Anomaly-based Mobile Malware Detection: System Calls as Source for Features

Dominik Teubert, Fred Grossmann and Ulrike Meyer
Department of Computer Science, RWTH Aachen University, Aachen, Germany

Keywords: Mobile Malware, Machine Learning, Anomaly Detection, One-class SVMs, Hidden Markov Models.

Abstract: Mobile malware nowadays poses a serious threat to end users of mobile devices. Machine learning techniques have a great potential to automate the detection of mobile malware. However, prior work in this area mostly focused on using classifiers that require training with data from both the benign as well as the malicious class. As a consequence, training these models requires feature extraction from large amounts of mobile malware, a task that becomes increasingly difficult considering the obfuscation and emulator detection capabilities of modern mobile malware. In this paper we propose the use of one-class classifiers. The advantage of using these models is that they are exclusively trained with data from the benign class. In particular, we compare generative as well as discriminative modeling approaches, namely Hidden Markov Models and one-class Support Vector Machines. We use system calls as source for our features and compare the discriminatory power of binary feature vectors, frequency vectors, as well as temporally ordered sequences of system calls.

1 INTRODUCTION

In recent years, the popularity of mobile devices such as smartphones and tablets has continuously increased. According to market research company IDC, nearly 1.3 billion smartphones were shipped in 2014 (IDC, 2014). Gartner Inc. predicts that by 2018 more than 50 percent of users will turn to their mobile devices first for all online activities (Gartner, 2014). However, the broad range of capabilities of today's mobile devices, as well as their unique suitability for, e.g., mobile payment or premium rate services, make mobile devices an attractive target for malware authors. The accumulation of large amounts of potentially sensitive data on the devices also contributes to the fact that mobile malware recently emerged from a theoretical danger to a real threat for end users and mobile operators.

Commercial mobile malware detection systems to date mainly use a signature-based approach, which requires the vendors to continuously obtain and analyze new malware samples in order to generate up-to-date signatures. This signature generation process is time-consuming and costly—particularly in face of the rapidly increasing number of new malware samples. Consistently, the detection rates reported in a recent evaluation of four of these mobile malware detection systems are rather low, ranging from 20.2% to 79.6% (Zhou and Jiang, 2012). Another disadvantage of the signature-based approach in the mobile context is that the scanning process has a negative impact on battery lifetime and conflicts with user demands on system responsiveness.

Alternative approaches to mobile malware detection (e.g., (Dini et al., 2012; Burguera et al., 2011; Shabtai et al., 2012; Bose et al., 2008)) make use of machine learning techniques in order to automate malware detection based on monitoring characteristic and measurable properties of a single app or the operating system and using them as *features* for classification. Most of the prior approaches in this area, however, make use of multi-class classifiers that try to discriminate between malicious and benign system behavior. That is, these approaches train a classifier with correctly annotated feature vectors from the benign as well as the malicious class and thus vitally depend on finding and analyzing malware samples for training purposes. While various features have been proposed in this context, a particularly promising approach (also known from host-based intrusion detection (Forrest et al., 1996; Warrender et al., 1999; Yeung and Ding, 2003; Mutz et al., 2006; Maggi et al., 2010)) is using system call traces as source of features: Since system calls provide the interface between user-space and kernel-space, all malicious actions that may harm the system have to pass this inter-

Teubert, D., Grossmann, F. and Meyer, U.
Anomaly-based Mobile Malware Detection: System Calls as Source for Features.
DOI: 10.5220/0005652900260036
In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016)*, pages 26-36
ISBN: 978-989-758-167-0

face. To date two approaches exist that make use of system call features in the mobile context (Burguera et al., 2011; Dini et al., 2012). However, these approaches focus on frequencies of system calls only and do not consider other options to derive features from system call traces.

In contrast to prior work, in this paper we propose the use of one-class classifiers, i.e., models that exclusively rely on data from one class only (in our case the benign class) in the training phase. Specifically, we compare generative as well as discriminative modeling approaches, namely Hidden Markov Models (HMMs) and one-class Support Vector Machines (SVMs) as two state-of-the-art models of this type. The obvious advantage of such models is that they allow model training without access to data from the malicious class. Most importantly, however, they can be expected to be able to also detect zero-day malware that heavily differs from known malware as they define anomalies as deviations from benign behavior rather than similarity to known malicious patterns. We concentrate on system calls as source for features. However, instead of considering only frequencies of system calls in observed traces as in (Burguera et al., 2011; Dini et al., 2012), we propose two additional feature types derived from system call traces, namely binary vectors indicating the occurrence of system calls, and (temporally ordered) sequences of system calls. We comparatively evaluate the two modeling methods in combination with the different feature types on pairs of benign apps and their malicious repackaged counter parts. Specifically, we train a model with feature vectors derived from system call traces of a benign app and then test if the model is able to detect feature vectors derived from traces of the app's malicious counterpart.

The remainder of this paper is structured as follows: Section 2 reviews relevant related work. Section 3 describes how features that are used throughout this paper are derived from monitored system call data. Section 4 introduces the models used. Section 5 describes our experimental setup. Section 6 presents the results, and puts them into context.

2 RELATED WORK

The work published so far in the area of mobile malware detection aims to overcome the shortcomings of signature-based approaches, namely the great effort that is necessary to create signatures. Thus, most of the related work on mobile malware detection focuses on finding malicious patterns in an automated fashion by means of machine learning. These approaches significantly vary in the features and models used. In the learning phase, typically a classifier is trained with features derived from observed data. In the subsequent detection phase, the features are derived from the data observed while the mobile device is under regular usage and are tested against the trained model.

Following the two most important ways to analyze (mobile) malware, the approaches can be classified into those that use static analysis to gather the data needed to derive features and those that use dynamic analysis to monitor behavioral features at runtime.

