APPROVAL SHEET

Title of Thesis: ANDROID MALWARE DETECTION AND CLASSIFICATION USING MACHINE LEARNING TECHNIQUES

Name of Candidate: Satyajit Padalkar
MS Computer Science, 2014

Thesis and Abstract Approved: _____

Dr. Anupam Joshi
Professor
Department of Computer Science and
Electrical Engineering

Date Approved: _____

Curriculum Vitae

Name: Satyajit Padalkar

Degree and date to be conferred: Master of Science, Summer 2014

Secondary Education:

- Tuljaram Chaturchand College, Baramati, Maharashtra

Collegiate institutions attended:

- University of Maryland Baltimore County, MS Computer Science 2014
- Pune Institute of Computer Technology (PICT), Bachelor of Engineering 2006

Major: Computer Science

Professional positions held:

- Sr. Software Developer, CalSoft Pvt. Ltd. (2/07/2011 – 8/08/2012)
- Analyst Developer, Goldman Sachs (7/19/2010 – 2/04/2011)
- Software Engineer, Symphony Services Corp. Ltd. (7/17/2006 – 7/09/2010)

ABSTRACT

Title of Thesis: Android Malware Detection & Classification Using Machine Learning Techniques

Satyajit Padalkar, MS Computer Science, 2014

Thesis directed by: Dr. Anupam Joshi, Professor
Department of Computer Science and
Electrical Engineering

Android is popular mobile operating system and there are multiple marketplaces for android applications. Most of these market places allow applications to be signed using self-signed certificates. Due to this practice there exists little or very limited control over the kind of applications that are being distributed. Also advancement of android root kits is making it increasingly easier to repackage existing android applications with malicious code. Conventional signature based techniques fail to detect these malwares. So detection and classification of android malwares is a very difficult problem to solve.

We present a method to classify and detect such malwares by performing dynamic analysis of the system call sequences. Here we make use of machine learning techniques to build multiple models using distributions of syscalls as features. Using these models we predict whether given application is malicious or benign. Also we try to classify given application to specific known malware family. We also explore deeplearning methods such as stacked denoising autoencoder (SdA) algorithms and its effectiveness.

We experimentally evaluate our methods using a real dataset of 600 malicious applications spread across 38 malware families along with 25 popular benign applications from various areas. We find that deeplearning algorithm (SdA) is most accurate in detecting a malware with lowest false positives while AdaBoost performs better in classifying a malware family.

ANDROID MALWARE DETECTION AND CLASSIFICATION USING MACHINE LEARNING TECHNIQUES

by

Satyajit Padalkar

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2014

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Dedicated

to the memory of my dear mother, Kalpana

&

to my father, Mahadev Padalkar, who has put in tremendous efforts while raising his both sons. Over these years he has taxed himself dearly while providing for my education. Hopefully, I would make him feel proud by completing my master's studies.

ACKNOWLEDGMENTS

I'd like to acknowledge first and foremost, my advisor Dr. Anupam Joshi who was a constant source of inspiration and much needed guidance. I am grateful for his patience through out my thesis. I would also like to thank Dr. Xuxian Jiang of NCSU for trusting me with his dataset. This thesis would also not have been possible without the constant help, regular feedback from all my colleagues working in the ebiquity lab at UMBC. And last but not the least, a big thanks to my brother Vaibhav, Maiee, Vaihini and all my awesome friends who stood by me through thick and thin in the last 2 years.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

In this chapter we briefly describe the android platform and the current threat landscape in mobile devices domain. Next, we discuss the motivation for our work, the problem this thesis tries to solve, and describe the proposed solution. Chapter two briefly touches upon prior work in this domain. Chapter three describes the design and architecture of the proposed malwares detection system while giving a walkthrough of the multiple modules in system and their individual role. Chapter four dives into the methodology we followed while conducting our experiments and how we tested its effectiveness. Chapter five describes the results obtained from our experiments followed by a short concluding piece which describes possible enhancements and future work.

1.1 Android Platform

In the last few years smartphone usage has increased dramatically. It is estimated that in year 2014 USA would have 163.9 million [1] smartphone users and globally there would 1.204 billion [2] users.

Figure 1.1 provides current smartphone usage trends in USA. Smartphone technology is advancing over period of time providing higher computational power, newer sensors, improved network speed and with higher battery capacity. Due to this smartphones are making

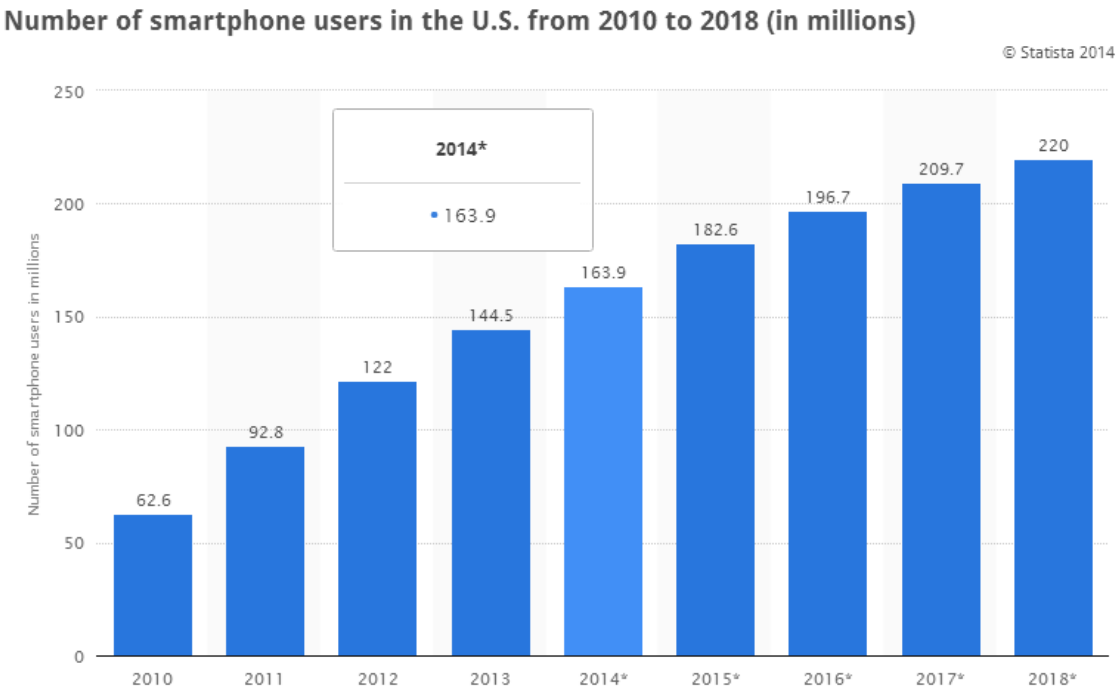


FIG. 1.1. Smartphone Usage in USA

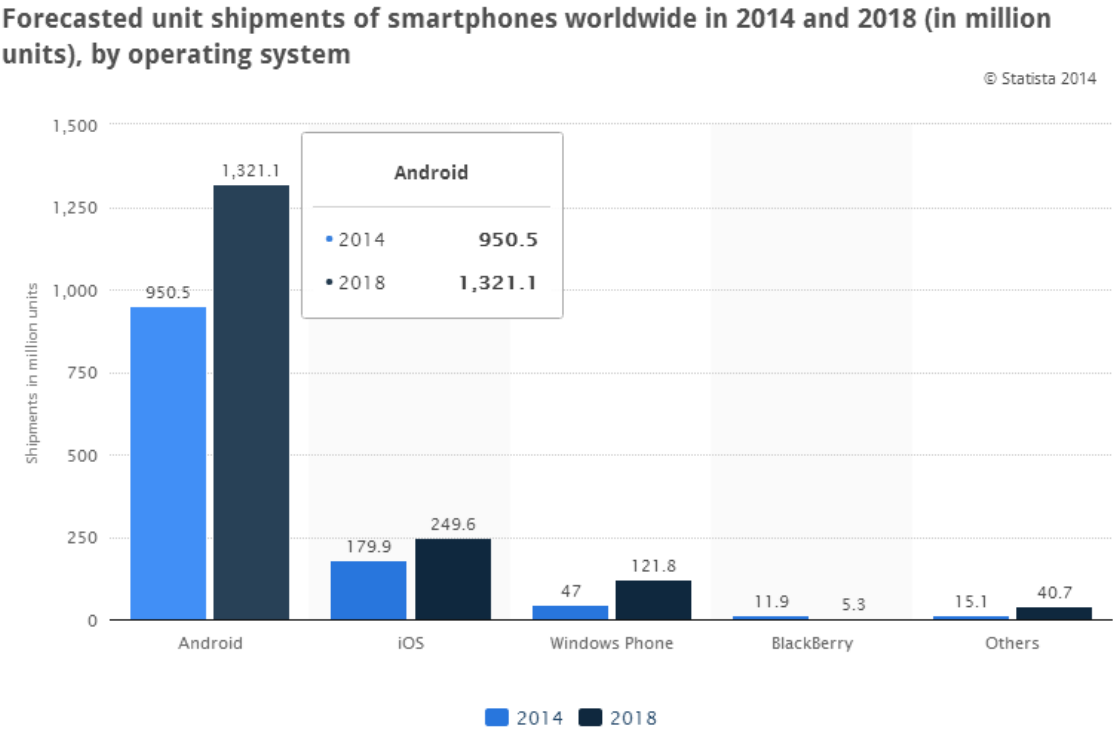


FIG. 1.2. Android OS Market Share 2014 - 2018

inroads to newer and wider application areas, e.g education, retail, hospitality, insurance, entertainment and many more.

Android is most popular operating system world wide and it is expected to be shipped on 1.32 billion [2] smartphones in year 2018. This kind of popularity not only attracts the attention of developers but also from those with malicious intentions. Android applications can be downloaded from various market places. Usually developer needs to cryptographically sign the application before publishing it on the android store, but most developers use self-signed certificates for this purpose. Due to this practice there exists limited authentication mechanism. Also there are tools [3] to reverse engineer and repackage these applications with custom code inserted in to it. As a result, repackaged application is most used attack vector on android platform.

The rapid increase in the amount of new repackaged malicious applications is of great concern, since many of the modern antivirus products are not able to keep up with this growing threat and keep their malware signature database up to date. A research study by F-Secure Labs [4] shows that less than 5% of all recorded malware specimens were detected by existent malware scanners.

The detection rate is even worse when code obfuscating techniques are used while repackaging these applications. The malicious payload can be modified in multiple different ways to evade the signature based antivirus. For example inserting junk code that doesn't alter functionality, by rearranging the code structure and by replacing part of code with equivalent code.

These kind of changes allow repackaged applications to produce various variants of similar application with different signatures. Though the signature is different, in general the behaviour of the application remains same. Due to this, it is impossible to detect all variants of malware with a few known signatures. This problem can be addressed by dynamically analysing behaviour of application and its interaction with android platform. Here we

model the behaviour of application by analysing the system call sequence and its distribution.

This thesis aims at finding robust and effective techniques for detection and classification of android malware. Though malware detection is primary focus of this thesis work, we also want to classify the given malware to its known family. As android malware families are well documented for their infection mechanism, behaviors and possible removal steps, knowing malware family name provides valuable information. This information would be useful for narrowing the scope of the investigation and reduce the response time required to mitigate the attack. We use android system calls as feature set for building different classifiers. Classifiers are built using boosting on an ensemble of weak learners. We conduct an extensive experimental study to evaluate the performance of different boosting techniques and ensembles. The classification done using syscalls distribution is much robust and shows significant association with the malware's family. For detection purpose we also build deeplearning model, using Stacked Denoising Autoencoder (SdA) approach. The SdA model is most robust with lowest false positive rate.

While constructing features for malware detection and classification we perform dynamic analysis of each specimen app on an instrumented android device. Each application run provides log files with system call sequence that was observed by android platform for given application. We build the system call distribution using these log files and use that as a feature set for classifying given application to malicious or benign app. Also we use this distribution for classifying the given dataset of malicious apps into corresponding known malware families.

Before performing this analysis, caution should be taken as these malicious apps are variants of repacked applications. For collecting reasonable system calls multiple runs needs to be done. Also for avoiding any bias towards specific system calls we normalize the dataset using Z-score. Our experiments show that malware detection using SdA classifier provides

better performance and accuracy than other approaches that we evaluated. Several experiments were performed to check for the robustness and the validity of the claim that SdA performs better with highest detection rate and lowest false positive rate. The stability of SdA is tested by introducing noise in the input data set.

Chapter 2

OVERVIEW

In this section we will briefly describe the android application, its publication mechanism, current threat landscape and algorithms that we would for classification.

2.1 Android Applications

Android applications are usually written in Java (although some have ”native” C calls), and are distributed as APK (Android Package) files. Those APK files are in fact Zip archives, which contain compiled Java classes (in Dalvik DEX format), application resources, and an AndroidManifest.xml binary XML file containing application metadata. The APK also contains a public key and its associated $X.509$ certificate, bundled as a PKCS#7 message in DER format.

2.1.1 Naming conventions

When creating a new project, the Android developer documentation dictates that a full ”Java-language-style” package name be used, and that developers ”should use Internet domain ownership as the basis for package names (in reverse) [5].” This creates package names such as com.google.gmail for the mobile GMail application. To avoid name conflicts, package names must be unique across the entire universe of applications. Using

reversed domain names theoretically limits potential namespace conflicts to a developer's own domain.

2.1.2 Signing applications

All Android applications must be cryptographically signed by the developer; an Android device will not install an application that is not signed. Typical Java tools, such as keytool and jarsigner, may be used to create a unique keypair and sign the mobile application.
In Android, the only key distribution mechanism used consists in bundling developer's public key with the application. Further, Android has no requirement for a keypair to be certified by a Certificate Authority (CA). In a study researchers [6] have observed that more than 99% of the 76,480 applications they studied use self-signed certificates.
In other words, the primary purpose of the keypair is to distinguish between application authors, but not to provide any stronger security properties. In practice, keypairs are also used for:

- Ensuring that applications that are allowed to automatically update are signed by the same key as the previous version.
- Potentially allowing applications signed by the same key to share resources.
- Granting or denying permissions to a family of applications signed by the same key.
- Removing all applications signed with the same key from the Android market and potentially from all connected devices when one of these applications is flagged as malware [7].

