectors into the union or intersection space efficiently has proved to be non-trivial.

C. Models

Finally, we use the One-Class Support Vector Machine because we have far more benign examples than malicious ones. However, we do have *some* malicious examples, so a semi-supervised approach may be more powerful. For instance, in [2], the authors reduce the problem of novelty detection in a semi-supervised setting to the problem of Neyman-Pearson Classification [29]. This could be combined with [9], in which the authors describe a method for training Support Vector Machines for Neyman-Pearson Classification. Such a semi-supervised classifier would likely have more discriminitive power than the 1C-SVM.

Appendix

As mentioned in Section III, we wish to solve the constrained quadratic optimization problem (1). Towards that end, (1) is transformed into a Lagrangian [27]:

$$\begin{aligned} L(w,\vec{\xi},\rho,\vec{\alpha},\vec{\beta}) &= \frac{1}{2}\|w\|^2 + \frac{1}{\nu l}\sum_i \xi_i - \rho \qquad (11) \\ &\quad - \sum_i \alpha_i \big((w \cdot \phi(x_i)) - \rho + \xi_i \big) - \sum_i \beta_i \xi_i \ , \end{aligned}$$

which gives a new form for (2):

$$f(x) = \operatorname{sign}\left(\sum_i \alpha_i k(x_i, x) - \rho \right) \ . \qquad (12)$$

From this, we can derive a dual objective function [27],

$$\min_{\vec{\alpha}} \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j k(x_i, x_j) \qquad (13)$$

$$\text{such that } 0 \le \alpha_i \le \frac{1}{\nu l}, \sum_i \alpha_i = 1 \ .$$

However, in order to guarantee that the dual function (13) has a solution, we must place certain restrictions on $k(\cdot,\cdot)$. In particular, if the kernel is not a Mercer kernel (defined below), there may be x_i such that the dual objective function can become arbitrarily large, so that the constrained quadratic optimization problem has no solution [5].

A. Mercer Kernels

Definition. A function $k : X \times X \to \mathbb{R}$ is called a *(real-valued) positive-definite kernel* if it is symmetric ($k(x,y) = k(y,x)$), and

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_i c_j k(x_i, x_j) \ge 0 \qquad (14)$$

for any $n \in \mathbb{N}$, $x_i \in X$ and $c_i \in \mathbb{R}$. Another way of expressing this requirement is that the matrix $\mathbf{K}$ defined by $\mathbf{K}_{ij} = k(x_i, x_j)$ is symmetric, and all of its eigenvalues are non-negative. Note that if $k(x_i, x_j) \ge 0$ for all $x_i, x_j \in X$ and is symmetric, (14) holds.

The theory of positive-definite kernels is based largely on the work of Mercer [22], so the requirement that kernels be positive-definite is often called the *Mercer condition*, and such kernels are referred to as *Mercer kernels*.

It is worth noting a few key properties regarding the combination of Mercer kernels:

Definition. A set X is called a *cone* if $\{\lambda x | x \in X\} X \subseteq X$ for all $\lambda \ge 0$. A set is called *convex* if, for every $(x,y) \in X$, the line segment connecting x and y, $\overrightarrow{xy} \subseteq X$.

Theorem A.1 ([1], 1.11): If X is a nonempty set, the set of all Mercer kernels over $X \times X$ is a convex cone.

Corollary A.2: If $k_1, k_2 : X \times X \to \mathbb{R}$ are Mercer kernels, then $k_+(x,y) = \lambda_1 k_1(x,y) + \lambda_2 k_2(x,y)$ is also a Mercer kernel. In other words, the set of Mercer kernels over $X \times X$ is closed under positive linear combination.

Proof: Consider $k_\rho(x,y) = \rho k_1(x,y) + (1-\rho) k_2(x,y)$ for some $\rho \in [0,1]$. Because the set of Mercer kernels over $X \times X$ is convex, $k_\rho(x,y)$ is a Mercer kernel. Then, consider the kernels

$$k_{\lambda_1}(x,y) = \frac{\lambda_1}{\rho} k_1(x,y) \ \text{ and } \ k_{\lambda_2}(x,y) = \frac{\lambda_2}{1-\rho} k_2(x,y).$$

Because the set of Mercer kernels over $X \times X$ is a cone, both of these are Mercer kernels. Combining the two, we get

$$\begin{aligned} k_*(x,y) &= \rho \frac{\lambda_1}{\rho} k_1(x,y) + (1-\rho) \frac{\lambda_2}{1-\rho} k_2(x,y) \\ &= \lambda_1 k_1(x,y) + \lambda_2 k_2(x,y). \end{aligned}$$

□

Theorem A.3 ([1], 1.12): If $k_1, k_2 : X \times X \to \mathbb{R}$ are Mercer kernels, then $k_\times(x,y) = k_1(x,y) k_2(x,y)$ is a Mercer kernel.