The former line of work includes (Arp et al., 2014),(Zhang et al., 2014), and (Aafer et al., 2013), in which API-level features of varying complexity are derived and used to train classifiers with the goal to distinguish malicious from benign apps. Approaches of this category are not able to cope well with malware that uses obfuscation techniques as these render automated features extraction based on static analysis infeasible in many cases. Approaches from the latter category typically monitor system- or user-behavior-based features. In an early approach, Bose *et al.* use a temporal logic to define behavioral signatures of API calls (Bose et al., 2008). To be able to also match partial signatures at run-time, the authors use SVMs for classification. Xie *et al.* proposed *pBMDS*, a system that correlates user inputs with system call traces (Xie et al., 2010). The authors use HMMs to model theses correlations to detect anomalies that indicate the infection with malware. Shabtai *et al.* proposed an anomaly detection framework called *Andromaly*, which uses a large set of very different run-time features (such as CPU load and number of incoming SMS messages) and compares six different standard classifiers on this data (Shabtai et al., 2012). Contract-based approaches that try to infer policies apps have to meet in a dynamic and collaborative manner also fall into this category (Dini et al., 2014; Aldini et al., 2014).

Recently, the idea using system call traces as source for features, originally proposed in the area of host-based intrusion detection (Forrest et al., 1996; Warrender et al., 1999; Eskin et al., 2002; Yeung and Ding, 2003; Mutz et al., 2006; Maggi et al., 2010), was adapted to mobile malware detection. With *Crowdroid* Burguera *et al.* proposed a solution that collects system call traces of Android apps using a user-space app, i.e., system calls are monitored on a per-app basis (Burguera et al., 2011). The traces are further preprocessed locally into feature vectors representing the frequency of each monitored system call. On a remote server, the feature vectors are then clustered into two classes using the k-means algorithm. The crucial step is then to label these

two classes as benign and malicious. Burguera *et al.* propose to solve this problem by applying a crowdsourcing approach: for each app, feature vectors are collected by the crowd and submitted to the server. After clustering, the larger of the two resulting classes is labeled as benign, following the assumption that benign apps are executed more often than their malicious counter parts. This approach has two major shortcomings: first, it always results in two classes, even if only benign vectors are present and second, the approaches rely on the participation of a sufficient number of honest users in order to guarantee that the assumption holds.

Similar to Burguera *et al.*, Dini *et al.* use frequencies of system calls as features. However, the system calls are not monitored on a per-app basis, but system-wide. Furthermore, only 11 of the most sensitive system calls are taken into account. The resulting features are complemented with two additional features: (1) whether or not the user is idle and (2) the count of sent SMS messages. They use the k-nearest neighbors (KNN) classifier and train it with benign traces generated with the help of real users as well as artificially generated malicious traces. This manual creation and later interpolation of malicious feature vectors is a major drawback of the system, since it requires detailed assumptions on how malicious actions influence the system call counts. Another weakness of the system is the combination of the system call counts and the two additional features into the same feature vector, since it is not clear to which extent the system call counts contribute to the discrimination power of the system. So it is very likely that e.g., for SMS sending malware the classifier learns over-simplistic rules of the form "malware detected if the user is idle and SMS messages are sent". The heavy reliance on the idleness-indicator allows smart malware to bypass detection very easily by becoming active if and only if the user is active as well.

Nearly all of the aforementioned approaches use benign as well as malicious data to train a multi-class classifier in the learning phase (with (Xie et al., 2010) being the only exception). This diminishes the advantages of implementing anomaly detection in its purest form based on a one-class classifier trained on benign data only. In particular, such one-class classifiers do not require the collection and processing of malware samples. In addition, they can be expected to be able to also detect zero-day malware that heavily differs from known malware as they define anomalies as deviations from the model of benign behavior rather than similarity to known malicious patterns. We therefore propose the use of one-class classifiers in this paper and compare two state-of-the-art models of this type. While so far only the use of frequencies of system calls as feature was studied (e.g. (Burguera et al., 2011; Dini et al., 2012)), we compare three different features derived from system calls. Specifically, we evaluate the different combinations of one-class classifiers and feature types with respect to their power to differentiate between a benign app and a repackaged malicious version of the same app. Note that as most other recent work in this area we focus on the market leading Android platform, which is most targeted by mobile malware authors. Most Android malware comes as repackaged malicious version of some benign app (Zhou and Jiang, 2012).

3 FEATURE DERIVATION

While system calls as a source for features were considered in different approaches in behavior-based mobile malware detection (see Section 2), the field to date lacks a systematic comparison of different features derived from sequential system call data. Notably, prior work solely focused on the frequency feature, neglecting to investigate other system call derived features of varying complexity.

In this paper we study the following three types of increasingly complex features:

- **Binary Feature:** The *binary feature* is the most straightforward way to process system call data. It indicates whether or not a specific system call was invoked. A binary feature vector therefore is a bitvector of the form $x_{bin} \in \{0,1\}^n$, where n is the number of system calls available on the operating system.
- **Frequency Feature:** The *frequency feature* has a structure similar to that of the binary feature. However, instead of a bit indicating the presence of a system call the vector now consists of integers indicating the number of occurrences for each call. This yields a vector of the form $x_{frq} \in \mathbb{N}^n$.
- **Sequence Feature:** The most complex feature we use in this paper is the *sequence feature*. Unlike the other two types of feature vectors, sequence vectors capture the temporal ordering of system calls. The use of sequence vectors requires some preprocessing, since models typically require sequences to have equal length. We use the standard sliding window approach to solve this problem: A window of fixed size is moved over the sequence with a specific step-size to generate sequences of fixed length. Each sequence vector therefore has the form $x_{seq} \in S^k$, where S denotes the set of system calls and k the size of the sliding window.

4 MODELING

In the following we briefly describe the theoretical foundation of the two state-of-the-art one-class models we use in the rest of this paper, namely HMMs and one-class SVMs. While HMMs are generative, probabilistic models, SVMs are discriminative and non-probabilistic models allowing us to compare the suitability of different algorithm types for the anomaly detection use case. The former is particularly suited to model temporally ordered data as needed for the sequential feature. The one-class SVMs on the other hand, can be used in connection with all three feature types introduced in Section 3.