On the other hand, due to the absence of any certification authority or PKI, signatures on Android do not provide any assurance about the identity of the signer. In essence, Android ensures that the AngeryBird application is correctly signed by somebody, but cannot prove

that the Rovio Entertainment Ltd, the company behind the AngeryBird application, actually signed the AngeryBird application.

2.2 Application Repackaging

An existing application redistributed with a different signing key, often with functionality not present in the original version, is said to be repackaged. Some, all, or none, of the application's existing functionality can be preserved in the repackaged version.

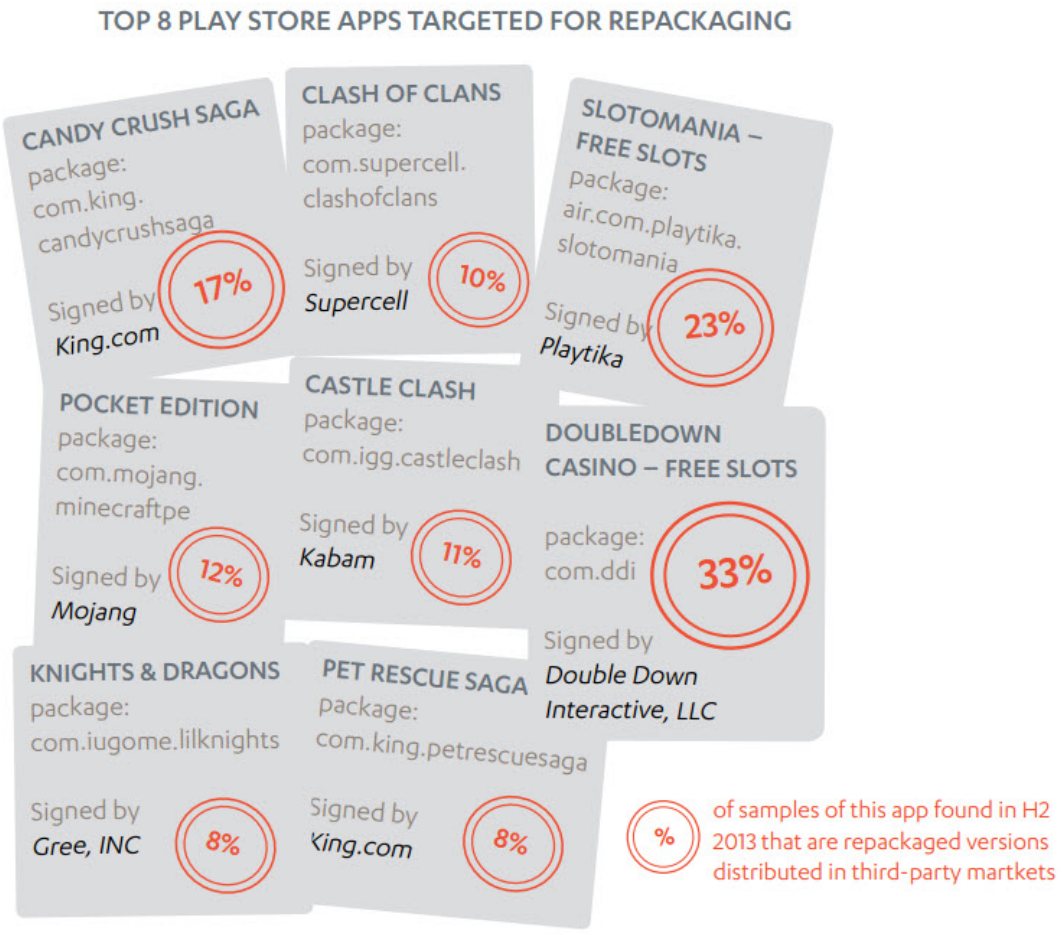


FIG. 2.1. Top 8 Repackaged Applications

Applications can be repackaged for many reasons other than to distribute malware. For instance, a repackager may simply wish to add advertising to an existing application to profit from somebody else's application. Application repackaging falls broadly in two classes: spoofing and grafting. Figure 2.1 shows few of most popular repackaged applications found on android market places.

2.2.1 Spoofing

Mobile applications can simply be published under false pretenses, spoofing little or none of the features a legitimate application would possess. To deceive the user, a malicious program may advertise to be an existing application, or a non existent application that may plausibly exist, yet provide none of the expected functionality. As previously shown in peer-to-peer networks [8] and search-engine result poisoning [9, 10], this type of attack could flood a market with enough false positives to attract users.

As an example, in July 2011, the legitimate Netflix application only supported specific devices and versions of Android. Unsupported devices could not locate and install the official application in the market. In October 2011, a fake version of the Netflix application was published in the official Android Market, claimed to "support" all devices, and thus appeared to owners of devices that could not download the legitimate application. The fake application displayed a plausible login screen, but then simply stole service credentials. Once credentials were entered, the application uninstalled itself [10].

2.2.2 Grafting

To achieve the desired functionality of a legitimate application, an attacker may elect to graft malware onto an existing application, and subsequently republish the modified application. The attacker starts by downloading and extracting an existing application. To do so, she unzips the APK archive to extract the application components (class files and manifest).

Then, she adds malware to the application, and repackages it. Adding malware may require to reverse the DEX-formatted Java classes. While not entirely straightforward, tools such as undx [11], baksmali, dedexer, or ded [12] can often successfully decompile .dex files to source code. DEX can also be converted to a typical Java jar collection of classes using the dex2jar utility, at which point a typical Java decompiler can be used.

In the quite common case in which the .dex file does not need to be fully reversed to source code, much of the disassembly and repackaging process can be automated. For instance, apktool [3] can unpack and repackage an existing .apk with two commands. apktool has several side effects that result in non-required changes to the repackaged .apk. For instance, some files may be compressed in the repackaged application regardless of whether or not original file was compressed. With automatic compression the repackaged file may actually be smaller than the original despite the addition of malicious code. These side effects may be undesirable for an attacker that wishes for the application to remain as similar as possible to the original application.

In addition to the class files, the attacker may need to modify the AndroidManifest.xml, since this is where applicationlevel permissions are specified. This can be done using a tool such as AXMLPrinter2 [13]. For instance, the malware to be added to the existing application may require the INTERNET or SEND_SMS permissions, even if the permission is not specified in the original application.

Last, prior to publishing the ”new” application to one or more markets, the attacker must cryptographically sign the application. The signing can easily be performed with standard Java tools, e.g.,using jarsigner. Since Android uses self-signed certificates, such signatures will pass installation-time checks.

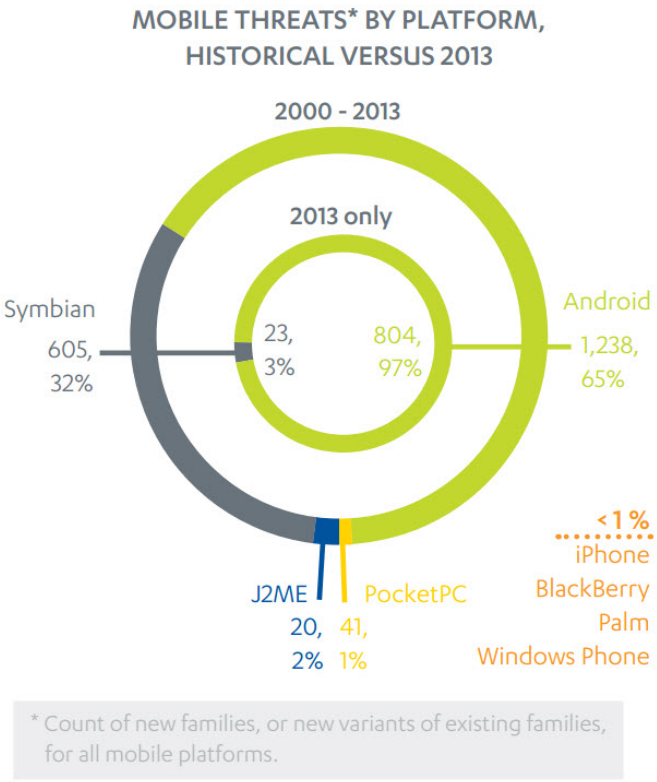


FIG. 2.2. Mobile Threats by Platform

2.3 Current Android Malware Landscape

Recent studies shows that Android system is increasingly being exploited by numerous malwares. Also most of these malwares are specifically created for android system. Figure 2.2 shows the recent mobile threat landscape. Also there is upward trend in using repackaged applications as a attack vector. Figure 2.3 shows that most of malicious apps come form third party market place. In 2013, Google play store found $\tilde{0}.1\%$ applications to be malicious (136 from 132,798 unique applications recieved).

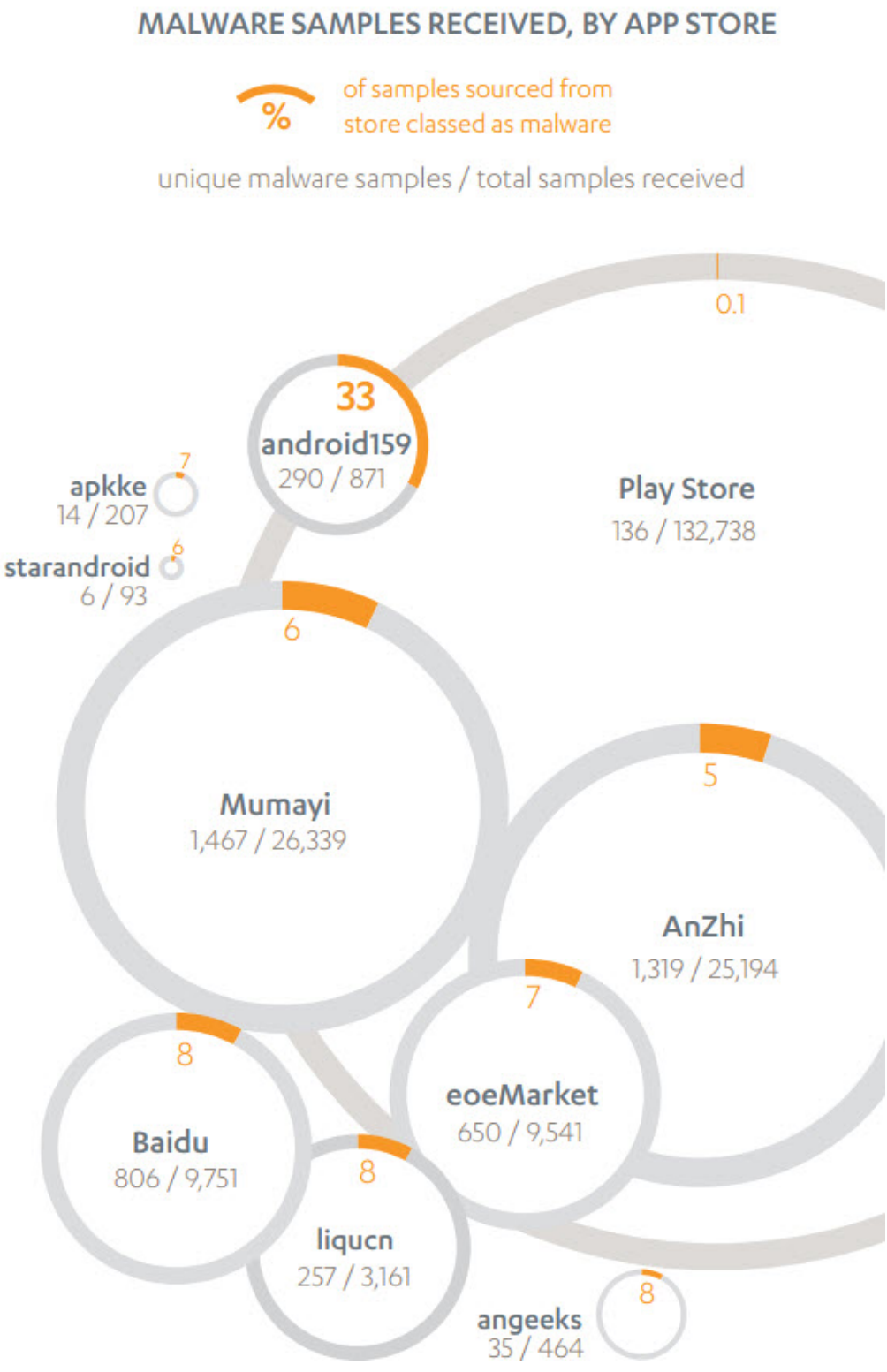


FIG. 2.3. Android Malwares by Market Place

2.4 Boosting

‘ Boosting is a machine learning technique for reducing the bias that sometimes results from supervised learning [14]. The main idea is to use an ensemble of weak learners to iteratively create a strong learner for classification. Generally, boosting algorithms consist of iteratively learning weak classifiers with respect to a distribution and then combining them to construct a final strong classifier with good classification performance. When the classifiers are added, they are typically weighted in a way that is closely related to the weak learners' accuracy. Every time a weak learner is added, the fitted classification data used to recalculate the instance weights. Input instances that are miss-classified gain weight, and instances that are classified correctly lose weight. Thus, subsequent weak learners focus more on the inputs that previous weak learners miss–classified. We use AdaBoostM2 (which are discussed next) for classification of the malware specimens into families.