B. More on $k_{str}(\cdot,\cdot)$

The definition of $k_n(\cdot,\cdot)$ lends itself to a naïve algorithm that takes $O(|\Sigma|^n)$ time, which is clearly infeasible. By constructing a recursive definition of $k_n(\cdot,\cdot)$, we can reduce the time to $O(n|s||t|)$.

Definition. The recursive version of $k_n(\cdot,\cdot)$ uses a dynamic programming approach that introduces a number of intermediate computations, k'_i and k''_i for $1 \le i < n$:

$$\begin{aligned} k'_0(s,t) &= 1 \text{ for all } s,t \\ k''_i(s,t) &= 0 \text{ if } \min(|s|,|t|) < i \\ k'_i(s,t) &= 0 \text{ if } \min(|s|,|t|) < i \\ k_i(s,t) &= 0 \text{ if } \min(|s|,|t|) < i \\ k''_i(s \circ x, t \circ y) &= \lambda k''_i(s \circ x, t) \text{ if } x \ne y \\ k''_i(s \circ x, t \circ x) &= \lambda \left(k''_i(s \circ x, t) + \lambda k'_{i-1}(s,t) \right) \\ k'_i(s \circ x, t) &= \lambda k'_i(s,t) + k''_i(s \circ x, t) \end{aligned}$$

then, the final row:

$$k_n(s \circ x, t) = k_n(s,t) + \sum_{j : t_j = x} k'_{n-1}(s, t[1:j-1]) \lambda^2$$

Proof of Theorem 5.2: Consider the feature space transformation for each $u \in \Sigma^n$

$$\phi_u(s) = \sum_{\vec{\imath} : u = s[\vec{\imath}]} \lambda^{l(\vec{\imath})}$$

which maps each string to the weighted length of all occurences of the subsequence u. Then, because this feature space is simply real numbers, we can define its inner product naturally:

$$\begin{aligned} \langle \phi_u(s), \phi_u(t) \rangle &= \sum_{\vec{\imath} : u = s[\vec{\imath}]} \lambda^{l(\vec{\imath})} \sum_{\vec{\jmath} : u = t[\vec{\jmath}]} \lambda^{l(\vec{\jmath})} \\ &= \sum_{\vec{\imath} : u = s[\vec{\imath}]} \sum_{\vec{\jmath} : u = t[\vec{\jmath}]} \lambda^{l(\vec{\imath}) + l(\vec{\jmath})} \end{aligned}$$

Then, we can define a full feature space consisting of the cartesian product over all u. This gives us the inner product

$$\begin{aligned}\langle\phi(s),\phi(t)\rangle &= \sum_{u\in\Sigma^n}\langle\phi_u(s),\phi_u(t)\rangle \\ &= \sum_{u\in\Sigma^n}\sum_{\vec{\imath}:u=s[\vec{\imath}]}\sum_{\vec{\jmath}:u=t[\vec{\jmath}]}\lambda^{l(\vec{\imath})+l(\vec{\jmath})} \\ &= k_n(s,t)\end{aligned}$$

So $k_n(\cdot,\cdot)$ is an inner product in $\mathbb{R}^{|\Sigma^n|}$, so it is a Mercer kernel. $k_{str}(\cdot,\cdot)$ is therefore a positive-weighted finite sum of Mercer kernels, and thus a Mercer kernel itself. □

Acknowledgment

This work started as a senior project together with Nate Diamond, David Evans and Keith Ripley.

References

[1] Christian Berg, Jens Peter Reus Christensen, and Paul Ressel. *Harmonic Analysis on Semigroups*. Springer-Verlag, 1984.

[2] Gilles Blanchard, Gyemin Lee, and Clayton Scott. Semi-supervised novelty detection. *Journal of Machine Learning Research*, 11, 2010.

[3] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Architecture of a morphological malware detector. *Journal in Computer Virology*, 5(3):263–270, 2009.

[4] Abhijit Bose, Xin Hu, Kang G. Shin, and Taejoon Park. Behavioral detection of malware on mobile handsets. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08, pages 225–238, 2008.