4.1 Hidden Markov Models

HMMs (Rabiner, 1989) are finite state machines (FSMs) that describe a doubly embedded stochastic process. The first process is discrete and stationary and is modeled with a fixed number of states. Since this process is defined as first-order Markov process, an additional restriction known as the Markov property is introduced. This property refers to the constraint that the process behavior at any time t depends only on the current and the directly preceding state. While this first stochastic process is considered not to be observable (i.e., it is assumed to be hidden), a second stochastic process exists through which means the first process is observable. At any time t this second process generates an output O_t, which is referred to as observation. Note that the observation probability $P(O_t \mid q_t = S_j)$ only depends on the current state q_t and not on any preceding states or observations. Since the internal behavior (in particular the sequence of visited states during generation) of the model is hidden, the generated observation sequence $\mathbf{O} = O_1 O_2 \ldots O_T$ is the only model behavior that is visible from the outside. These observation sequences, which directly map to the data the model is trained with, show that HMMs naturally operate on sequential data.

The complete specification of a first-order HMM λ comprises finding a reasonable value for the number of hidden states N and specifying the alphabet of the observation symbols including its size M. In addition, three probability matrices A, B and π have to be specified. Here, A denotes a matrix of state transition probabilities, B denotes a matrix of observation probabilities and π is a vector of initial state probabilities. Since the number of hidden states N and the size of the alphabet M are implicitly defined by the dimensions of the matrices A and B, $\lambda = (A, B, \pi)$ represents a compact notation of the parameter set of an HMM.

In the context of this paper, two basic problems of HMMs and their efficient algorithmic solutions are of interest. First, the *training problem* covers the question how a model λ can be trained given a set of training sequences $X = \{\mathbf{O}^1, \mathbf{O}^2, \ldots, \mathbf{O}^l\}$, so that $P(X \mid \lambda)$, that is the probability of λ generating X, is maximized. This problem is most commonly solved using the *Baum-Welch algorithm*, an iterative procedure for model parameter estimation from the class of expectation-maximization (EM) algorithms. Second, the *evaluation problem* covers the issue how $P(\mathbf{O} \mid \lambda)$, i.e., the probability that λ generates $\mathbf{O}$, could efficiently be computed given an HMM λ and an observation sequence $\mathbf{O} = O_1 O_2 \ldots O_T$. This latter problem can be solved using the *forward algorithm*, an algorithm that, instead of enumerating every possible state sequence like the naïve approach does, splits the observation sequence into two subsequences, one covering the range from time 1 to t and the second one covering the range from $t+1$ to T. This idea can be encoded into the so-called *forward variable* α and it leads to a recursive definition and thus to an efficient formulation of the calculation.

HMMs are furthermore generative models and support one-class training, so that they can readily be applied to anomaly detection. Since HMMs have their strengths in temporal pattern recognition we use them solely to model the sequence vectors. For this, the sequences are preprocessed with a sliding window of fixed size. The gathered fixed-length sequences are then used to train an HMM with the help of the Baum-Welch algorithm. In the detection phase the monitored sequences are preprocessed in the same way and tested against the model using the forward algorithm. Both training and detection phase are described in more detail in Section 5.

4.2 One-class SVMs

SVMs have gained great popularity in recent years due to their outstanding classification performance in a number of application areas. Besides these empirical results, there are theoretical founded arguments from statistical learning theory for a good generalization performance of SVMs. These theoretical guarantees on how well SVMs generalize to unseen data make it a particular interesting modeling approach, since those predictions are generally rare for learning algorithms. However, the original application area for SVMs are multi-class problems. Due to the advances of Schoelkopf *et al.* (Schölkopf et al., 1999) the advantages of SVMs can also be used if data for only one of two classes is available, as it is the case in anomaly detection where only benign system behav-

ior is observed during training. In contrast to HMMs, SVMs are discriminative and non-probabilistic models.

A classical two-class SVM places a hyperplane between two classes of points that make up the training data. New data points are classified according to which side of the hyperplane they lie on. To achieve a good generalization performance, a SVM maximizes the *margin* between the two different classes. A one-class SVM as described by Schoelkopf *et al.* (Schölkopf et al., 1999) places a separating hyperplane with maximal distance to the origin. This way the data space is partitioned into two subspaces: one holds points that are alike to the training points and the other has points that are very different. If we allow some slack while defining the hyperplane boundary, a few training points may be labeled as outliers.

Training a one-class SVM corresponds to solving a quadratic maximization problem, where the output is a sparse vector of data points that define the decision boundary, the *support vectors*. The form of this decision boundary can be adjusted by tuning the parameters of the SVM. Finding the optimal parameters specific to the problem at hand is referred to as *model selection*. In the one-class SVM formulation the free parameter that influences the placement of the hyperplane is ν. Here, the parameter ν gives an upper bound on the fraction of data points the one-class SVM may designate as outliers during training. Another interpretation for ν is to consider it a lower bound on the fraction of support vectors. Thus, selecting an appropriate ν is a crucial part of model selection.

Kernels. The biggest advantage of (one-class) SVMs, especially when dealing with structured data, is the support of kernels. The main idea of kernels is to map non-linearly separable data into a higher dimensional feature space where the data points are linearly separable with the help of a hyperplane. In our context this allows us to model all three features described in Section 3 with a single modeling approach and to compare them against each other and against HMMs.

The simplest kernel is the *linear kernel*. It computes the similarity metric between two points simply as the standard dot product. Another frequently used kernel is the *Gaussian Radial Basis Function* (RBF) kernel. The corresponding kernel function is of the form: $k(\mathbf{x},\mathbf{x}') = e^{-\frac{||\mathbf{x}-\mathbf{x}'||}{\tau}}$. Because of the additional parameter τ, which can be interpreted as the kernel's width, we have to evaluate an extra dimension in the model selection process if we use the RBF kernel.