AdaBoost is the standard algorithm which is basis for AdaBoostM2. AdaBoost, short for "Adaptive Boosting", is a machine learning algorithm and can be used in conjunction with many other learning algorithms to improve their performance. AdaBoost is adaptive in the sense that subsequent classifiers built are corrected in favor of those instances miss–classified by previous classifiers. AdaBoost is sensitive to noisy data and outliers. In some problems, however, it can be less susceptible to the over-fitting problem than most learning algorithms. The classifiers that AdaBoost algorithm uses can be weak: as long as their performance is slightly better than random, the classification performance will improve the final model. AdaBoost fits a new weak classifier in each of a series of iterations $t = 1, \ldots, T$. For each iteration t, a distribution of weights D_t is updated. The weights indicate the importance of each input instance in the data set for the classification. For each iteration, the weights of each incorrectly classified input are increased, and the weights of each correctly classified input are decreased. This implies that the new classifier focuses

Given: $(x_1, y_1), \ldots, (x_m, y_m)$ where $x_i \in \mathcal{X}$, $y_i \in \{-1, +1\}$.
Initialize: $D_1(i) = 1/m$ for $i = 1, \ldots, m$.
For $t = 1, \ldots, T$:

- Train weak learner using distribution D_t.
- Get weak hypothesis $h_t : \mathcal{X} \to \{-1, +1\}$.
- Aim: select h_t with low weighted error:

$$\varepsilon_t = \Pr_{i \sim D_t} [h_t(x_i) \neq y_i].$$

- Choose $\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$.
- Update, for $i = 1, \ldots, m$:

$$D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$$

where Z_t is a normalization factor (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:

$$H(x) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t h_t(x) \right).$$

FIG. 2.4. AdaBoost algorithm

on the input which has so far been miss–classified. The AdaBoost algorithm [15] which serves as the standard base algorithm for other boosting techniques that we have used is shown in the figure 2.4.

In the context of malware classification, for boosting, the frequencies of the syscalls in each document are taken as the features/predictors and the family names are taken as the responses/target classes. Similarly the word distribution in each file is calculated. In this case the frequencies of the words in each document are taken as the features and the corresponding family names are taken as responses. The performance of the classification using the boosting algorithm AdaBoostM2 are compared with other algorithms. We experimentally evaluate the classification performance of AdaBoostM2 for malware family prediction.

2.4.1 AdaBoostM2

AdaBoostM2 is an extension of the standard AdaBoost algorithm. This boosting algorithm is designed for multi–class problems with weak base classifiers. The algorithm is designed to minimize a loose bound on the training error. Since there are multiple classes of metamorphic malware, we opt for AdaBoostM2. In AdaBoostM2, first, we allow the weak learner to generate more expressive hypotheses, which, rather than identifying a single class label, instead choose a set of plausible labels. This may often be easier than choosing just one class label. It is seen that the boosting algorithm AdaBoostM2, achieves boosting if each weak hypothesis has pseudo-loss slightly better than random guessing.

Rather than using the usual prediction error, we settle for having the weak hypotheses do well with respect to a more sophisticated error measure that we call the pseudo-loss. Unlike ordinary error which is computed with respect to a distribution over examples, pseudo-loss is computed with respect to a distribution over the set of all pairs of examples and incorrect labels. In the algorithm [15] described in figure 2.5, B is a set of all mislabels. On each round t of boosting, AdaBoostM2 supplies the weak learner with a mislabel distribution D_t. In response, the weak learner computes the hypothesis h_t. Intuitively, we interpret each mislabel (i, y) as representing the binary question of the form: Do you predict the label associated with x_i is y_i (correct) or y (incorrect)? The weak learners goal is to find the weak hypothesis h_t with small pseudo loss. Then the distribution D_t is updated with this pseudo-loss. After T iterations, the final hypothesis for AdaBoostM2 is formulated.

2.5 Stacked Denoising Autoencoder (SdA)

2.5.1 Notations

Let X and Y be two random variables with joint probability density $p(X, Y)$, with marginal distributions $p(X)$ and $p(Y)$. Throughout the text, we will use the following notation:

Input: sequence of m examples $\langle (x_1, y_1), \ldots, (x_m, y_m) \rangle$
with labels $y_i \in Y = \{1, \ldots, k\}$
weak learning algorithm **WeakLearn**
integer T specifying number of iterations

Let $B = \{(i, y) : i \in \{1, \ldots, m\}, y \neq y_i\}$
Initialize $D_1(i, y) = 1/|B|$ for $(i, y) \in B$.
Do for $t = 1, 2, \ldots, T$

1. Call **WeakLearn**, providing it with mislabel distribution D_t.
2. Get back a hypothesis $h_t : X \times Y \rightarrow [0, 1]$.
3. Calculate the pseudo-loss of h_t:
$$\epsilon_t = \frac{1}{2} \sum_{(i,y) \in B} D_t(i, y)\big(1 - h_t(x_i, y_i) + h_t(x_i, y)\big).$$
4. Set $\beta_t = \epsilon_t / (1 - \epsilon_t)$.
5. Update D_t:
$$D_{t+1}(i, y) = \frac{D_t(i, y)}{Z_t} \cdot \beta_t^{(1/2)(1 + h_t(x_i, y_i) - h_t(x_i, y))}$$
where Z_t is a normalization constant (chosen so that D_{t+1} will be a distribution).

Output the final hypothesis:
$$h_{fin}(x) = \arg\max_{y \in Y} \sum_{t=1}^{T} \left(\log \frac{1}{\beta_t} \right) h_t(x, y).$$

FIG. 2.5. AdaBoostM2 algorithm

Expectation: $E_{p(X)}[f(x)] = \int p(x)\, f(X)\, dx$ where $y = f(x) = s(Wx + b)$

Entropy: $H(X) = H(p) = E_{p(X)}[\, -log\, p(X)\,]$

Conditional entropy: $H(X|Y) = E_{p(X,Y)}[\, -log\, p(X|Y)\,]$

Kullback-Leibler divergence: $DKL(pkq) = E_{p(X)}[\, log\, p(X)q(X)\,]$

Cross-entropy: $H(pkq) = Ep(X)[-\, log\, q(X)] = H(p)\, + DKL(pkq)$

Mutual information: $(X;Y) = H(X) - H(X|Y)$

Sigmoid: $s(x) = \frac{1}{1+e^{-x}}$ and $s(x) = (s(x1), \ldots, s(xd))^T$

Bernoulli distribution with mean u : $B_u(x)$ and by extension $B_u(x) = (B_{u1}(x1), \ldots, B_{ud}(xd))$

The setup we consider is the typical supervised learning setup with a training set of $n(input, target)$ pairs $Dn = (x(1), t(1)), \ldots, (x(n), t(n))$, that we suppose to be an i.i.d. sample from an unknown distribution $q(X, T)$ with corresponding marginals $q(X)$ and $q(T)$.

2.5.2 The Basic Autoencoder

We begin by recalling the traditional autoencoder model such as the one used in [16] to build deep networks. An autoencoder takes an input vector $x \in [0, 1]^d$, and first maps it to a hidden representation $y \in [0, 1]^{d'}$ through a deterministic mapping $y = f(x) = s(Wx + b)$, parametrized by $\theta = W, b$. W is a $d' \times d$ weight matrix and b is a bias vector. The resulting latent representation y is then mapped back to a "reconstructed" vector $z = g_{\theta'}(y) = s(W'_y + b')$ with $\theta' = W', b'$. The weightmatrix W' of the reverse mapping may optionally be constrained by $W' = W^T$,in which case the autoencoder is said to have tied weights. Each training x^i is thus mapped to a corresponding y^i and a

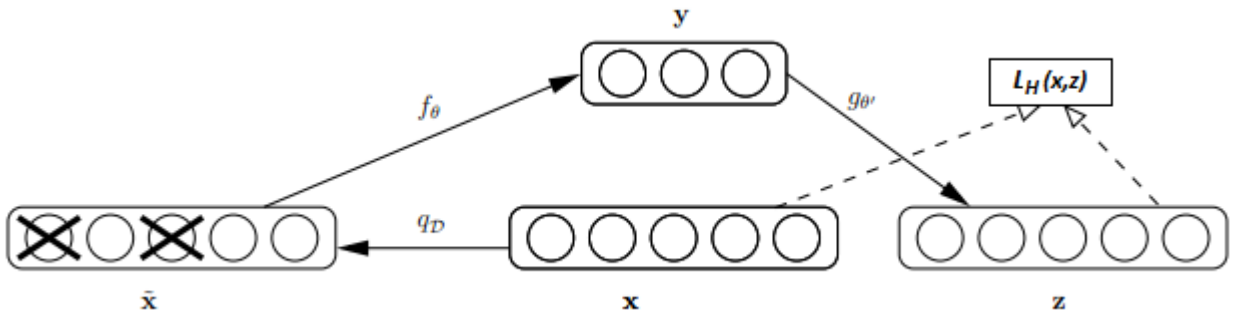


FIG. 2.6. An example x is corrupted to $\widetilde{x}$. The autoencoder then maps it to y and attempts to reconstruct x.

reconstruction z^i. The parameters of this model are optimized to minimize the average reconstruction error:

$$\begin{aligned} \theta^*, \theta'^* &= \underset{\theta,\theta'}{arg\,min} \frac{1}{n} \sum_{i=1}^{n} L(x^{(1)}, z^{(i)}) \\ &= \underset{\theta,\theta'}{arg\,min} \frac{1}{n} \sum_{i=1}^{n} L\left(x^{(i)}, g_{\theta'}(f_\theta(x^{(i)}))\right) \end{aligned} \tag{2.1}$$

where L is a loss function such as the traditional squared error $L(x, z) = ||x - z||^2$. An alternative loss, suggested by the interpretation of x and z as either bit vectors or vectors of bit probabilities (Bernoullis) is the $reconstructioncrossentropy$:

$$\begin{aligned} L_H(X, Z) &= H(B_X||B_Z) \\ &= -\sum_{k=1}^{d} [X_k \, logZ_k + (1 - X_k) \, log\,(1 - Z_k)] \end{aligned} \tag{2.2}$$

Note that if x is a binary vector, $L_H(x, z)$ is a negative log-likelihood for the example x,

given the Bernoulli parameters z. Equation 2.4 with $L = L_H$ can be written

$$\theta^*, \theta'^* = \underset{\theta,\theta'}{arg\,min} E_{q^0(X)} \left[L_H \left(X, g_{\theta'}(f_\theta(X)) \right) \right] \tag{2.3}$$

where $q^0(X)$ denotes the empirical distribution associated to our n training inputs. This optimization will typically be carried out by stochastic gradient descent.

2.5.3 The Denoising Autoencoder

To test our hypothesis and enforce robustness to partially destroyed inputs we modify the basic autoencoder we just described. We will now train it to reconstruct a clean repaired input from a corrupted, partially destroyed one. This is done by first corrupting the initial input x to get a partially destroyed version $\widetilde{x}$ by means of a stochastic mapping $\widetilde{x}\tilde{q}_D(\widetilde{x}|x)$. In our experiments, we considered the following corrupting process, parametrized by the desired proportion v of destruction: for each input x, a fixed number vd of components are chosen at random, and there value is forced to 0, while the others are left untouched. The procedure can be viewed as replacing a component considered missing by a default value, which is a common technique. A motivation for zeroing the destroyed components is that it simulates the removal of these components from the input. For our examples, this corresponds to salt noise. Note that alternative corrupting noises could be considered. The corrupted input $\widetilde{x}$ is then mapped, as with the basic autoencoder, to a hidden representation $y = f_\theta(\widetilde{x}) = s(W\widetilde{x} + b)$ from which we reconstruct a $z = g_{\theta'}(y) = s(W'_y + b')$ (refer 2.6 schematic process). As before the parameters are trained to minimize the average reconstruction error $L_H(X, Z) = H(B_X||B_Z)$ over a training set, i.e. to have z as close as possible to the uncorrupted input x. But the key difference is that z is now a deterministic function of $\widetilde{x}$ rather than x and thus the result of a stochastic mapping of x.

Let us define the joint distribution: $q^0(X, \widetilde{X}, Y) = q^0(X)q_D(\widetilde{X}|X)\delta_{f_\theta(\widetilde{X})}(Y)$ where $\delta_u(v)$

puts mass 0 when $u \neq v$. Thus Y is a deterministic function of $\widetilde{X}$. The $q^0(X, \widetilde{X}, Y)$ is parametrized by θ. The objective function minimized by stochastic gradient descent becomes:

$$\theta^*, \theta'^* = \underset{\theta,\theta'}{arg\,min} E_{q^0(X,\widetilde{X})} \left[L_H \left(X, g_{\theta'}(f_\theta(\widetilde{X})) \right) \right] \tag{2.4}$$

So from the point of view of the stochastic gradient descent algorithm, in addition to picking an input sample from the training set, we will also produce a random corrupted version of it, and take a gradient step towards reconstructing the uncorrupted version from the corrupted version. Note that in this way, the autoencoder cannot learn the identity, unlike the basic autoencoder, thus removing the constraint that $d' < d$ or the need to regularize specifically to avoid such a trivial solution.

Chapter 3

RELATED WORK AND DISCUSSION

Forrest $et\ al.$ [17] first proposed the system call malware detection schemes by building a database of normal system call sequences. Warrender $et\ al.$ [18] extended this idea by using hidden Markov models (HMMs) to model sequences of normal system calls. Other researchers Burguera $et\ al.$, Creech $et\ al.$ [19, 20] adapted artificial neural network to perform intrusion detection. Kruegel $et\ al.$ [21] proposed to use system call arguments to improve the performance of host-based intrusion detection. Maggi $et\ al.$ [22] proposed to cluster similar system calls or similar system call arguments to further improve the accuracy.

Earlier researchers [23, 24] also used SOM for network intrusion detection by clustering system call arguments such as user name, connection type, and connection time. Gao $et\ al.$ and Garfinkel [25, 26] perform real-time anomaly detection based on differences between execution graphs and the replica graphs constructed using system call traces and runtime information (e.g., return addresses).

Traditional system call based intrusion detection approaches have to collect all system calls made by the target process, as there is no clear boundary to reduce the collection scope. This increases both noise and design complexity of intrusion detection.

Crowdroid [19] collects system calls on smartphones and sends system call statistics to a

centralized server. Based on the theory of crowdsourcing, symptoms that are shared by a small number of devices are treated as abnormal. Similarly, Paranoid Android Portokalidis *et al.* [27] runs a daemon on the phone to collect behaviors and a virtual replica of the phone in the cloud. The daemon transmits collected behaviors to the cloud and the virtual phone replays the actions happening on the smartphone based on the collected behaviors. Both Crowdroid and Paranoid Android incur 15-30% overhead to smartphone devices.

