TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Sicherheit in der Informatik

Machine Learning for Anomaly Detection under Constraints

Bojan Kolosnjaji

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr. Jens Grossklags

Prüfer der Dissertation:

1. Prof. Dr. Claudia Eckert
2. Prof. Dr. Apostolis Zarras

Die Dissertation wurde am 04.06.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 22.10.2019 angenommen.

Acknowledgements

First of all, I would like to thank my family for their support throughout my studies. With their help and motivating words I was able to persevere with my research work despite difficult moments during this journey. Next, I would like to thank my supervisor, Prof. Dr. Claudia Eckert, for the opportunity to work at the Chair of IT Security and for advice on my research work and thesis writing. Furthermore, in my work I benefited from the cooperative and supportive colleagues and students from the Chair. Special thanks to Prof. Dr. Apostolis Zarras for closely following my research, for collaboration and useful advice. Also, I would like to thank George Webster for collaboration and leading the development of the *Malware Zoo* of our Chair, which was a crucial source of data for my experiments. Last but not least, I would also like to thank my collaborators from University of Cagliari, Prof. Dr. Giorgio Giacinto and Prof. Dr. Battista Biggio, for hosting me at their Lab and giving me an opportunity to work with their group.

Abstract

Detection of anomalous events is an important problem in data processing. The anomalies need to be detected not only for the purpose of noise cancellation but also because they give more information for system diagnostics. For example, security-related anomalies give us information about potential malicious functionality in a computer system. These anomalies can very often be effectively detected with a machine learning approach, where we use available methods ranging from unsupervised learning to classification-based approaches to construct detection models. Machine learning methods provide a promising solution for anomaly detection since they enable us to create predictive statistical models for detection and differentiation of anomalies based on data. However, these methods do not naturally support optimal functionality for scenarios with constraints imposed by the environment and limited resources.

Motivated by the lack of such approaches, we investigate scenarios where anomaly detection can be improved for constrained environments. This thesis contains the methodology that we developed for these scenarios and the results that we obtained. The approaches for the particular, primarily IT security-related, scenarios are divided into multiple chapters. First, we improve the topic-modeling-based malware variant detection with an approach for an environment with a high influx of malware samples. Our approach includes solutions for constraints on modeling choices where the model needs to be frequently retrained. In the next chapter, we investigate the robustness of neural-network-based malware detection using static malware features. These detection methods need to be applicable in the environment with random and adversarial noise, because of the variability in the malware code and possible evasion attempts. In the third scenario, we investigate unsupervised anomaly detection under the limited size of the parameter set for the purpose of sensor-based authentication on mobile devices. Furthermore, we investigate the constraints in data collection for sequence-aware behavioral malware analysis. The last chapter introduces the notion of communication-efficient learning for anomaly detection, where we design and test a model for budget-constrained learning from multiple experts.

Zusammenfassung

Ein grundlegendes Problem in der Datenverarbeitung ist das Erkennen von ungewöhnlichen Ereignissen. Anomalien müssen nicht nur zum Zwecke der Rauschunterdrückung identifiziert werden, sondern auch weil sie umfassende Informationen für eine Systemdiagnose liefern. Zum Beispiel können sicherheitsrelevante Anomalien Informationen über potenziell schädliche Funktionen in einem System darstellen. Solche Anomalien können wirkungsvoll mittels Verfahren des maschinellen Lernens erkannt werden. Verfahren des maschinellen Lernens sind effektive Maßnahmen, weil sie es uns ermöglichen, vorhersagbare statistische Modelle zur Erkennung und Differenzierung von Anomalien basierend auf Daten zu nutzen. Jedoch sind diese Verfahren häufig wenig geeignet in Szenarien, die speziellen Einschränkungen unterliegen, die durch die Umgebung oder durch beschränkte Ressourcen bestimmt werden.