Besides these two standard kernels which we use for the binary and frequency vectors, the spectrum kernel introduced by Leslie *et al.* (Leslie et al., 2002) is used in the context of sequence vectors. In contrast to HMMs, the monitored trace is not split into fixed-length sequences in a preprocessing step, but implicitly by the spectrum kernel. Given a trace of system calls, the spectrum kernel extracts all k-mers (sequences of length k) and compares the number of occurrences of these k-mers in the traces via the dot product.

5 EXPERIMENTAL SETUP

In this section we provide details on how we evaluate the different combinations of modeling approaches and features. We start by explaining how we obtain the benign apps used to train our models and the malware samples used for the assessment of the detection performance of the trained models. Note that we train models on a per app basis; i.e. each benign app is used to train an individual model for this one app and the detection performance of this model is then tested with features derived from the benign app as well as its malicious counterpart. I.e., while our models do not require any malicious data for training purposes, determining their detection quality naturally requires malicious data. We then describe how we monitor system call traces of benign and malicious apps. This is followed by the definition of the metrics used to quantify the detection quality of our models. Finally, we describe the evaluation steps for the two modeling approaches in detail.

5.1 Malware Samples and Benign Apps

Throughout the evaluation we will study repackaged malware, i.e., legitimate Android apps that were trojanized with a malicious payload. Specifically, we train a model with features derived from the system call traces of a benign app and then evaluate how well the model is able to differentiate between features derived from traces of the benign app in question and features derived from traces of its malicious counterpart. The decision for this scenario follows two crucial insights: On the one hand repackaged malware has a massive prevalence. According to Zhou and Jiang (Zhou and Jiang, 2012) 86% of the Android malware available at that time were repackaged variants of legitimate applications. Considering only repackaged malware, therefore, does not constitute a major restriction. On the other hand, the case of repackaged malware meets the strengths of anomaly detection, since one can learn a clean model from the

Table 1: Overview over the repackaged apps.

Package Name	MD5	Type	Shortcut
com.appspot.swisscodemonkeys.steam	5895bcd066abf6100a37a25c0c1290a5	fun app	steam
com.bwx.bequick	8b540b320359cc1b842dd800a9e49628	utility	bequick
com.camelgames.mxmotor	e5b7b76bd7154dea167f108daa0488fc	game	mxmotor
com.camelgames.shootu	ffc735a75a6135a282ff4172fff2f371	game	shootu
com.creativem.overkill	5c8c0433e7f0c5f8c686a92b9eb462ca	game	overkill
com.estrongs.android.pop	82f5bf4509e35c3ae7172a0dd7c3ecf4	utility	estrongs
com.forthblue.pool	042177a7d144d5e3264a85341f539cb5	game	forthblue
com.gamelio.DrawSlasher	b87f2f3a927bf967736ed43ca2dbfb60	game	drawslasher
com.googlecode.netsentry	17a2b038d1d9080b42ca8e497dddafb8	utility	netsentry
org.zwanoo.android.speedtest.Speedtest	b18e9a2ab55ec87b2fcac0227e61d20e	utility	speedtest
tencent.qqgame.lord	1bcda430eda6f2606d50f917d485500a	game	qqgame

Table 2: Details on the repackaged apps according to virustotal.com.

App	Malware Fam.	Ratio	First Sub.
steam	Adrd	42 / 55	2011-03-01
bequick	RootSmart	34 / 54	2013-10-28
mxmotor	BaseBridge	38 / 56	2012-06-11
shootu	Dowgin	24 / 54	2014-08-01
overkill	BaseBridge	39 / 54	2012-10-08
estrongs	PjApps	40 / 56	2011-10-23
forthblue	Voxv	27 / 54	2013-03-10
drawslasher	GoldDream	41 / 55	2011-07-06
netsentry	Geinimi	40 / 55	2012-01-04
speedtest	Mseg	28 / 50	2014-03-26
qqgame	BaseBridge	47 / 56	2011-06-05

benign version and can evaluate this model by testing the trace of the malicious version against it. Conversely, this means that a generalization to more complex cases is needless, if the models fail to achieve sufficient detection performance in the simple case of discriminating between benign apps and malware repackaged with the same app.

Note that the effort required to find pairs of a legitimate app and its trojanized counterpart is very high and therefore the overall number of evaluated apps is limited. The difficulty here resides not in finding malicious apps (which we obtained through a cooperation with virustotal.com), but rather in finding the exact benign counterpart for such a trojanized app, i.e., the benign app with *with identical version tag*. As there is no central repository for benign apps that also lists former versions additional effort is required to find and obtain these apps. Fortunately, since we train our classifiers on a per-app basis, the classification performance would not profit from a higher number of apps. To make the evaluation as meaningful as possible despite the limited number of samples, we focused on finding a set of apps that cover a broad range of expected behavior. Overall, we found 11 pairs using this approach, including 9 different malware families and covering a 3 year time span between 03/2011 and 03/2014. In Table 1, we provide an overview of the malware samples, stating the *Package Name* of the sample, its *MD5* hash, the *Type*, and a *Shortcut* introduced for further reference in this paper. Table 2 gives further details about the evaluated samples, where *App* identifies the app with its shortcut, *Malware Family* denotes the malware family the majority of the search engines at virustotal stated, the *Detection Ratio* gives the ratio of search engines at virustotal that detected the sample, and *First Submission* gives the date when the sample was uploaded to virustotal for the first time.

5.2 Data Acquisition

To generate extensive data sets from the found app pairs, we utilized an automated approach using the Android emulator that ships with the Android SDK. The actual system call monitoring is done by an Android app that in turn uses the Linux user-land tool *strace* to attach to all running processes. Since user interaction has to be mocked in our automated approach, we used the monkeyrunner—an SDK tool that is often used to stress test apps in the development process—to inject random touch and keystroke events. With help of this automated setup 50 individual runs of each (benign or malicious) app were monitored. However, due to crashes during monitoring some very short traces had to be excluded from the data set. The resulting *strace* log files were then preprocessed to extract the features described in Section 3 and to format the data for the different modeling approaches.