SmartSiren [28] gathers and reports communication information to a proxy for anomaly detection. Besides the runtime overhead, propagating sensitive communication data to a public server might be a privacy concern for users. Recent work has explored using specific subsets of system calls for smartphone security. Isohara *et al.* [29] monitor a pre-defined subset of system calls such as open on smartphones. pBMDS [30] hooks input-event related functions (e.g., sys_read() for keyboard events, specific drivers for touch events) to collect system calls related to user interaction behaviors (e.g., GUI events). It then performs malware detection using HMMs. This requires significant changes to operating system and are requires custom kernel modules.

Moser *et al.* [31] monitor system calls executed when a program tries to terminate, with the intention of understanding how malware evades detection in virtualized test environments. Bayer *et al.* [32] create malicious application behavioural profiles by combining system call traces with system call dependency and operation information. Kolbitsch *et al.* [33] generate hard to obfuscate models that track the dependencies of system calls in known malware, which they then can use to detect similar type of malware. CloudAV [34] intercepts every open system call and extracts the target file. It compares this signature with signatures of abnormal files maintained in the cloud to detect the access of malicious files. DroidRanger [35] utilizes a signature-based algorithm to detect known malware from markets and monitors sensitive API accesses to discover zero-day malware. These techniques due to there nature has very high false positive rate.

Several researchers have also applied machine learning to statically detect malicious applications based on their permissions [31,35,36]; however, the root exploit malware would be able to defeat these approaches. Previous work has been done to automatically respond to malicious attacks on networked hosts. For example, Somayaji $et\ al.$ [37] delay anomalous system calls with increasing delay durations to prevent security violations such as a buffer overflow. Feinstein $et\ al.$ [38] dynamically apply filtering rules when a DDoS attack is detected. Balepin $et\ al.$ [39] use a cost model to compare predefined exploit costs with various predefined benefits to select the best response to a compromised system resource. Garfinkel $et\ al.$ [26] sandbox applications by using user defined policies to control access to system resources by blocking certain system calls. Additionally, their tool emulates access to these resources to prevent applications from crashing. These changes are either intrusive to application or depends on the user knowledge. Not every user is aware of all security related permission and its impact. So user defined detection or response rules would fail.
In contrast to above all approaches, we try to use noised version of system calls with unsupervised deep learning (SdA) method. As a result, our system is more stable and resilient to noisy input and can work on limited input data set. Also our system does not require any kernel related changes to android operating system which makes it more viable and practical solution.

Chapter 4

SYSTEM DESIGN

4.1 Architecture Overview

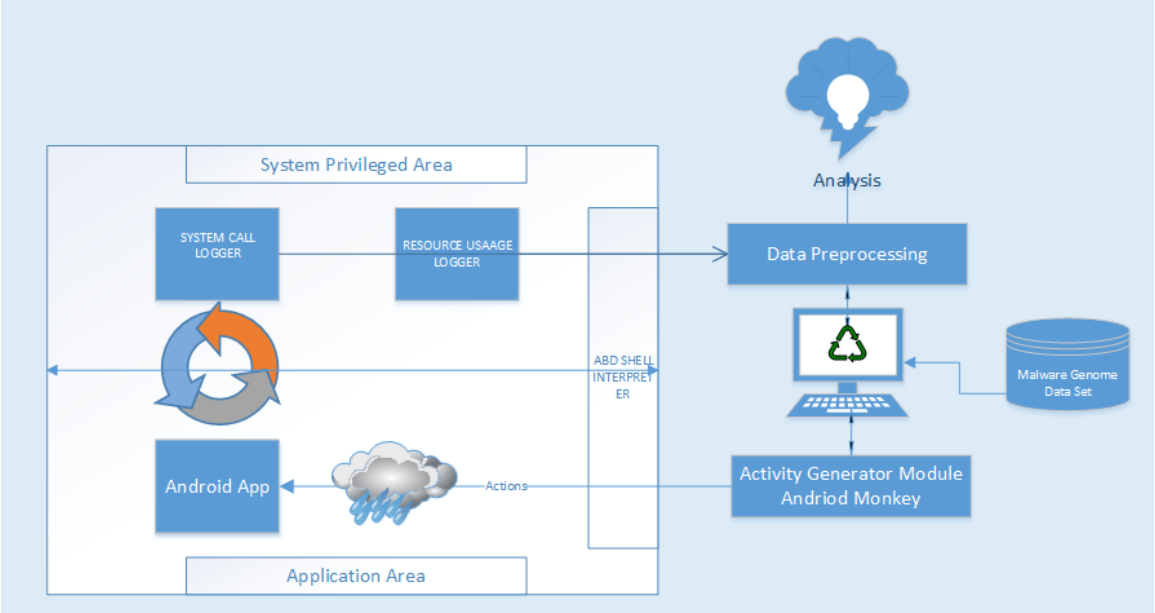


FIG. 4.1. System for Android Malware Classification

The architecture of system is modular in nature and is designed to accommodate various repository locations. Malware samples can be located on a local storage or over a network storage. The system is also designed to automate data collection from android device and

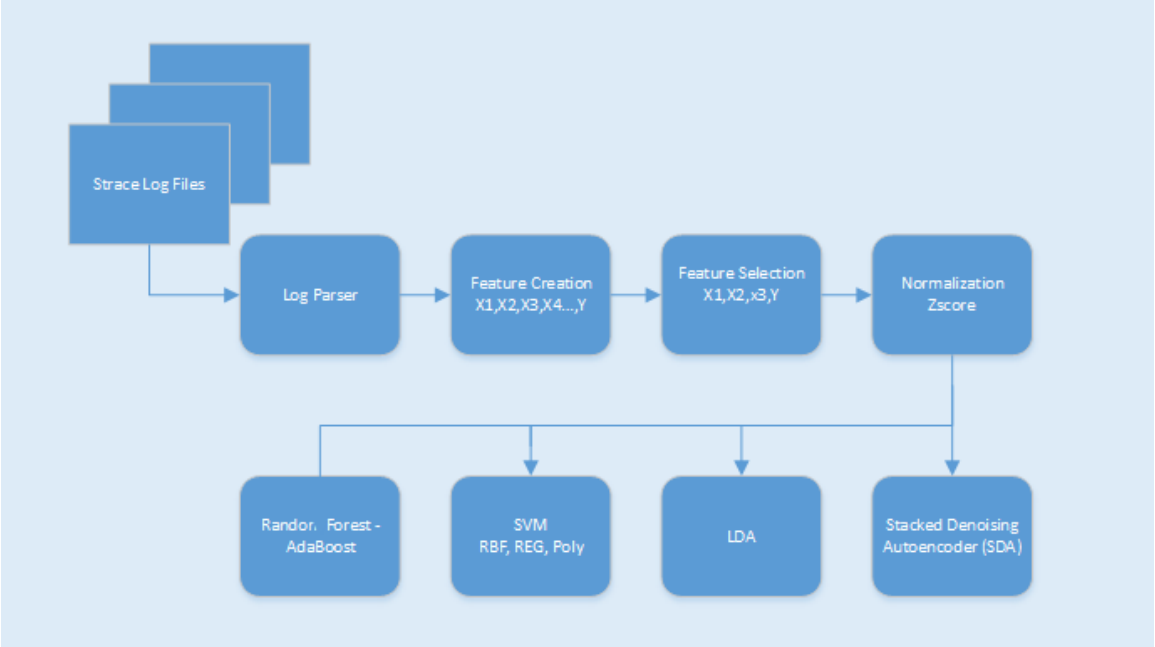


FIG. 4.2. Data Processing & Knowledge Generation

provide all housekeeping services that might be required by user to run various malicious application on a android device in a batch mode. System is capable of collecting system call and resource consumption related trace files. Also it can collect power related information for given application run. Figure 4.1 shows the block diagram of our system. System consists of five major modules:

4.1.1 Data Reader

This module can read data from a local disk or over a network. This module traverse all directories of malware families one by one and grabs the malware sample (.apk file) from that directory for analysis. Each apk file is provided to 'House Keeping' module.

4.1.2 House Keeping

This module is responsible for installation of malicious application on a android phone. We can use android emulator as well. Following activities are performed by House Keeping module.

- Installation of malicious application on a android phone.
- Informing Action Generator module about newly installed application.
- Removal of malicious application from a android phone.
- Collect log files generated for given application run.

4.1.3 Action Generator

This module is built on top of 'Android Monkey' tool. As per user configuration, this module generates pseudo-random streams of user events such as clicks, touches or gestures, as well as a number of system-level events. If required user can generate same sequence of actions. As a default action, tool is configured to generate random actions to mimic the user behavior.

4.1.4 Activity Logger

This module is responsible for logging all system calls and resource consumption related information on a android phone. This module requires "root" permission and needs to be installed on the "system" partition of the android phone.

4.1.5 Data Preprocessor

Once all log files are collected for the experiment run, this module converts collected data into single csv formatted file. Two csv file are generated by this module. These files are

used for:

- Binary classification, where target is set to binary value (0 or 1).
- Multiclass classification, where target is set to ”malware family” name.

This module is responsible for data massaging, feature selection and running various algorithms for experiment. This module can be located on local system or in the cloud.

4.2 System Input

The system requires android malware repository location. Expected directory structure is shown in figure 4.3 where every subdirectory represents $MalwareFamliyName$ and each $.apk$ file represent the android application specimen. System also expects non malicious application to be placed under $benign\_apps$ subdirectory.

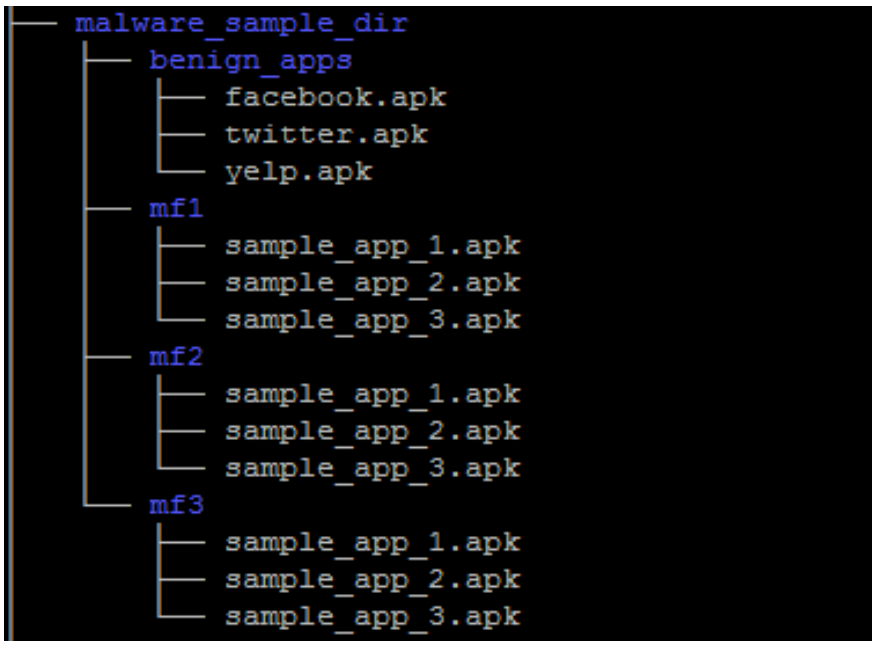


FIG. 4.3. Input Directory Structure

4.3 System Output

For each malware sample system generates a log file on a Android device. System pulls this log file and places into the same subdirectory from where the application specimen, apk file, was selected for installation. To resolve multiple runs of same application, system adds timestamps at the end of filename. All these logs files are then parsed into a csv file where frequency of each system call forms the feature and malware family name forms the target label. The system generates two out files.

- b_android_apps_run.csv : file for binary class classification (maliciousbenign).
- m_android_apps_run.csv : file for multi class classification (malware families).

Typically user needs to normalise the input before building a classifier on this data. We perform this step in two stages.

- Feature Selection: We use $sparseLDA$ normalization algorithm for feature selection.
- Normalize: We use Zscore normalization technique for our analysis.

This would generate a matrix of size $N \times W$, where W is selected features with target as a last column and N is the number of application run for given dataset. This matrix contains normalized values and then used as a input to various classifiers. Binary classification is done to detect malicious vs benign apps from given data set. Multiclass classification is done to classify given application into malware families. We use various algorithms such as RandomeForest, AdaBoost, SVM, Neural Network, SdA for classification. Results are discussed in the section 6.

Chapter 5

DATA AND LAB SETUP

5.1 Hardware

We used $Samsung\ Galaxy\ Nexus$ mobile phone for conducting all experiments. The hardware and software specs are mentioned in Table 5.1.

Table 5.1. Test Device Hardware Specs

Features	Samsung Galaxy Nexus
Operating System	Andriod 4.2 (Rooted)
Main Memory	1024 MB
Processor	Dual core, 1200 MHz, ARM Cortex-A9
Battery Capacity	1750 mAh
Wifi	802.11 a, b, g, n
Sensors	GPS, Proximity, Accelerometer, Gyroscope, Compass, Barometer

5.2 Data Set and Feature Selection

For our analysis, we use Malware Genome Data Set [40]. The data set consists of 38 malware families and 925 malicious applications. Table 5.3 on page 33 provides information regarding malware family and associated number samples. We selected 25 benign applications from various application areas such as Travel, Food, Entertainment, Finance and Games. These applications are listed in Table 5.4.

As discussed earlier in section 4.2 on page 27 we used our system to generate the normalized dataset for 600 malicious and 25 benign applications. For our analysis, feature selection is done using sparseLDA normalization algorithm. Table 5.2 shows all the features that we selected for model building. After feature selection, data set is normalized using Zscore normalization method. This normalized data set is the base data set for various classifiers that we evaluated. The overall data pipeline is shown in the figure 4.2.

5.3 Experiments

For all experiments, we partitioned data set into training set (70%) and testing set (30%). All models were evaluated for training error, test error and overall error rate. Figure 5.1 shows the principle components of data set using svd method. Here we can observe that data set is densely clustered. Due to this classification problem becomes challenging.

5.4 Malware Detectoion

For malware detection we setup the problem as binary classification problem. All benign applications are labelled as 0 while malicious applications are labelled as 1. We evaluate different models for our dataset which includes Neural Network, SVM (RBF), RandomForest, Decision Trees, AdaBoost and SdA. The results are discussed in Section 6.1.