In dieser Arbeit werden solche Szenarien untersucht, um verbesserte Verfahren zur Anomalieerkennung für den Einsatz in Umgebungen, in welchen Ressourcenbeschränkungen vorherrschen, zu entwickeln. Unsere Lösungsansätze sind in erster Linie auf Szenarien aus dem Bereich der IT-Sicherheit zugeschnitten und werden in verschiedenen Kapiteln getrennt behandelt. Zunächst verbessern wir Topic-Modelling-basierte Malware-Erkennung in Umfeldern mit hohem Aufkommen neuer Malware-Exemplare. Hier müssen Einschränkungen beim Bestimmen der Modellparameter hingenommen werden, weil deren Anpassung kontinuierlich über die Lebenszeit des Modells hinweg erfolgen muss. Danach betrachten wir die Robustheit solcher statischen Malware-Erkennungsverfahren, welche auf neuronalen Netzwerken beruhen. Diese Erkennungsverfahren müssen aufgrund einer Vielzahl von unterschiedlichen Malware-Implementierungen und deren Laufzeitumgebungen in der Lage sein sowohl mit stochastischem als auch mit bösartig induziertem Rauschen umzugehen. Des Weiteren untersuchen wir die nicht-überwachte Anomalieerkennung für sensorbasierte Authentifikation auf mobilen Endgeräten, wobei die Menge von Modellparametern eingeschränkt bleibt. Darüber hinaus betrachten wir Randbedingungen in Bezug auf Datengewinnung bei der Analyse des sequenziellen Verhaltens von Malware. Das letzte Kapitel stellt das Konzept des kommunikationseffizienten Lernens zur Anomalieerkennung vor, indem wir ein von mehreren Experten lernendes Modell mit einem eingeschränkten Budget konzipieren und testen.

Contents

List of Figures

List of Tables

Publications

Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *26th European Signal Processing Conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

Bojan Kolosnjaji, Antonia Hüfner, Claudia Eckert, and Apostolis Zarras. Learning on a budget for user authentication on mobile devices. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2018.

Benjamin Taubmann and Bojan Kolosnjaji. Architecture for resource-aware vmi-based cloud malware analysis. In *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*, pages 43–48. ACM, 2017.

Bojan Kolosnjaji, Ghadir Eraisha, George Webster, Apostolis Zarras, and Claudia Eckert. Empowering convolutional networks for malware classification and analysis. In *International Joint Conference on Neural Networks (IJCNN)*, pages 3838–3845. IEEE, 2017.

Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, pages 137–149. Springer, Cham, 2016.

Bojan Kolosnjaji, Apostolis Zarras, Tamas Lengyel, George Webster, and Claudia Eckert. Adaptive semantics-aware malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 419–439. Springer, Cham, 2016.

Published work out of scope of the dissertation:

Noëlle Rakotondravony, Benjamin Taubmann, Waseem Mandarawi, Eva Weishäupl, Peng Xu, Bojan Kolosnjaji, Mykolai Protsenko, Hermann De Meer, and Hans P Reiser. Classifying malware attacks in iaas cloud environments. *Journal of Cloud Computing*, 6(1):26, 2017.

George D Webster, Bojan Kolosnjaji, Christian von Pentz, Julian Kirsch, Zachary D Hanif, Apostolis Zarras, and Claudia Eckert. Finding the needle: A study of the pe32 rich header and respective malware triage. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 119–138. Springer, 2017.

Andreas Fischer, Thomas Kittel, Bojan Kolosnjaji, Tamas K Lengyel, Waseem Mandarawi, Hermann de Meer, Tilo Müller, Mykola Protsenko, Hans P Reiser, Benjamin Taubmann, et al. Cloudidea: A malware defense architecture for cloud data centers. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 594–611. Springer, Cham, 2015.

Bojan Kolosnjaji and Claudia Eckert. Neural network-based user-independent physical activity recognition for mobile devices. In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 378–386. Springer, 2015.