5.3 Metrics

To quantify the performance of the different modeling approaches and features we utilize the common metrics true positive rate (TPR), false positive rate (FPR), and (balanced) accuracy (ACC). For both of our models the assessment of the TPR, i.e., the proportion of correctly classified malware, is straightforward, since it involves simply the test of all malicious traces of an app against the model trained with all benign traces of an app. We also refer to the TPR as *detection rate*. The calculation is as follows:

$$TPR = \frac{\#\ true\ positives}{\#\ true\ positives + \#\ false\ negatives}$$

Evaluating the FPR requires a validation set that is not included in the training set. We use k-fold cross-validation with $k = 5$ to assess the FPR, i.e., our original data set of benign traces is partitioned into 5 ran-

domly chosen subsets. In every of 5 iterations 4 subsets constitute the training set and one subset is used as validation set, so that every subset is used as validation set exactly once. The FPR can then be calculated as follows:

$$FPR = \frac{\#\ false\ positives}{\#\ false\ positives + \#\ true\ negatives}$$

To summarize the overall performance of our models we use the balanced accuracy, which is defined as the arithmetic mean of the true results, i.e., true positive rate and true negative rate:

$$ACC = \frac{TPR + TNR}{2}$$

5.4 Modeling Approaches

While there are differences between the modeling approaches regarding the preprocessing and the actual training, the scope of the models stays the same. Since we use repackaged apps for evaluation, we are able to create models on a per-app basis, using the benign versions as a base line to detect anomalies. An individual model therefore represents the benign behavior of a single app. Building the evaluation setup on a per-app basis also means that the overall percentage of detectable apps will not significantly change when including more apps, since a broad range of different malware families is included in our test set.

HMM Modeling. The HMM modeling initially uses a sliding window preprocessing to obtain sequences of length 8 from all of the monitored traces. This preprocessing step therefore is the actual derivation of the *sequence vectors* described in Section 3. The obtained sequence vectors from the benign traces are then taken as input for the Baum-Welch algorithm. A crucial issue when training HMMs is that of hidden space cardinality, i.e., the absolute number of hidden states of the model. Since there is no general rule to determine the optimal number of hidden states ((Yeung and Ding, 2003) recommends a number close to the number of used system calls in the traces, what we used as a hint), we optimized this parameter across our 11 repackaged apps which resulted in the use of 76 states.

After the training of an HMM representing the benign behavior of a repackaged app, we are able to test traces against this model. The forward-algorithm is used to calculate the so called production probability, i.e., the probability that the sequence was generated by the model. To finally decide whether a monitored app can be classified as malware or not, two different thresholds are applied. The first threshold determines if a single sequence is suspicious. This threshold is determined by testing a single validation trace, i.e., a benign trace not included in the training set, against the model. The obtained probabilities are then sorted and the minimum of these values is used as the threshold. Note that this first threshold is determined on a per-app basis. The second threshold, however, is static for all apps and is used to ultimately decide if an app is considered malware. For this second threshold we select and evaluate two different static values. The first one optimizes the overall average accuracy and is set to 0.0001, stating that if 1 or more anomalies are found in 10,000 sequences the app is classified as malware. The second one is set to 0.01 and optimizes the overall false positive rate (see Table 6 (b)). Since the determination of the second threshold depends on the availability of malicious traces, it is set static across all apps. The rationale behind this approach is to find a value that is as generic as possible, so that it also can be applied on new apps for which no malicious counterpart is available. In Section 6 we also briefly describe the results that can be achieved when optimizing the second level threshold on a per-app basis (see Table 7).

One-class SVM Modeling. The preprocessing required for the one-class SVM modeling varies depending on the chosen feature representation and the selected kernel. Binary vectors can be used as input for the linear and the RBF kernel directly, whereas frequency vectors have to be normalized first. Normalizing the frequencies to the interval $[0,1]$ eliminates the overweighting of common system calls that occur in benign traces. We divide each system call frequency by the maximal value that was recorded during a benign trace, if this value is different from zero. System calls that do not occur in the benign traces are not normalized. For the spectrum kernel, we map each system call trace to a string, with each character representing one system call. To this end, we reduced the number of system calls that are taken into account to those that occur at least once during monitoring.

After preprocessing, we train the one-class SVM with the data of the benign applications. To conduct the model selection with one-class SVMs, we have to determine the parameter ν. Additional parameters that are used within the specific kernel have to be selected as well. In case of the linear or the sequence kernel we conduct a one-dimensional grid search for ν, whereas for the RBF kernel a two-dimensional grid search is performed to find the additional kernel parameter τ. We sample the search space exponentially, testing the values $0.9^0, \ldots, 0.9^{249}$ for ν for all kernels except when using the RBF kernel. Sampling a two-dimensional grid for training with the RBF kernel re-

Table 3: Repackaged apps evaluated with one-class SVM on the binary feature.

App	ACC	TPR	FPR
steam	83.67	95.74	27.45
bequick	95.62	99.55	22.00
mxmotor	90.22	100.00	18.37
shootu	90.34	96.51	15.56
overkill	84.85	98.00	28.57
estrongs	66.32	100.00	69.57
forthblue	74.32	100.00	54.29
drawslasher	82.47	91.49	26.00
netsentry	94.38	97.22	6.34
speedtest	71.43	90.00	47.92
qqgame	72.90	100.00	54.72
Avg.	82.41	97.14	33.71

(a) RBF kernel

App	ACC	TPR	FPR
steam	98.98	100.00	1.96
bequick	85.40	87.50	24.00
mxmotor	100.00	100.00	0.00
shootu	53.41	5.81	1.11
overkill	88.89	80.00	2.04
estrongs	70.53	61.22	19.57
forthblue	63.51	43.59	14.29
drawslasher	68.04	93.62	56.00
netsentry	99.44	97.22	0.00
speedtest	58.16	22.00	4.17
qqgame	73.83	100.00	52.83
Avg.	78.20	71.91	16.00

(b) Linear kernel

Table 4: Repackaged apps evaluated with one-class SVM on the frequency feature.