Table 5.2. Slected Feature Set

_llseek	access	bind	cacheflush
chmod	clock_gettime	clock_nanosleep	clone
close	connect	dup	epoll_ctl
epoll_wait	exit_group	fchown32	fcntl64
fdatasync	flock	fork	fstat64
fsync	futex	getdents64	getpid
getpriority	getsockname	getsockopt	gettid
gettimeofday	getuid32	ioctl	kill
listen	lseek	lstat64	madvise
mkdir	mmap2	mprotect	munmap
nanosleep	open	pipe	poll
prctl	pread	pwrite	read
recvfrom	rename	restart_syscall	rmdir
rt_sigreturn	sched_yield	sendto	setpriority
setsockopt	shutdown	sigprocmask	socket
socketpair	stat64	statfs64	tgkill
umask	uname	unlink	write
writev			

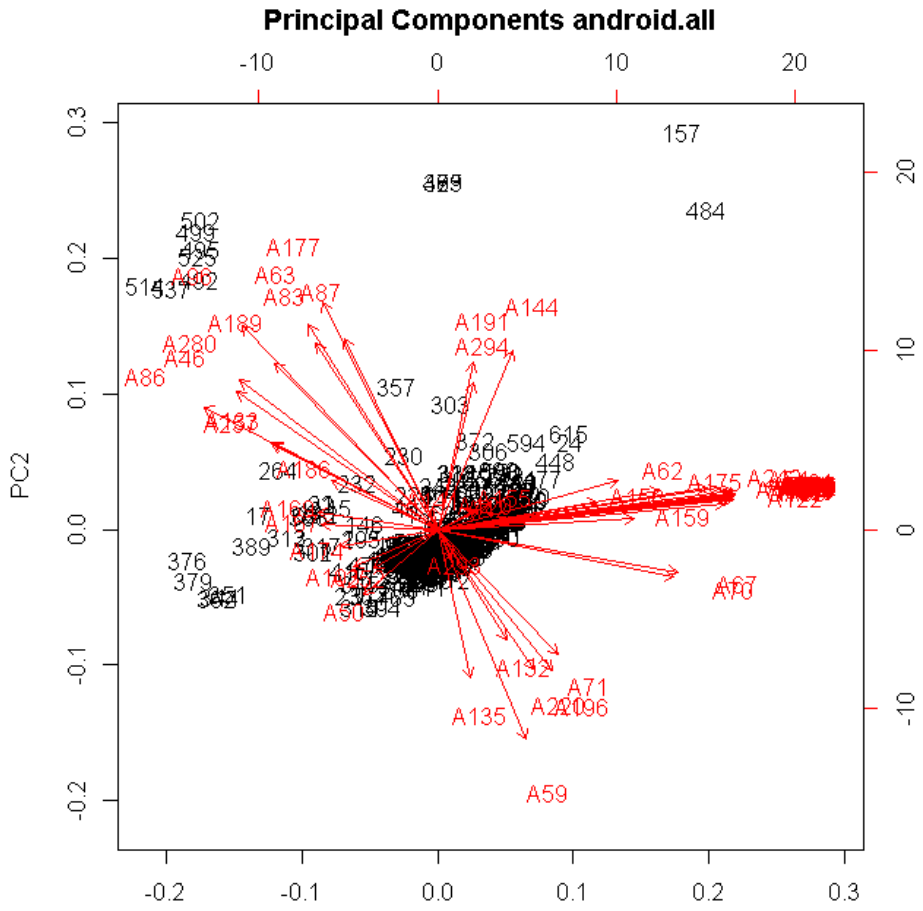


FIG. 5.1. Principle components of data set using SVD method

Table 5.3. Malware Genome Project Dataset

TotalFamilies	38				
TotalSamples	925				
Malware Family	#	Malware Family	#	Malware Family	#
ADRD	22	Asroot	8	BaseBridge	122
BeanBot	8	Bgserv	9	CruseWin	2
DogWars	1	DroidCoupon	1	DroidDeluxe	1
DroidDreamLight	46	DroidKungFu1	34	DroidKungFu2	30
DroidKungFu3	309	DroidKungFu4	96	DroidKungFuUpdate	1
Endofday	1	FakeNetflix	1	FakePlayer	6
GPSSMSSpy	6	GamblerSMS	1	Geinimi	18
GingerMaster	4	GoldDream	47	HippoSMS	4
Jifake	1	LoveTrap	1	Pjapps	58
Plankton	11	RogueLemon	2	RogueSPPush	9
SndApps	10	Spitmo	1	Tapsnake	2
Walkinwat	1	Zitmo	1	Zsone	12
zHash	11				

Table 5.4. Benign Android Applications

com.android.calculator2	com.perunlabs.app.slide
com.transloc.android	com.instagram.android
com.soundcloud.android	com.devuni.flashlight
com.foodspotting	com.google.android.apps.maps
com.fitnesskeeper.runkeeper.pro	net.skyscanner.android.main
com.xe.currency	com.rovio.angrybirds
com.kayak.android	com.digiplex.game
com.fifa.fifaapp.android	com.shazam.android
com.truecaller	edu.umbc.compass
com.example.hackathonproject	com.mobiata.flighttrack.free
com.intsig.camscanner	com.episode6.android.nycsubwaymap
com.zillow.android.zillowmap	com.shazam.android
flipboard.app	

5.4.1 Using SdA

The denoising autoencoders can be stacked to form a deep network by feeding the latent representation (output code) of the denoising auto-encoder found on the layer below as input to the current layer. The unsupervised pre-training of such an architecture is done one layer at a time. Each layer is trained as a denoising auto-encoder by minimizing the reconstruction of its input (which is the output code of the previous layer). Once the first k layers are trained, we can train the k+1-th layer because we can now compute the code or latent representation from the layer below. Once all layers are pre-trained, the network goes through a second stage of training called fine-tuning. Here we consider supervised fine-tuning where we want to minimize prediction error on a supervised task. For this we first add a logistic regression layer on top of the network (more precisely on the output code of the output layer). We then train the entire network as we would train a multilayer perceptron. At this point, we only consider the encoding parts of each auto-encoder. This stage is supervised, since now we use the target class during training. The out of this generates the final model that we would be using for our experiments. Figure 5.2 shows the overall design of our approach for SdA stack.

5.5 Malware Family Classification

For malware family classification we setup the problem as multiclass classification problem. All benign applications are labelled as 'Clean' while malicious applications are labelled as as per there family name. We evaluate different models for our dataset which includes RandomForest, SVM (RBF), AdaBoost and SdA. The results are discussed in Section 6.1.

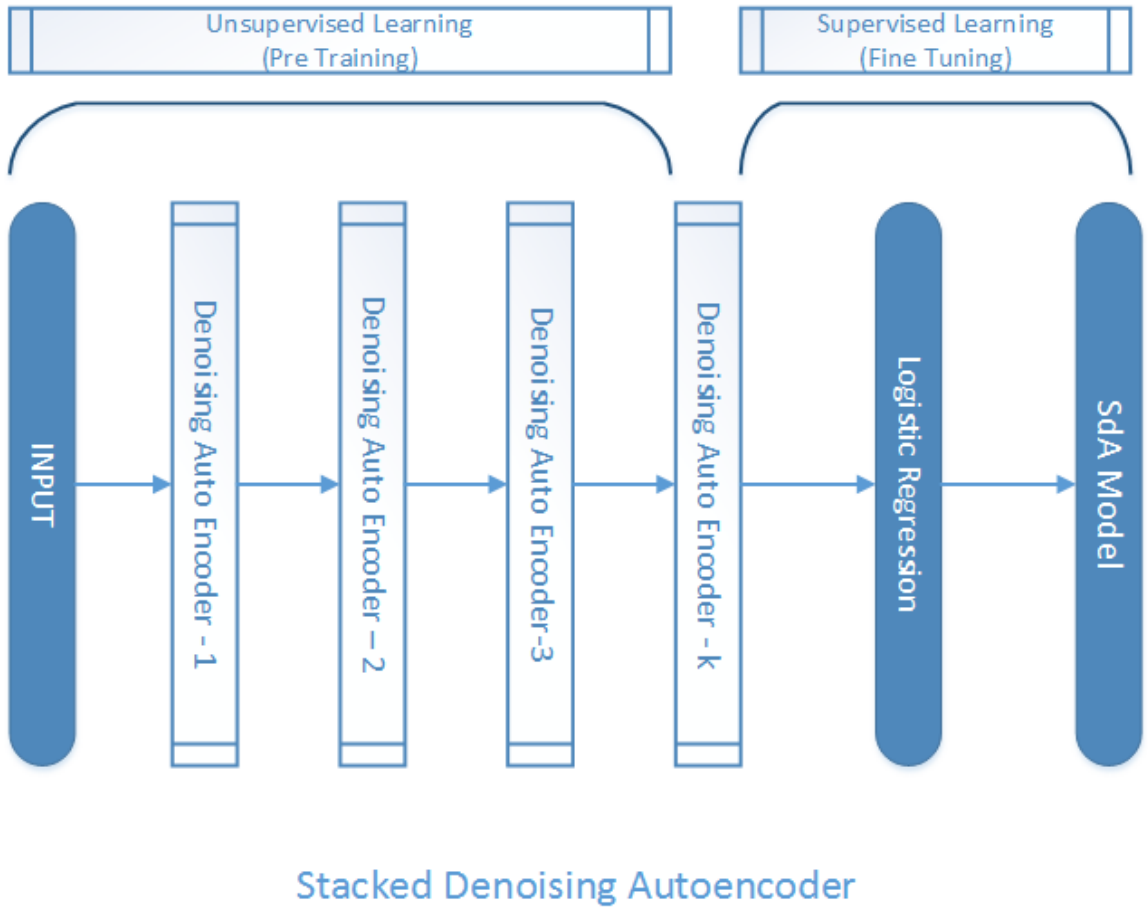


FIG. 5.2. Design of SdA model

5.6 Evaluation Parameters

Here we compare the accuracy of various model for classification problem. We also calculate other The precision, accuracy, specificity, recall, F-measure for the given models are calculated as follows:

$$\mathbf{Precision} = \frac{TP}{(TP + FP)}$$
$$\mathbf{Accuracy} = \frac{TP + TN}{(TP + TN + FP + FN)}$$
$$\mathbf{Recall} = \frac{TP}{(TP + FN)}$$
$$\mathbf{Specificity} = \frac{TN}{(TN + FN)}$$
$$\mathbf{F - measure} = \frac{2 \times Precision \times Recall}{(Precision + Recall)}$$

We also use 'Receiver Operating Characteristic' (ROC) curve which illustrates the performance of a binary classifier system as its discrimination threshold is varied. It is created by plotting the fraction of true positives out of the total actual positives (TPR = true positive rate) vs. the fraction of false positives out of the total actual negatives (FPR = false positive rate), at various threshold settings. TPR is also known as sensitivity or recall.

$$\mathbf{FPR} = \frac{\mathbf{FP}}{(\mathbf{FP} + \mathbf{TP})}$$

Chapter 6

RESULTS

6.1 Malware Detection

Table 6.1. Malware Detection Results for Test Set

Model	Accuracy	Precision	Recall	F-measure	Specificity
SdA	0.996	0.9231	1	0.96	0.9958
Decision Tree	0.972	0.7777	0.5833	0.6666	0.9916
Ada Boost	0.976	0.875	0.5833	0.7	0.9958
SVM rbf	0.972	0.7778	0.5833	0.6666	0.9916
Random Forest	0.98	1	0.5833	0.7368	1
Linear	0.94	0.4211	0.6666	0.5163	0.9538
Neural Net	0.976	0.75	0.75	0.75	0.9874

The results of our experiments are shown in the Table 6.1 and Table 6.2. The SdA model has the highest accuracy rate. The model performance increases with increase in number batch size but after 80 batch size the results are stable and we do not get any improvement with performance. Figure 6.2 shows the performance of SdA model error rate vs batch

Table 6.2. Malware Detection Results for Overall Data Set

Model	Accuracy	Precision	Recall	F-measure	Specificity
SdA	0.9984	0.9629	1	0.9811	0.99834
Decision Tree	0.9809	0.85	0.6538	0.7391	0.995
Ada Boost	0.9904	0.9545	0.8076	0.875	0.99834
SVM rbf	0.9872	1	0.6923	0.8182	1
Random Forest	0.992	1	0.8076	0.8936	1
Linear	0.9729	0.6285	0.8461	0.7213	0.9784
Neural Net	0.9904	0.8846	0.8846	0.8846	0.995

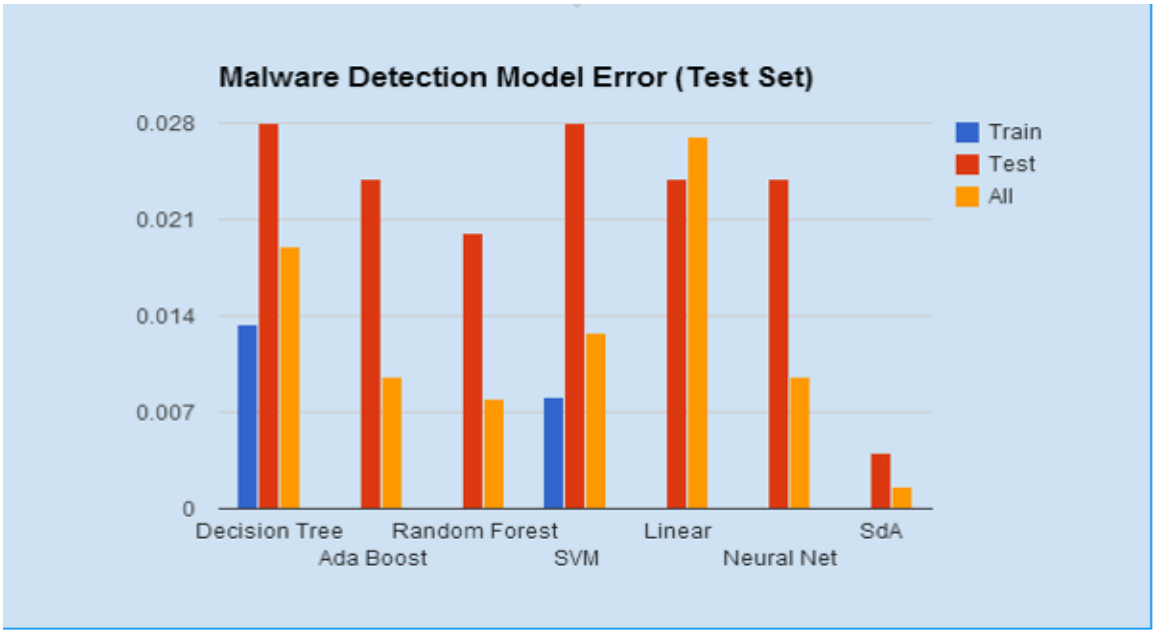


FIG. 6.1. Malware detection model error

size. Using spearmint tool we have found optimal values for SdA model. For pre-training and fine-tuning we use 0.01 as a learning rate. We train our model with 200 pre-training epochs and fine-tuning epochs 100. The Figure 6.1 shows model error observed for various algorithms. The ROC curve analysis testing and overall dataset is shown in Figure 6.3 and Figure 6.4 respectively.