Chapter 1

Introduction

Anomaly detection is an area of research that encompasses approaches to find patterns in data that differ from the expected behavior. The detection of anomalous behavior is important in many scenarios, such as malware or cyberattack detection, fault detection, surveillance, medical image analysis. Apart from rule-based approaches of examining the conformance of data against a set of predefined constraints, statistical machine learning methods have gained importance for solving anomaly detection problems, since they can be more accurate in a noisy environment. Learning-based methods enable us to train anomaly detection procedures based on capturing statistically relevant features for differentiation between expected and outlier data points and structures. While machine learning-based methods are used in many currently state-of-the-art anomaly detection systems, in their baseline versions these methods are not adapted to the scenarios where one needs to leverage the potential of learning from data, but also preserve this functionality and algorithm performance in the presence of various constraints. In this work, we walk through multiple use case scenarios with various types of constraints imposed by the conditions where machine learning methods are applied for detecting outlier patterns. We show in these scenarios that anomaly detection can be used under limiting conditions as well and explain adaptations to machine learning methods that enable high performance in detection and classification of anomalous events.

1.1 Motivation

There are many application areas where it is useful to detect and analyze anomalies in data. Outliers in measurements made by industrial system monitoring devices indicate possible faults or defects, therefore early detection of anomalies is important for system reliability. Anomalous values produced by a home sensor network can indicate malfunction of appliances. Outliers in medical analysis data, such as EEG signals and X-ray images indicate possible health issues. The detection of these outliers is very important for early detection of various medical problems. Fraud detection in various areas, such

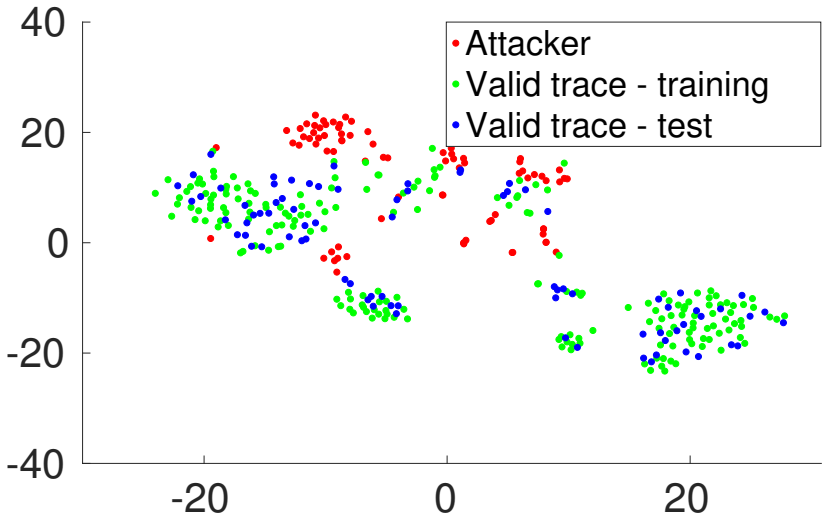


Figure 1.1: User authentication using an anomaly detection approach

as insurance industry, mobile communication and credit card payment can be improved by defining and solving these problems as anomaly detection. The area of computer security has a high potential benefit from anomaly detection, as malicious programs and cyberattacks can be considered as anomalies with respect to the benign programs and activity in information systems. The structure of malicious programs is different from benign programs, since the attackers need to include malicious functionality in their code. Furthermore, observed behavior of malicious programs is different, since the goal differs from the benign activity. Anomaly detection methods can also be used for authentication, where we differentiate invalid users, i.e. attackers as anomalies w.r.t. the valid users. An illustration of an anomaly detection test result is shown on Figure 1.1. This figure contains an example of measurements for sensor-based authentication on a mobile device projected to a 2D graph, where measured values for an attacker differ from the footprint of a valid user.

The earliest systems for detecting anomalies relied on a predefined set of rules that enabled fast check if the data complies with the expected normal behavior. However, these systems are not always sufficient in the presence of statistical noise, where it is difficult to define rules that cover the necessary patterns for anomaly detection. Rigidly defined systems contain clear properties assumed for regular data, but are not flexible enough for more complex problems where differentiation between normal and outlier behavior is more subtle.