App	ACC	TPR	FPR
steam	62.24	68.09	43.14
bequick	95.26	100.00	26.00
mxmotor	63.04	67.44	40.82
shootu	91.48	96.51	13.33
overkill	54.55	88.00	79.59
estrongs	51.58	100.00	100.00
forthblue	59.46	79.49	62.86
drawslasher	59.79	70.21	50.00
netsentry	92.70	97.22	8.45
speedtest	56.12	80.00	68.75
qqgame	53.27	96.30	90.57
Avg.	67.23	85.75	53.05

(a) RBF kernel

App	ACC	TPR	FPR
steam	55.10	72.34	60.78
bequick	98.54	99.55	6.00
mxmotor	54.35	11.63	8.16
shootu	52.27	5.81	3.33
overkill	59.60	34.00	14.29
estrongs	51.58	79.59	78.26
forthblue	56.76	43.59	28.57
drawslasher	56.70	14.89	4.00
netsentry	97.75	97.22	2.11
speedtest	54.08	56.00	47.92
qqgame	74.77	100.00	50.94
Avg.	64.68	55.88	27.67

(b) Linear kernel

quires much more work, so we limit the search space to $0.9^0,\ldots,0.9^{49}$ for ν and $2^3,\ldots,2^{-13}$ for τ. Since we decrement the exponent by 2 each turn for τ, we search 450 parameter combinations when using the RBF kernel while only evaluating 250 parameters on other kernels.

Linear and RBF kernel do not need any preprocessing, we just have to provide the feature vectors. The spectrum kernel on the other hand works on strings. Internally, it breaks down the string into sequences of length 8 using the sliding window technique. Since each system call is represented by a `char` after the preprocessing, we can concatenate 8 system calls to make up a 64-bit word. These 64-bit words are in turn interpreted as unsigned integers to build up a sparse vector. This vector has a dimensionality of 2^{64}, however, due to the sparse representation it suffices to save the actually occurring sequences.

6 EVALUATION RESULTS

In this section we present and compare the results of our evaluation of HMMs and one-class SVMs (see Section 4) in the context of distinguishing malicious from benign Android apps based on features extracted from system calls (see Section 3). We evaluate HMMs on the *sequence feature* only (see Table 6), as HMMs reasonably work on sequential data only. One-class SVMs are tested on different combinations of kernels and features. Specifically, the *binary feature* and the *frequency feature* are evaluated in combination with the linear kernel and the RBF kernel (see Table 3 and Table 4), while the *sequence feature* is evaluated with the spectrum kernel, which naturally works on sequences (see Table 5). In the following we first compare the performance of the individual features in combination with different modeling approaches averaged over all apps. We start with the *binary feature*, continue with the *frequency feature* and finally discuss the *sequence feature*. Then we take a closer look at the results on a per-app basis. Finally, we discuss and compare the overall discriminatory power of the three different feature types proposed.

In Table 3 we summarize the one-class SVM results for the *binary feature*. Here the RBF kernel shows superior results regarding the accuracy, while the linear kernel has a lower detection rate, but also a much lower false positive rate. The comparison of both kernels using the *frequency feature* presented in Table 4 shows a very low average accuracy for both kernels, with only individual apps achieving an acceptable ratio of detection rate and false positive rate.

While the RBF kernel shows slightly better average accuracy than the linear kernel on both the *binary* and the *frequency feature*, it also shows much higher false positive rates in both cases. We reckon that this behavior is due to an overfitting of the RBF kernel to the training data, which is underpinned by the significantly higher support vector ratio of the RBF kernel compared with the linear kernel. Since the RBF kernel also needs more time for training, the linear kernel

Table 5: Repackaged apps evaluated with one-class SVM with spectrum kernel on the sequence feature.

App	ACC	TPR	FPR
steam	56.12	89.36	74.51
bequick	97.81	99.55	10.00
mxmotor	61.96	41.86	20.41
shootu	77.27	87.21	32.22
overkill	55.56	42.00	30.61
estrongs	51.58	100.00	100.00
forthblue	79.73	92.31	34.29
drawslasher	74.23	70.21	22.00
netsentry	96.07	97.22	4.23
speedtest	60.20	34.00	12.50
qqgame	65.42	96.30	66.04
Avg.	70.54	77.27	36.98

(a) Constant window size 8

App	ACC	TPR	FPR
steam	58.16	65.96	49.02
bequick	98.54	99.55	6.00
mxmotor	63.04	39.53	16.33
shootu	78.98	90.70	32.22
overkill	56.57	52.00	38.78
estrongs	55.79	85.71	76.09
forthblue	79.73	92.31	34.29
drawslasher	74.23	70.21	22.00
netsentry	97.19	97.22	2.82
speedtest	63.27	34.00	6.25
qqgame	65.42	96.30	66.04
Avg.	71.90	74.86	31.80

(b) Varying window size optimized on per-app basis

Table 6: HMM evaluated on repackaged apps.

App	ACC	TPR	FPR
steam	99.34	100.00	1.32
bequick	78.33	100.00	43.34
mxmotor	65.78	43.90	12.35
shootu	92.75	95.06	9.57
overkill	82.21	73.47	9.05
estrongs	71.12	51.02	8.78
forthblue	60.06	33.33	13.21
drawslasher	64.44	30.00	1.11
netsentry	89.24	86.21	7.73
speedtest	83.70	78.43	11.03
qqgame	78.59	98.15	40.97
Avg.	78.69	71.78	14.40

(a) Threshold 0.0001, optimizing accuracy

App	ACC	TPR	FPR
steam	100.00	100.00	0.00
bequick	88.58	77.16	0.00
mxmotor	50.89	2.44	0.67
shootu	96.55	95.06	1.96
overkill	57.14	14.29	0.00
estrongs	50.00	0.00	0.00
forthblue	49.85	2.56	2.86
drawslasher	53.75	7.50	0.00
netsentry	91.38	82.76	0.00
speedtest	50.00	0.00	0.00
qqgame	50.00	0.00	0.00
Avg.	67.10	34.71	0.50

(b) Threshold 0.01, optimizing false positive rate

has a superior balance between accuracy, false positive rate, and training performance.