6.2 Malware Family Classification

Figure 6.5 shows the over all results for malware family classification. In general Adaboost model performs better than other models. Model performance is provided in Table 6.3.

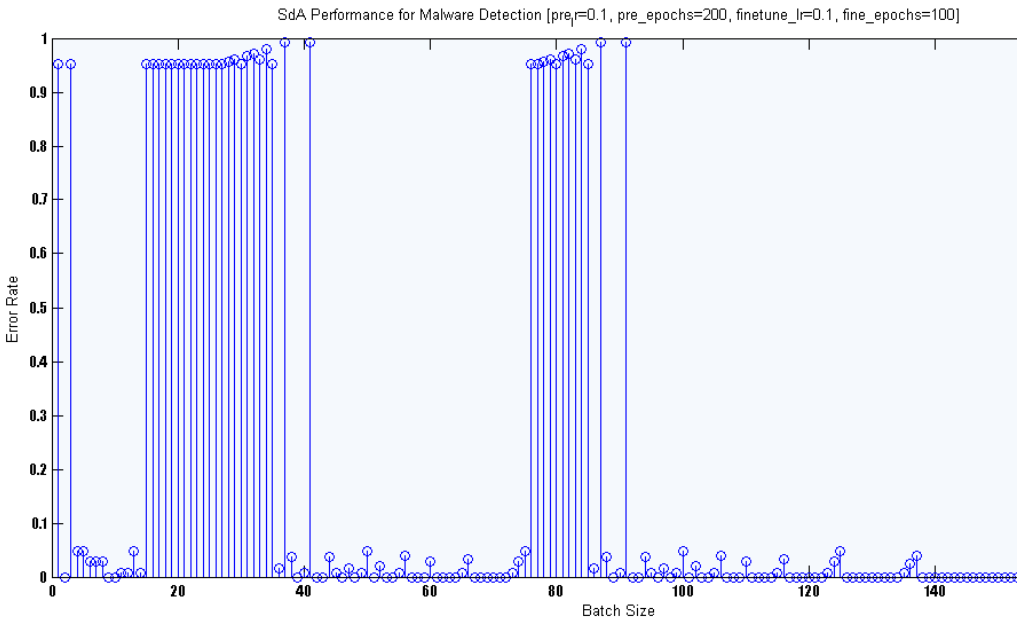


FIG. 6.2. SdA performance Error rate Vs Batch Size

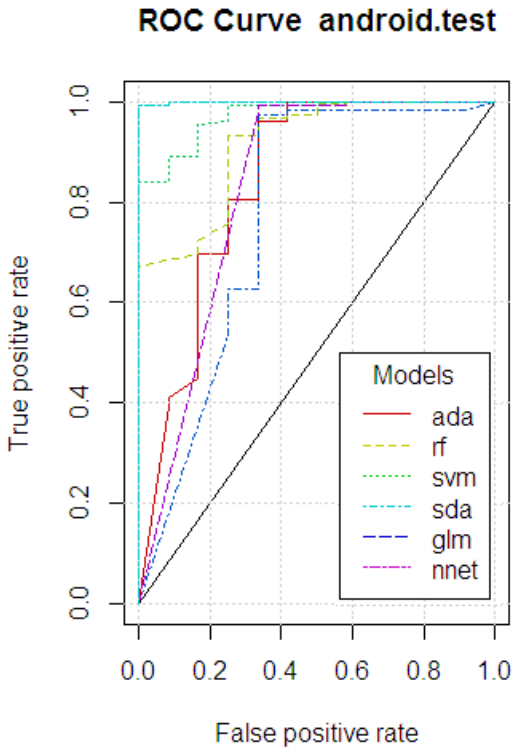


FIG. 6.3. ROC curve analysis for malware detection models using test data set.

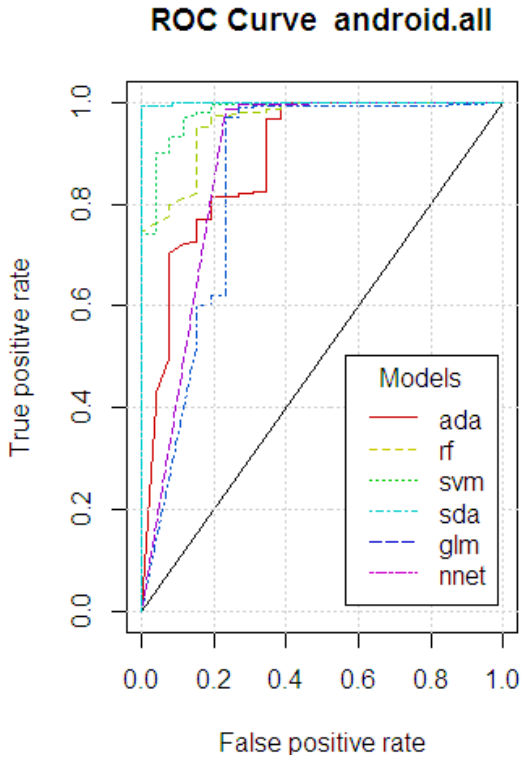


FIG. 6.4. ROC curve analysis for malware detection models using overall data set.

Table 6.3. AdaBoost, Random Forest, SVM , SdA model's accuracy for malware family classification

Model	Train	Test	All
AdaBoost	0.9910525	0.7014421	0.8931112
randomForest	0.9860335	0.6723404	0.8634812
SVM_RBF	0.8324022	0.6340426	0.7576792
SVM_REG	0.9860335	0.4468085	0.7730375
SdA	0.833215	0.6632455	0.7632453

6.2.1 Randomforest

We found that random forest algorithm provides best results for forest of 1500 trees. After that results show no marginal improvements over accuracy. Model performance for testing and overall data set is shown in Figure 6.6 and Figure 6.7 respectively.

6.2.2 AdaBoost

We use AdaBoost to learn strong classifiers form ensembles of weak learners (decision trees), by varying parameters like minleaf value for the decision trees, the number of iterations for the weak classifier learning and the learning rate. Our aim is to maximize the robustness of the classifier by having fewer and more compact trees. Performance of model for testing and overall data set can is shown in Figure 6.9 and Figure 6.10 respectively. The minleaf parameter controls the depth of these tree learners. Our experimental results show that a minleaf value of 30 provides good results for AdaBoost. The boosting model performance increases with increase in number trees but after 50 trees in the results are stable and we do not get any improvement with performance. Figure 6.8 shows the performance of

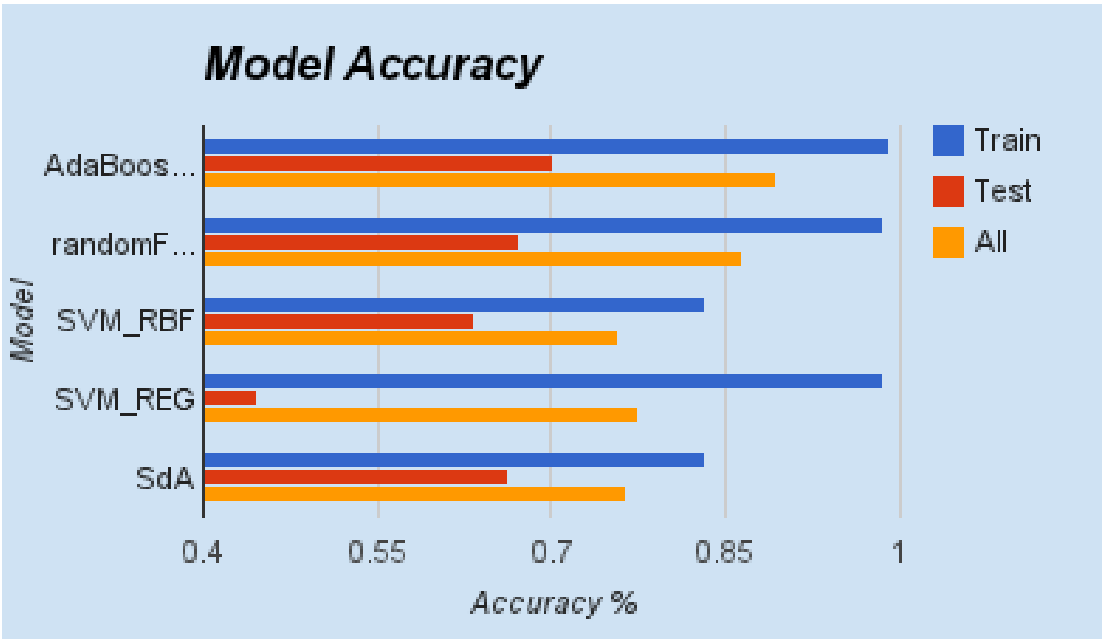


FIG. 6.5. AdaBoost, Random Forest, SVM , SdA model's Accuracy

AdaBoost model error rate vs. number of tree. Overall both random forest and AdaBoost performs better with malware family classification problem.

For AdaBoost algorithm we choose 50 weak learners since minimum number of learners would mean maximum performance. However, experiments are done for different number of learners (n) values - 1, 5, 10, 20, 50 , 100 and 150. It is seen that the values of positive predictive value and accuracy for different number of learners follow an increasing trend for AdaBoost classification. The classifier trees AdaBoost are constructed to see what features are being used while classification. The robustness of the classification techniques is tested by obtaining the features in the classifier trees for different values of n. It is seen that the set of features used by the classifier trees are consistent for the given range of n values. After several runs of experiments we take a suitable value of n = 50 as standard.

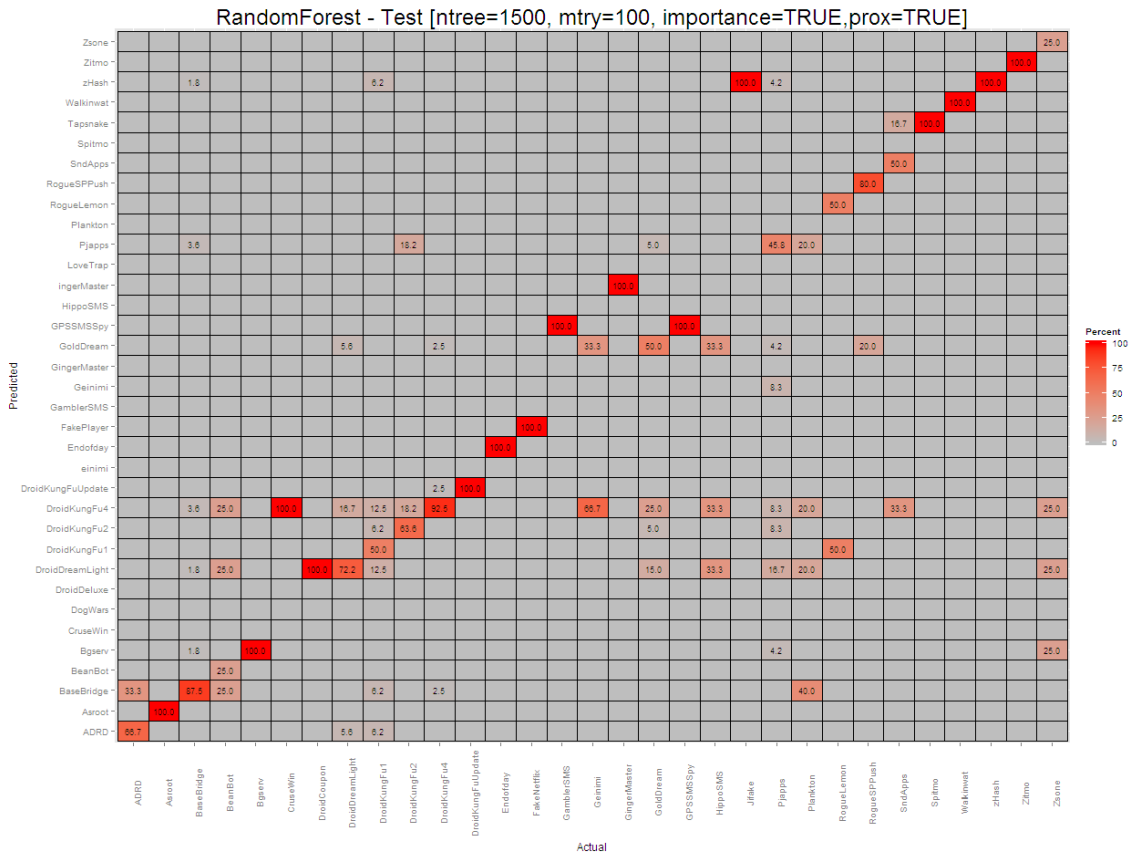
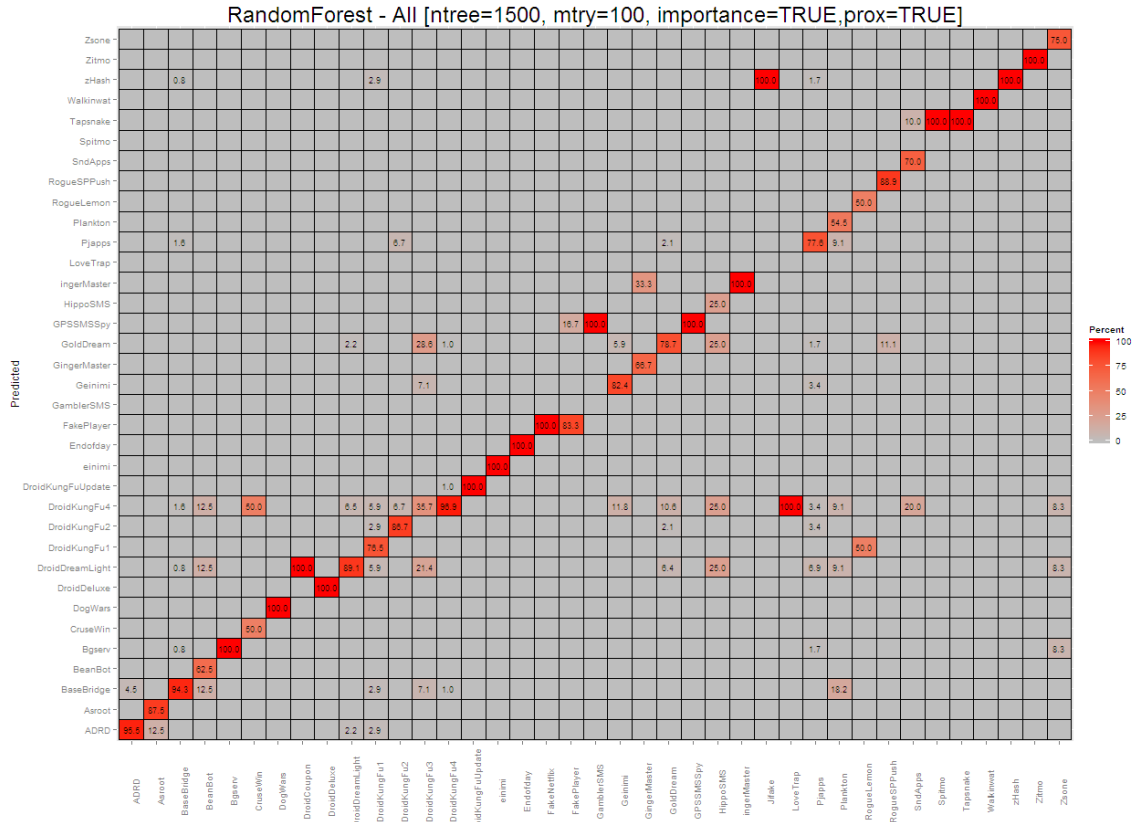