Learning-based approaches are an important step to solve this problem. We live in the age of big data, where it is easier to gather large amounts of measurements that characterize the functionality of the system under examination. Therefore there is a need to design statistical methods that leverage the vast amounts of data for modeling

the behavior of system under examination, in order to better differentiate between usual and anomalous measurements.

However, training of machine learning algorithms can require vast amount of different resources and the resulting systems need to operate in a wide range of working conditions. Systems for behavioral monitoring can require storage of measurements for a large number of events. Nonparametric machine learning methods require a parameter set to be frequently enlarged and updated. Complex machine learning training algorithms need vast computational resources for the optimization of model parameters. Data used to train and test machine learning methods can have limited reliability and even be manipulated to an extent by an adversary. These and many other constraints impose a need for adaptation of anomaly detection methods to these conditions.

We consider multiple scenarios where this kind of constraints can appear. These example scenarios are described in the following subsections.

1.1.1 Malware Detection and Classification

Malware detection is becoming an increasingly important problem, as growing number and variety of malicious executables is reported by security companies. These companies currently need to analyze hundreds of thousands of unique malicious samples on a daily basis, which directly affects their performance. For instance, Virustotal Statistics [4] reports up to million of malware samples detected per day. Rule-based detection approaches are not adequate to tackle this problem, as new malware instances integrate sophisticated techniques to deceive rigid detectors and proceed with their functionality without being detected. Static rule-based signatures and even heuristics generated by manual reverse engineering are seriously challenged by the possibility to easily create a large number of new malware samples and circumvent these detectors.

Antivirus and other security companies can obtain large numbers of malicious and benign samples. However, accurate analysis and classification can be difficult and time-consuming. One approach to automate this analysis of a large number of samples and detect novel malware samples is to use machine learning methods. For instance, Schultz et al. use n-gram based approach to build a machine learning-based malware detector [107]. On the other hand, Rieck et al. [97] use data from behavioral monitoring to create a machine learning model for classification of malware.

Although it has been proven that machine learning provides improvements in detection and classification of malware, systems that include machine learning are not without their own problems. For instance, we need to consider issues such as lack of labeled data for training of supervised models and heterogeneity of input data coming from static and dynamic malware analysis tools for machine learning models. However, there is also a problem that we need to cope with various constraints when constructing machine learning models. For instance, in case of gathering data for dynamic analysis for a large number of malware samples, we need to take care of resource constraints in data acquisition. Many times we need to update our model to include the new malware

variants, which incurs a need for scalable online learning models. Furthermore, we need to cope with the possibility that adversaries might actively attempt to circumvent our detection model. An adversary can find a way to circumvent the detection by repeatedly creating new malware samples and querying our system in order to achieve desired detection or classification result. We need to evaluate our detector for adversarial environment and find robust methods that prevent this kind of attacks.

1.1.2 Smartphone User Authentication

Smartphones have become multifunctional communication tools used by a large and growing amount of people. As the number and sophistication of smartphone applications grows, users are tempted to store more sensitive information, making security and privacy an increasingly important issue. While smartphones are easy to carry around, they are also prone to getting lost or stolen. If a criminal is able to unlock a stolen phone, he can potentially access a breadth of personal information, such as e-mail accounts, private photos, or credit card information.

There are already authentication methods for smartphones, such as: PINs, swipe patterns, picture codes, and fingerprint sensors. However, these entry authentication options can be inconvenient, especially if a user needs to use them repeatedly before executing short actions, for example to answer messages. As a result, users tend to prefer weak passwords, choose highly increased lock periods, or leave their phones completely unlocked. There is also research work that shows existing vulnerabilities of entry authentication methods. For instance, PINs and swipe patterns are vulnerable to both smudge attacks [13] and shoulder surfing [105], while fingerprint sensors may be tricked by fingerprint copies [45].