Table 5 shows the evaluation of the *sequence feature* in the one-class SVM case with a fixed window size of 8 in (a) and a varying window size between 2 and 8 that optimizes the accuracy in (b). Overall, the one-class SVMs with spectrum kernel rank behind the one-class SVMs on the *binary feature* and the HMMs with respect to the accuracy. In Table 6 (a) the results for the HMMs on the *sequence feature* are summarized, showing an average accuracy on one level with one-class SVMs with linear kernel on the *binary feature*. Note that HMMs show the lowest false positive rate of all models and that this value can further be optimized as shown in Table 6 (b). This optimization that brings the FPR to a real-world applicable level of below 1%, however, comes to the price of only detecting 36% of the apps fairly reliably. For the sake of completeness we also included the detection performance for a per-app optimized second level threshold in Table 7, showing only a slight improvement compared to the static threshold applied in Table 6 (a).

Turning to the results for individual apps, our evaluation shows that there are apps that show good detection performance across model boundaries (in particular *netsentry*, followed by *bequick*) and that there are apps that show the opposite result, i.e., bad performance across model boundaries (*estrongs*, *drawslasher*, *forthblue*, and *qqgame*). Figure 1 gives an overview over the achieved accuracy for all tested apps and models, where *frq*, *bin*, and *seq* denote

Table 7: HMM evaluated on repackaged apps with per-app optimized second level threshold.

App	ACC	TPR	FPR	Thres.
steam	100.00	100.00	0.00	0.01
bequick	92.97	99.49	13.56	0.0008
mxmotor	65.78	43.90	12.35	0.0005
shootu	96.55	95.06	1.96	0.01
overkill	82.21	73.47	9.05	0.0001
estrongs	71.12	51.02	8.78	0.0001
forthblue	60.06	33.33	13.21	0.0001
drawslasher	64.44	30.00	1.11	0.0001
netsentry	93.10	86.21	0.00	0.004
speedtest	83.70	78.43	11.03	0.0001
qqgame	78.83	98.15	40.49	0.0002
Avg.	80.80	71.73	10.1	

the *frequency feature*, the *binary feature*, and the *sequence feature*, followed by the used kernel. For the two apps that are well detectable, all but one model achieve an accuracy above 85%. In particular, *netsen-*

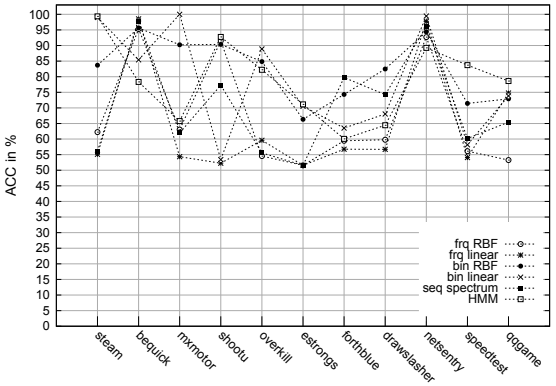
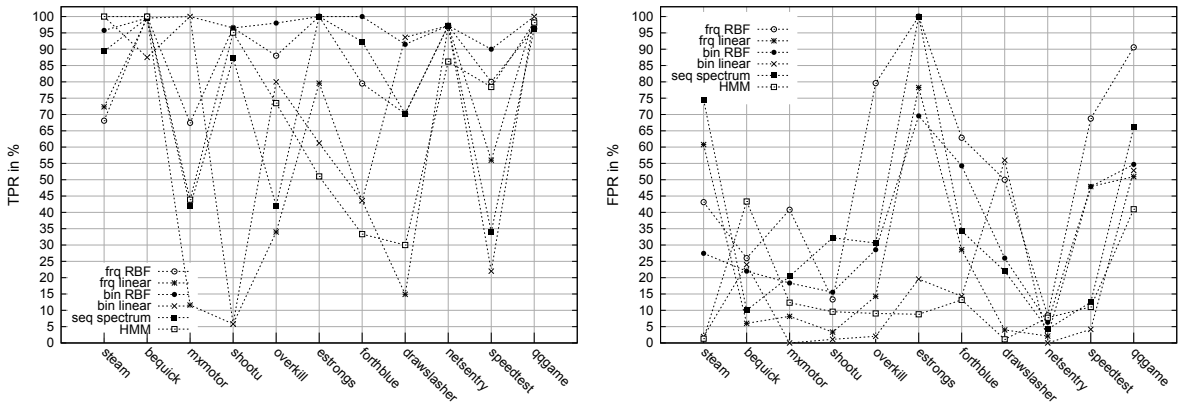


Figure 1: Overview of the accuracy of all evaluated models and features.



(a) True positve rate

(b) False positive rate

Figure 2: Overview of all evaluated models and features.

try also shows prominent peaks in Figure 2 (a) that indicate a high detection rate and in Figure 2 (b) that indicate a low FPR across models. In the case of the poorly detectable apps, only two models achieve an accuracy above 80%, the vast majority of the models perform clearly below 80%. An explanation for this is that both apps with good detection performance are fairly simple utility or fun apps, while the latter apps are a feature-rich file browser (*estrongs*) and games (*drawslasher*, *forthblue*, and *qqgame*), and thus much more complex.

Therefore, it is to assume that the additional malicious payload is much easier to discriminate from the simple apps than from the complex apps. Our results also show, however, that a number of apps show no clear trend across different models and that sufficient accuracy for these apps can only be achieved in few of our models.