FIG. 6.6. Random forest model performance on test data set [ntree=1500, mtry=100, importance=True, prox=True]



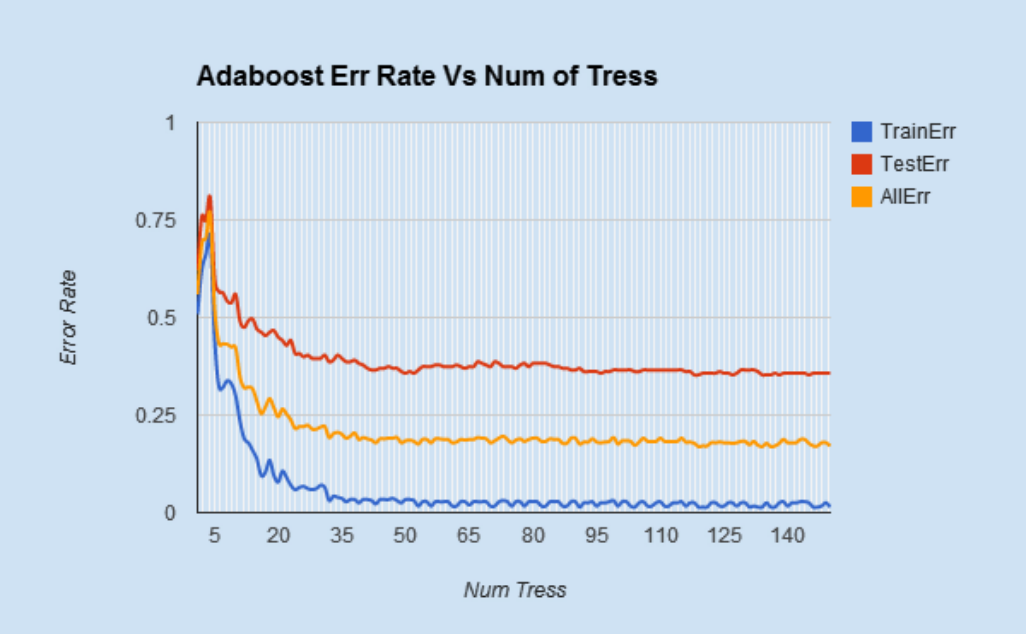


FIG. 6.8. Performance of Adaboost: Error Rate Vs Num of Trees

6.2.3 Support Vector Machine

We also evaluate the performance of SVM for given dataset. We trained our dataset on SVM machine using rbf kernel. SVM model performs better with rbf kernel for binary classification but provides poor results for malware family classification. Model performance for test and overall dataset is shown in Figure 6.11 and Figure 6.12 respectively.

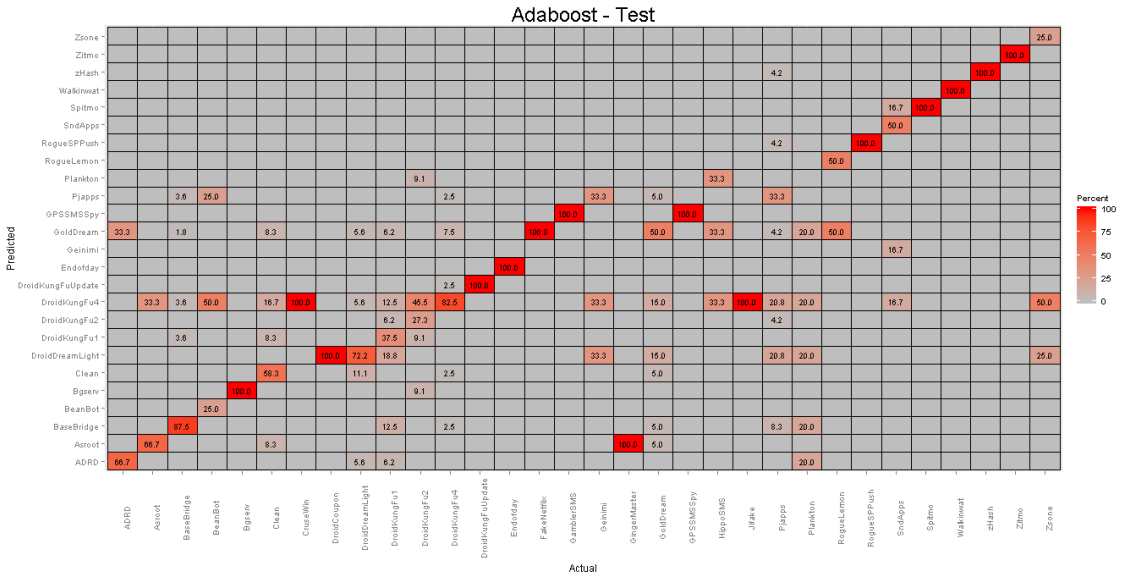


FIG. 6.9. Adaboost model performance on testing data set for malware family classification

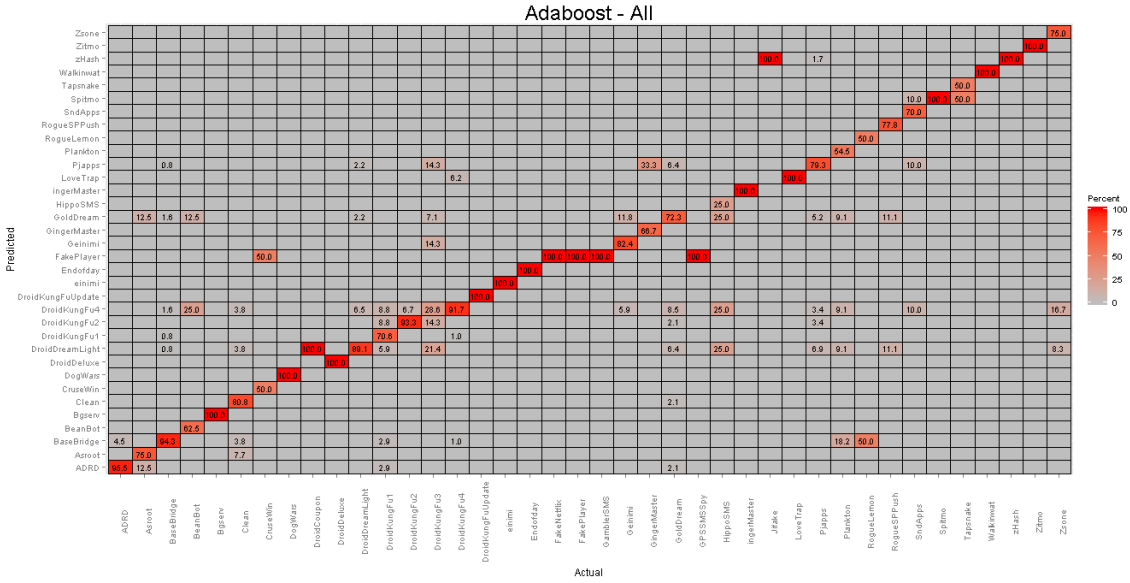


FIG. 6.10. Adaboost model performance on overall data set for malware family classification

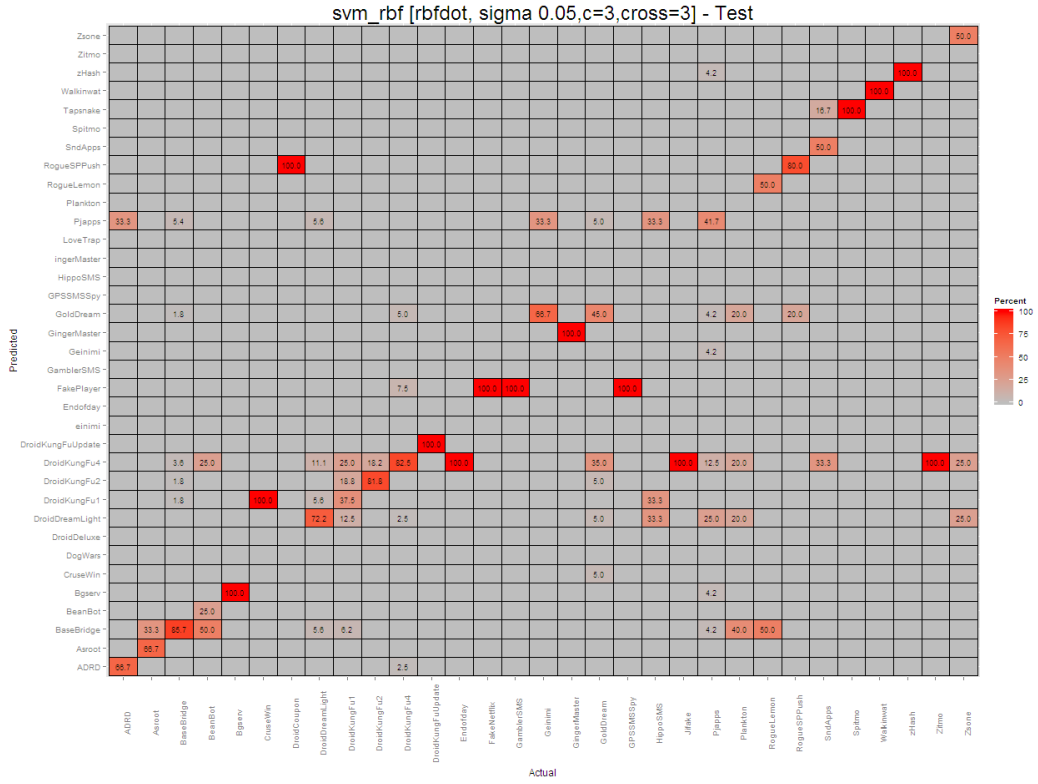


FIG. 6.11. SVM RBF model performance on test data set. [sigma = 0.05, c = 3 , cross = 3]

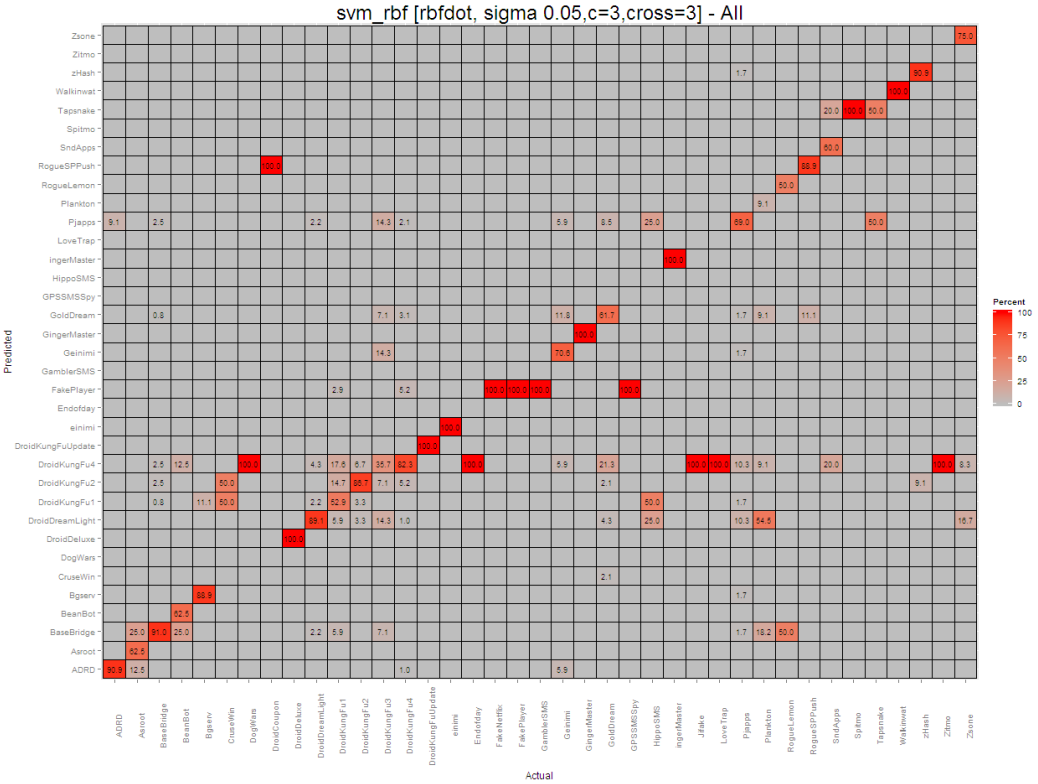


FIG. 6.12. SVM RBF model performance on overall data set. [sigma = 0.05, c = 3 , cross = 3]

Chapter 7

CONCLUSION

Repackaged Android malware can change its code and structure in a way that the signatures of the variants are completely different making the detection of the malware variant very difficult. Though there would be significant code changes due change in victim repackaged application, in general the behavior of the malware and thus its functionality stays the same. We use the distribution of syscalls to detect a malware and categorize it into the specific known malware family. Our experiments indicate that SdA model performs better than rest of the models for malware detection. While random forest and Adaboost algorithm performs better when we try to classify given application to the specific malware family. We also find that SdA algorithm is robust and stable even with corrupted input data.