One approach that was proposed as an alternative is continuous sensor-based authentication. In this type of authentication, we do not only authenticate a user at the start of his activity. Instead, the user behavior is continuously monitored in order to detect deviations that would indicate activities of an unauthorized user. The idea is to not interrupt the user unnecessarily, but to do anomaly detection in the background, while the user is executing the usual activities on the smartphone. If a large deviation from usual activity or sensor value patterns is found, the smartphone can be locked and further actions can be taken to secure the phone and the data stored on it. Using continuous authentication and capturing user-specific patterns we should be able to reliably protect the phone content, as user behavior has been proven difficult to mimic [20]. These behavioral patterns can be detected using machine learning methods. Based on data gathered from the smartphone user and an anomaly detection approach, we can continuously protect the device.

However, having such functionality in the background requires more resources, which could present a problem on devices with limited processing power, memory or battery capacity. Therefore solving this type of problem would benefit from resource-aware anomaly detection approaches.

1.1.3 Crowdsourcing

Supervised machine learning depends on our ability to gather sufficiently large sets of labeled data. In a high number of scenarios, we can benefit from labeled data gathered by querying multiple separate sources with varying reliability and join this data into one machine learning model. However, many times we are limited in the number of data sources that we are allowed to use, because of budget constraints, limited processing power or insufficient communication bandwidth. This causes the need for a framework that would enable resource-efficient combination of labeled data from multiple sources.

An important fitting scenario to this framework is crowdsourcing. When using crowdsourcing systems such as Amazon Mechanical Turk, we can gather labeled data from multiple annotators. After gathering this training data, we need to join the labeling decisions and train one machine learning system while leveraging all of the annotations. Relying on a concept of *wisdom of crowds* [113], we could assume that if we gather data from a sufficient number of people, on average most of them will give correct labeling decision. However, gathering data from a large amount of annotators can be expensive, as workers on Amazon Mechanical Turk require payment for their work. Therefore we would benefit from a procedure of prior determination of the annotators' expertise. In this case, the joining process can be optimized by estimating the expertise of the annotators, selecting the optimal subset and determining weights of the annotators' labels. Moreover, in a rising area of *Internet of Things* (IoT) we can have similar problems. In IoT systems, a large number of sensors provide information about the environment. Communication among multiple sensors can be costly in terms of processing power. Furthermore, this communication can be done among sensors that are limited with power consumption constraints. In intrusion detection it is useful to use results from multiple anomaly detection systems. For example, although there are multiple antivirus programs that can label software as benign or malicious programs of certain type, only a minority of the programs contain updated labels for newest malware families and classify malware with high granularity. Therefore it would be useful to find a subset of label sources that are reliable enough to be used for malware triage purposes.

1.2 Problem Statement

Relying on these scenarios, we define our anomaly detection constraints. Our goal is to model the constraints imposed by the environment in which the machine learning methods for anomaly detection are employed and to investigate the possibilities of adapting these methods for such constrained environment. We take special consideration for problems that appear in scenarios related to computer security, where anomaly detection can be helpful to uncover malicious activity. The problems of this type provide a motivation to design approaches for anomaly detection under constraints.

In particular, we consider the following research problems:

1. Semi-supervised malware variant detection using adaptive topic models

   How to enable nonparameteric topic modeling methods for adaptive classification of malware samples and detect samples from both predefined and previously unknown malware families?

2. Detection of malicious binaries in an adversarial environment

   How do the methods of detecting malicious binaries based on neural networks work under constraints of data with adversarial noise?

3. Online mobile user authentication with limited modeling budget

   How to construct kernel-based mobile user authentication systems under computational resource constraints?

4. Communication-efficient learning from multiple annotators

   Can we leverage statistical modeling of data labelers during training of machine learning models to select a small number of reliable labelers, optimize the method for combining their results during training and reduce communication needed for the testing phase?