Taking a step back to the overall picture, our evaluation shows that one-class SVM-based modeling achieves the best performance with the *binary feature*, i.e., with the simplest of the three features, outperforming the *frequency feature* and the *sequence feature* considerably. Using the accuracy as basis for these comparisons, the *sequence feature* processed by the spectrum kernel performs second best, while the *frequency feature* lags far behind. The bad performance of the *frequency feature* is especially surprising, since this feature shows good performance in prior work (see (Dini et al., 2012; Burguera et al., 2011)). In the case of *MADAM* by Dini *et al.* an explanation could be found in the enrichment of the feature vectors (see Section 2), leaving it unclear how much the additional features contribute to the detection performance. The fact that the best performance with one-class SVMs can be achieved with the *binary feature* suggests that the introduction of system calls by the malicious payload that are not present in the benign traces is most discriminative in the context of this model. The strong performance of the *binary feature* compared to the more complex features indicates, that the variations of the system call frequencies and the additional information encoded in the temporal ordering is not distinctive enough that it can be reliably learned by our one-class SVM-based classifier. The inferior result regarding the *sequence feature* in the one-class SVM case, however, is contrasted by the strong performance of this feature in the HMM-based approach. This, in turn, suggests that the used spectrum kernel does not capture the temporal structure of the underlying data nearly as well as the HMM does.

The most pronounced result of our evaluation is, that the structure and complexity of the feature has to be taken into consideration when choosing the actual modeling approach. Taking the most important values—accuracy, detection rate and false positive rate—into account, it is remarkable that both one-class SVMs with linear kernel on the *binary feature* and HMMs on the *sequence feature* achieve the best results with almost identical average values. Simplified one can state that there is not one best modeling approach for anomaly-based mobile malware detection on system calls, but that there are best modeling approaches for each particular derived feature.

7 CONCLUSION

In this paper, we proposed the use of one-class classifiers in the context of mobile malware detection. These classifiers have the advantage that they are trained with data from the benign class only, do thus not require access to malware samples during training, and have a great potential to detect new malware

that significantly differs from known malware. We systematically compared HMMs (a generative probabilistic model) and one-class SVMs with different kernels (discriminative, non-probabilistic model) in combination with three different feature types derived from system call traces. We showed that HMMs in combination with the sequence feature slightly outperform the other modeling approaches. The most discriminative feature type, however, strongly depends on the model chosen. In particular, despite of the good performance of the sequence feature in the HMM case, the binary feature outperformed both the frequency feature and the sequence feature in the one-class SVM case. Overall, our evaluation shows that the discriminatory power of the features derived from system call traces varies greatly on a per-app basis which indicates that not all malicious behavior is appropriately covered with system call traces. We identified the heavy use of inter-process communication (IPC) on the Android platform as a main reason. In future work we plan to study low-level features that provide superior discrimination power than system calls and to evaluate them with state-of-the art models from the field of one-class classification.

REFERENCES

Aafer, Y., Du, W., and Yin, H. (2013). Droidapiminer: Mining api-level features for robust malware detection in android. In *SecureComm*, volume 127 of *LNICST*, pages 86–103. Springer.

Aldini, A., Martinelli, F., Saracino, A., and Sgandurra, D. (2014). Detection of repackaged mobile applications through a collaborative approach. *Concurrency and Computation: Practice and Experience*.

Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K., and Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of NDSS*.

Bose, A., Hu, X., Shin, K. G., and Park, T. (2008). Behavioral detection of malware on mobile handsets. In *Proceedings of ACM MobiSys*, pages 225–238.

Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. (2011). Crowdroid: behavior-based malware detection system for android. In *Proceedings of ACM SPSM*.

Dini, G., Martinelli, F., Matteucci, I., Saracino, A., and Sgandurra, D. (2014). Introducing probabilities in contract-based approaches for mobile application security. In *Data Privacy Management and Autonomous Spontaneous Security*, pages 284–299. Springer.

Dini, G., Martinelli, F., Saracino, A., and Sgandurra, D. (2012). Madam: A multi-level anomaly detector for android malware. In *MMM-ACNS*, volume 7531 of *LNCS*, pages 240–253. Springer.

Eskin, E., Arnold, A., Prerau, M., Portnoy, L., and Stolfo, S. (2002). A geometric framework for unsupervised anomaly detection: Detecting intrusions in unlabeled data. In *Applications of Data Mining in Computer Security*, pages 77–101. Springer.

Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. (1996). A sense of self for unix processes. In *Proceedings of IEEE S&P*.

Gartner (2014). Gartner says by 2018, more than 50 percent of users will use a tablet or smartphone first for all online activities. http://www.gartner.com/newsroom/id/2939217.

IDC (2014). Worldwide smartphone growth forecast. http://www.idc.com/getdoc.jsp?containerId=prUS25282214.

Leslie, C. S., Eskin, E., and Noble, W. S. (2002). The spectrum kernel: A string kernel for SVM protein classification. In *Proceedings of Pacific Symposium on Biocomputing*, pages 566–575.

Maggi, F., Matteucci, M., and Zanero, S. (2010). Detecting intrusions through system call sequence and argument analysis. *IEEE TDSC*, 7(4):381–395.

Mutz, D., Valeur, F., Vigna, G., and Kruegel, C. (2006). Anomalous system call detection. *ACM TISSEC*, 9(1):61–93.

Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286.

Schölkopf, B., Williamson, R. C., Smola, A. J., Shawe-Taylor, J., and Platt, J. C. (1999). Support vector method for novelty detection. *NIPS*, 12:582–588.

Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., and Weiss, Y. (2012). "Andromaly": a behavioral malware detection framework for android devices. *JIIS*, 38(1):161–190.

Warrender, C., Forrest, S., and Pearlmutter, B. (1999). Detecting intrusions using system calls: alternative data models. In *Proceedings of IEEE S&P*.

Xie, L., Zhang, X., Seifert, J., and Zhu, S. (2010). pBMDS: a behavior-based malware detection system for cellphone devices. In *Proceedings of ACM WiSec*.

Yeung, D. and Ding, Y. (2003). Host-based intrusion detection using dynamic and static behavioral models. *Pattern Recognition*, 36(1):229–243.

Zhang, M., Duan, Y., Yin, H., and Zhao, Z. (2014). Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of ACM CCS*.

Zhou, Y. and Jiang, X. (2012). Dissecting Android Malware: Characterization and Evolution. In *Proceedings of IEEE S&P*.