[5] Christopher J.C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–167, 1998.

[6] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 15–26, 2011.

[7] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[8] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[9] Mark A. Davenport, Richard G. Baraniuk, and Clayton D. Scott. Tuning support vector machines for minimax and neyman-pearson classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(10), 2010.

[10] Anthony Desnos. androguard. https://code.google.com/p/androguard/.

[11] Bryan Dixon, Yifei Jiang, Abhishek Jaiantilal, and Shivakant Mishra. Location based power analysis to detect malicious code in smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 27–32, 2011.

[12] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[13] Google. Manifest.permission — android developers. http://developer.android.com/reference/android/Manifest.permission.html.

[14] Kristen Grauman and Trevor Darrell. The pyramid match kernel: Efficient learning with sets of features. *Journal of Machine Learning Research*, 8, 2007.

[15] Jeffrey O. Kephart, Gregory B. Sorkin, William C. Arnold, David M. Chess, Gerald J. Tesauro, and Steve R. White. Biologically inspired defenses against computer viruses. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 1*, IJCAI'95, pages 985–996, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[16] Hahnsang Kim, Joshua Smith, and Kang G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08, pages 239–252, 2008.

[17] J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, December 2006.

[18] Lei Liu, Guanhua Yan, Xinwen Zhang, and Songqing Chen. Virusmeter: Preventing your cellphone from spies. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 244–264, 2009.

[19] Raymond W. Lo, Karl N. Levitt, and Ronald A. Olsson. Mcf: a malicious code filter. *Computers & Security*, 14(6):541 – 566, 1995.

[20] Huma Lodhi, Craig Saunders, John Shawe-Taylor, Nello Cristianini, and Chris Watkins. Text classification using string kernels. *Journal of Machine Learning Research*, 2, 2002.

[21] Siwei Lyu. Mercer kernels for object recognition with local features. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2005.

[22] J. Mercer. Functions of positive and negative type, and their connection with the theory of integral equations. *Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Mathematical or Physical Character*, 209:415 – 446, 1909.

[23] Francesca Odone, Annalisa Barla, and Alessandro Verri. Building kernels from binary strings for image matching. *IEEE Transactions on Image Processing*, 14(2), 2005.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Duchesnay E. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[25] Christy Pettey and Holly Stevens. Gartner says 428 million mobile communication devices sold worldwide in first quarter 2011, a 19 percent increase year-on-year. http://www.gartner.com/it/page.jsp?id=1689814.

[26] Christy Pettey and Holly Stevens. Gartner says sales of mobile devices grew 5.6 percent in third quarter of 2011; smartphone sales increased 42 percent. http://www.gartner.com/it/page.jsp?id=1848514.

[27] Bernhard Schölkopf, John C. Platt, John Shawe-Taylor, Alex J. Smola, and Robert C. Williamson. Estimating the support of a high-dimensional distribution. Technical Report MSR-TR-99-87, Microsoft Research, 2000.

[28] Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 38–, Washington, DC, USA, 2001. IEEE Computer Society.

[29] Clayton Scott and Robert Nowak. A neyman-pearson approach to statistical learning. *IEEE Transactions on Information Theory*, 51(11), 2005.

[30] Lookout Mobile Security. Security alert: Droiddream malware found in official android market. http://blog.mylookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/.

[31] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12, 2011.

[32] G.J. Tesauro, J.O. Kephart, and G.B. Sorkin. Neural networks for computer virus recognition. *IEEE Expert*, 11(4):5 –6, aug 1996.

[33] A. Walenstein, R. Mathur, M.R. Chouchane, and A. Lakhotia. Normalizing metamorphic malware using term rewriting. In *Source Code Analysis and Manipulation, 2006. SCAM '06. Sixth IEEE International Workshop on*, pages 75–84, 2006.

[34] Christian Wallraven, Barbara Caputo, and Arnulf Graf. Recognition with local features: the kernel recipe. In *IEEE International Conference on Computer Vision (ICCV)*, pages 257–264, 2003.

[35] B. Weisfeiler and A. A. Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 9, 1968.

[36] Liang Xie, Xinwen Zhang, Jean-Pierre Seifert, and Sencun Zhu. pbmds: a behavior-based malware detection system for cellphone devices. In *Proceedings of the third ACM conference on Wireless network security*, WiSec '10, pages 37–48, 2010.

[37] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proceedings of the 19th Network and Distributed System Security Symposium*, 2012.