As part of future work, it would be interesting to study to the performance of SdA using different stacking algorithms.

Appendix A

ANDRIOD SYSTEM CALLS

Table A.1. Andriod System Calls

ARM_NR_cacheflush	ARM_NR_set_tls	__ARM_NR_cacheflush	__ARM_NR_set_tls
__brk	__fcntl	__fcntl64	__fork
__fstatfs64	__getcpu	__getcwd	__getpriority
__ioctl	__llseek	__mmap2	__open
__openat	__ptrace	__reboot	__rt_sigaction
__rt_sigprocmask	__rt_sigtimedwait	__sched_getaffinity	__set_thread_area
__set_tls	__setresuid	__setreuid	__setuid
__sigsuspend	__statfs64	__sys_clone	__syslog
__timer_create	__timer_delete	__timer_getoverrun	__timer_gettime
__timer_settime	__waitid	_exit	_exit_thread
_flush_cache	_llseek	_newselect	_waitpid
accept	access	acct	bind
brk	cacheflush	capget	capset
chdir	chmod	chown	chown32
chroot	clock_getres	clock_gettime	clock_nanosleep
clock_settime	clone	close	connect
creat	delete_module	dup	dup2
epoll_create	epoll_ctl	epoll_wait	eventfd
eventfd2	execve	exit	exit_group
exit_thread	faccessat	fchdir	fchmod
fchmodat	fchown	fchown32	fchownat
fcntl	fcntl64	fdatasync	fgetxattr
flistxattr	flock	flush_cache	fork
fremovexattr	fsetxattr	fstat	fstat64
fstatat	fstatat64	fstatfs64	fsync
ftruncate	ftruncate64	futex	getcpu
getcwd	getdents	getdents64	getegid
getegid32	geteuid	geteuid32	getgid
getgid32	getgroups	getgroups32	getitimer

ioctl	ioprio_get	ioprio_set	kill
klogctl	lchown	lchown32	lgetxattr
link	listen	listxattr	llistxattr
llseek	lremovexattr	lseek	lsetxattr
lstat	lstat64	madvise	mincore
mkdir	mkdirat	mknod	mlock
mlockall	mmap	mmap2	mount
mprotect	mremap	msync	munlock
munlockall	munmap	nanosleep	newselect
open	openat	pause	perf_event_open
personality	pipe	pipe2	poll
prctl	pread	pread64	ptrace
pwrite	pwrite64	read	readahead
readlink	readv	reboot	recvfrom
recvmsg	removexattr	rename	renameat
restart_syscall	rmdir	rt_sigaction	rt_sigprocmask
rt_sigreturn	rt_sigtimedwait	sched_get_priority_max	sched_get_priority_min
sched_getaffinity	sched_getparam	sched_getscheduler	sched_rr_get_interval
sched_setaffinity	sched_setparam	sched_setscheduler	sched_yield
select	sendfile	sendmsg	sendto
set_thread_area	set_tls	seteuid	seteuid32
setgid	setgid32	setgroups	setgroups32
setitimer	setpgid	setpriority	setregid
setregid32	setresgid	setresgid32	setresuid
setresuid32	setreuid	setreuid32	setrlimit
setsid	setsockopt	settimeofday	setuid
setuid32	setxattr	shutdown	sigaction
sigaltstack	signalfd4	sigpending	sigprocmask
sigsuspend	socket	socketpair	stat
stat64	statfs64	swapoff	swapon

REFERENCES

[1] (2014) Number of smartphone users in the u.s. from 2010 to 2018 (in millions). [Online]. Available: http://www.statista.com/statistics/201182/forecast-of-smartphone-users-in-the-us/

[2] (2014) Expected unit shipments of smartphones worldwide (in millions) in 2014 and 2018 by operating systems. [Online]. Available: http://www.statista.com/statistics/309448/global-smartphone-shipments-forecast-operating-system

[3] G. Inc. (2011) android-apktool : a tool to reverse engineer android apk files. [Online]. Available: https://code.google.com/p/androidapktool

[4] F-Secure, "F-secure threat research lab threat report trends for H2 2013," 2014.

[5] "Android developer guide 4.0 r1." [Online]. Available: http://developer.android.com/guide/topics/manifest/manifest-element.html

[6] D. M. Blei, "Probabilistic topic models," *Commun. ACM*, vol. 55, no. 4, pp. 77–84, Apr. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133806.2133826

[7] M. SCHWARTS, "Google removes malware apps from android market." 2011.

[8] N. Christin, A. S. Weigend, and J. Chuang, "Content availability, pollution and poisoning in file sharing peer-to-peer networks," in *Proceedings of the 6th ACM conference on Electronic commerce*. ACM, 2005, pp. 68–77.

[9] J. P. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi, "deseo: Combating search-result poisoning." in *USENIX Security Symposium*, 2011.

[10] T. Moore, N. Leontiadis, and N. Christin, "Fashion crimes: trending-term exploitation on the web," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 455–466.

[11] M. Schönefeld, "Reconstructing dalvik applications," in *10th annual CanSecWest conference*, 2009.

[12] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security." in *USENIX security symposium*, vol. 2, 2011, p. 2.

[13] "android4me: J2ME port of Google's Android," 2011. [Online]. Available: https://code.google.com/p/android4me/downloads/list

[14] Y. Freund and R. E. Schapire, "A short introduction to boosting," in *In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1999, pp. 1401–1406.

[15] Yoav Freund and Robert E. Schapire, *Boosting: Foundations and Algorithms.*, 1st ed. MIT Press, 2012.

[16] Y. Bengio, "Learning deep architectures for AI," *Foundations and trends in Machine Learning*, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1658424

[17] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*. IEEE, 1996, pp. 120–128.

[18] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*. IEEE, 1999, pp. 133–145.

[19] I. Burguera, U. Zurutuza, and S. 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*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046619

[20] G. Creech and J. Hu, "A semantic approach to host-based intrusion detection systems using contiguous and discontiguous system call patterns," 2013.

[21] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the detection of anomalous system call arguments," in *Computer Security–ESORICS 2003*. Springer, 2003, pp. 326–343.

[22] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis," *IEEE Transactions on Dependable and Secure Computing (TODS)*, vol. 7, no. 4, p. 381395, Nov. 2008.

[23] L. Girardin and D. Brodbeck, "A visual approach for monitoring logs," in *Proceedings of the 12th USENIX Conference on System Administration*, ser. LISA '98. Berkeley, CA, USA: USENIX Association, 1998, pp. 299–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1037990.1038025

[24] P. Lichodzijewski, A. N. Zincir-Heywood, and M. I. Heywood, "Host-based intrusion detection using self-organizing maps," in *IEEE international joint conference on neural networks*, 2002, pp. 1714–1719.

[25] D. Gao, M. K. Reiter, and D. Song, "Behavioral distance for intrusion detection," in *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 63–81. [Online]. Available: http://dx.doi.org/10.1007/11663812_4

[26] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools." in *NDSS*, vol. 3, 2003, pp. 163–176.

[27] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: Versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 347–356. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920313

[28] J. Cheng, S. H. Wong, H. Yang, and S. Lu, "Smartsiren: Virus detection and alert for smartphones," in *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, ser. MobiSys '07. New York, NY, USA: ACM, 2007, pp. 258–271. [Online]. Available: http://doi.acm.org/10.1145/1247660.1247690

[29] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *Proceedings of the 2011 Seventh International Conference on Computational Intelligence and Security*, ser. CIS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1011–1015. [Online]. Available: http://dx.doi.org/10.1109/CIS.2011.226

[30] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu, "pbmds: A behavior-based malware detection system for cellphone devices," in *Proceedings of the Third ACM Conference on Wireless Network Security*, ser. WiSec '10. New York, NY, USA: ACM, 2010, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/1741866.1741874

[31] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, ser. SP '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 231–245. [Online]. Available: http://dx.doi.org/10.1109/SP.2007.17

[32] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[33] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 351–366. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855790

[34] J. Oberheide, E. Cooke, and F. Jahanian, "Cloudav: N-version antivirus in the network cloud," in *Proceedings of the 17th Conference on Security Symposium*, ser. SS'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 91–106. [Online]. Available: http://dl.acm.org/citation.cfm?id=1496711.1496718

[35] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets." in *NDSS*, 2012.

[36] P. Szor, *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[37] A. Somayaji and S. Forrest, "Automated response using system-call delays," in *Proceedings of the 9th Conference on USENIX Security Symposium - Volume 9*, ser. SSYM'00. Berkeley, CA, USA: USENIX Association, 2000, pp. 14–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251306.1251320

[38] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred, "Statistical approaches to ddos attack detection and response," in *DARPA Information Survivability Conference and Exposition, 2003. Proceedings*, vol. 1. IEEE, 2003, pp. 303–314.

[39] I. Balepin, S. Maltsev, J. Rowe, and K. Levitt, "Using specification-based intrusion detection for automated response," in *Recent Advances in Intrusion Detection*. Springer, 2003, pp. 136–154.

[40] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '10. New York, NY, USA: ACM, 2010, pp. 179–194. [Online]. Available: http://doi.acm.org/10.1145/1814433.1814453

[41] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, 1st ed. San Francisco, CA, USA: No Starch Press, 2012.

[42] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944937

[43] S. Attaluri, S. McGhee, and M. Stamp, "Profile hidden markov models and metamorphic virus detection," *Journal in Computer Virology*, vol. 5, no. 2, pp. 151–169, 2009.

[44] F. Leder, B. Steinbock, and P. Martini, "Classification and detection of metamorphic malware using value set analysis," in *MALWARE*, 2009, pp. 39–46.

[45] S. K. Agarwal and V. Shrivatsava, "An opcode statistical analysis for metamorphic malware," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, Oct. 2013.

[46] M. Steyvers and T. Griffiths, *Probabilistic Topic Models*. In T. Landauer, D McNamara, S. Dennis, And W. Kintsch, 2006, ch. Latent Semantic Analysis: A Road to Meaning. Laurence Erlbaum.

[47] W. Wong and M. Stamp, "Hunting for metamorphic engines," *Journal in Computer Virology*, vol. 2, no. 3, pp. 211–229, 2006.

[48] D. Lin and M. Stamp, "Hunting for undetectable metamorphic viruses," *J. Comput. Virol.*, vol. 7, no. 3, pp. 201–214, Aug. 2011. [Online]. Available: http://dx.doi.org/10.1007/s11416-010-0148-y

[49] A. Govindaraju, "Exhaustive statistical analysis for detection of metamorphic malware," Master's projects, Paper 66, 2010.

[50] P. Szr and P. Ferrie, "Hunting for metamorphic," in *In Virus Bulletin Conference*, 2001, pp. 123–144.

[51] N. Runwal, R. M. Low, and M. Stamp, "Opcode graph similarity and metamorphic detection," *Journal in Computer Virology*, vol. 8, no. 1-2, pp. 37–52, 2012.

[52] S. Coorp, "Symantec internet security threat report," 2009.

[53] "Vx heavens," http://www.vx.netlux.org/.

[54] "Virus total," http://www.virustotal.com/.

[55] "Plate notation for lda," http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation/.

[56] J. Aycock, *Computer Viruses and Malware (Advances in Information Security)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[57] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 12–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251365

[58] C. Seiffert, T. M. Khoshgoftaar, J. V. Hulse, and A. Napolitano, "Rusboost: Improving classification performance when training data is skewed," in *ICPR*. IEEE, 2008, pp. 1–4.

[59] *19th International Conference on Pattern Recognition (ICPR 2008), December 8-11, 2008, Tampa, Florida, USA*. IEEE, 2008.

[60] B. Bashari Rad, M. Masrom, S. Ibrahim, and S. Ibrahim, "Morphed virus family classification based on opcodes statistical feature using decision tree," in *Informatics Engineering and Information Science*, ser. Communications in Computer and Information Science, A. Abd Manaf, A. Zeki, M. Zamani, S. Chuprat, and E. El-Qawasmeh, Eds. Springer Berlin Heidelberg, 2011, vol. 251, pp. 123–131. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25327-0_11

[61] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion," *J. Mach. Learn. Res.*, vol. 11, pp. 3371–3408, Dec. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1953039

[62] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *Proceedings of the 14th conference on USENIX Security Symposium-Volume 14*. USENIX Association, 2005, pp. 11–11.

[63] Y. Zhou and X. Jiang, "Android malware genome project," *Disponibile a http://www. malgenomeproject. org*, 2012.

[64] ASRAR, “Fake netflix app,” oct 2011.

[65] T.-H. Ho, D. Dean, X. Gu, and W. Enck, “Prec: Practical root exploit containment for android devices,” in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, ser. CODASPY ’14. New York, NY, USA: ACM, 2014, pp. 187–198. [Online]. Available: http://doi.acm.org/10.1145/2557547.2557563