5. Anomaly detection with constraints on data acquisition

   Can we use machine learning to select events significant for neural network-based sequence modeling for anomaly detection and reduce the overhead of large-scale data collection?

This is not a complete set of constrained anomaly detection problems, as many other possible constraints, such as energy, data availability, or execution time are not separately considered. However, we address many important scenarios and set a foundation for additional solutions. Solving such particular research questions can give us the direction on how to design anomaly detection methods for constrained environments.

1.3 Organization

Chapter 2 contains an introductory text, consisting of basic information about anomaly detection approaches, with a survey of concepts and methods from machine learning that are important to understand the further chapters. In the Chapter 3 we describe the use of topic models under nonparametric learning and semi-supervised learning approach for detection and analysis of malicious program behavior. Chapter 4 contains the analysis of detection of malicious programs using convolutional neural networks under noisy and adversarial environment. In the Chapter 5 we consider constraints imposed on behavior-based authentication on mobile devices and we describe our approach using constrained unsupervised anomaly detection. Chapter 6 contains an approach to include

data acquisition constraints in the detection model for anomalous system call sequences. Communication constraints when gathering labels from distributed annotators constitute a central topic of Chapter 7. Finally, Chapter 8 contains final remarks and brings an outlook for future work in this area.

Chapter 2

Background

In this chapter we describe basic terms and concepts related to the thesis topic and broad area where it belongs. The background knowledge from this chapter is crucial for understanding the content in the later chapters.

2.1 Anomaly Detection

Anomaly detection problem appears in many research and application scenarios. This is a problem of differentiating between data that is considered "normal" and data that does not reflect the usual functionality of a system. This problem is related to the problem of *noise canceling*. However, there is a significant difference. In case of noise removal, we want to model the noise and remove it to improve the processing of our measurements. As opposed to that, in anomaly detection we are interested in detecting outliers in order to study and analyze them, rather than just to work with normal data. Apart from anomaly detection, frequently used synonymous terms are outlier detection and novelty detection. The trained method or system for anomaly detection is also very often called *anomaly detector*.

Detection of points that do not conform to normal region imposes various challenges. Here are some examples:

- it is very difficult to find a model that encompasses all the normal behavior
- definition of normal behavior depends on the application area
- definition of normal behavior is susceptible to change that depends on the evolution of the system under consideration
- labeled data is not always available, especially for anomalous points
- noise in the data makes it difficult to differentiate between normal points and outliers

- results of anomaly detection are prone to malicious adversaries trying to promote outlier points as part of the normal functioning of the system
- depending on the application, we may not be interested in detecting all the outlying points, only outlying patterns that offer useful information

Because of the listed problems, it can be difficult to find a general solution to anomaly detection. Furthermore, there are different types of anomalies to consider [24]:

1. Point anomalies

   If an individual data point is not conforming to the distribution of the underlying data, it is considered a point anomaly. Real life example is a credit card transaction with an abnormal sum of money transferred to an unusual account number.

2. Contextual anomalies

   In case a data point is not anomalous by itself, but it appears in an unusual context, it is assigned as a contextual anomaly. When hunting for contextual anomalies we need to consider additional context variables, such as time or spatial coordinates. For example, if there is a peak in network traffic during the night coming from an office computer, that could indicate an anomaly.

3. Collective anomalies

   There are cases where, although one point is not considered an anomaly worth studying, multiple points taken together can bring another view on the underlying process. In case a group of points constitute a pattern that does not conform to the previously defined model or rule, we consider this a collective anomaly. One example scenario is detection of medical anomalies in EEG signals.

There are multiple approaches from preexisting areas of research adapted to detection of anomalous structures and behavior. According to the survey of Chandola et al. [24], these are:

1. Statistics

   Statistical methods use the assumption that normal instances should occur in high probability regions w.r.t a stochastic model. Instances that occur in the low probability regions are considered to be anomalies. Statistical methods can be used both with knowledge about the underlying data distribution (parametric methods), as well as with building this knowledge implicitly through data (non-parametric methods)

2. Information Theory

   Information-theoretic approaches exploit the fact that anomalies in data induce irregularities in the entropy patterns of the dataset. This means that we need to find the data instances whose inclusion introduces high difference in the entropy of data.

3. Spectral Theory

   Spectral techniques contain an approach to make the normal instances and anomalies separable. However, we need to find an embedding to a lower-dimensional subspace where a clear-cut separation can be made. Multiple algebraic methods, such as Principal Component Analysis [88] with its robust extensions and Multi-dimensional Scaling [76] enable us to find these embeddings.

4. Machine Learning

   Machine learning methods are motivated by the preexisting statistical approaches and in these methods we try to map the anomaly detection scenarios to standard machine learning problems. This enables us to use frameworks offered by well-known machine learning methods, such as kernel methods or neural networks. These frameworks also enable us to adapt the standard machine learning approaches to the conditions imposed by anomaly detection. In this thesis we make emphasis on this group of methods.

These categories of approaches are not completely distinct, and their differentiation is mainly in different formulation of problems and solutions, as a consequence of preexisting development in respective research areas. Although in the following text we focus on the formalisms and baseline methods of machine learning, some of these methods have their equivalent formulations in other categories in this list. Furthermore, it's possible to use methods from other categories to detect more obvious anomalies, before using the machine learning-based methods for more sophisticated anomaly detection.

2.2 Machine Learning for Anomaly Detection

In this section, we present a short introduction to machine learning. Since we base our work on machine learning-based anomaly detection methods in the following chapters, here we give an overview of most important machine learning terminology. When employing machine learning methods to make predictions about data points, such as determination of anomalous data regions, we use a training set of points (D_{train}), set of parameters (θ) and an objective function ($loss$). The objective function (loss function) depends on the ground truth outputs and the outputs of the function $f(X, \theta)$ that represents the model. We optimize this objective function in order to minimize the difference between the ground truth output and the output of the function f. This means that our

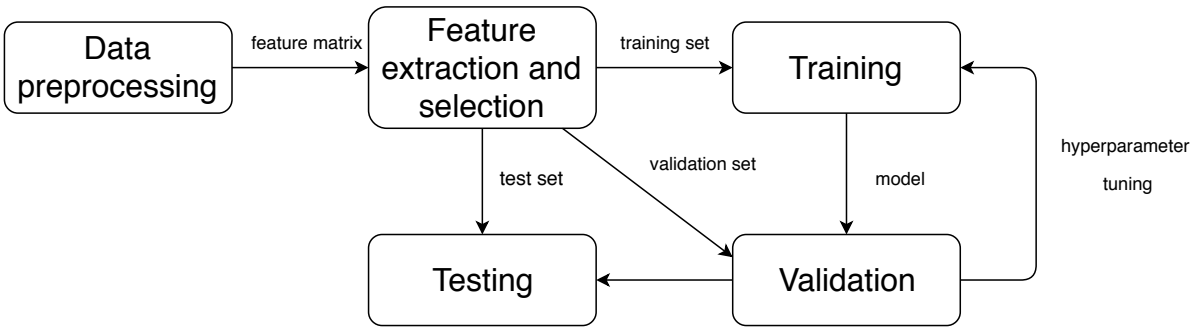


Figure 2.1: Machine learning pipeline

function f is able to predict the true outputs. In most standard approaches we assume a broad family of functions out of which we take f, and we use training algorithms to find the values of parameters (θ) that optimize the defined objective function.

Depending on the availability of the desired outputs for optimization, machine learning methods are categorized in:

- *Supervised learning* refers to those learning problems where supervised information is available directly in the training dataset, such as class labels.
- *Unsupervised learning* methods estimate parameters of the models based on the expected structure of the data. For example, clustering methods optimize cluster assignments based on inter-cluster separation and intra-cluster similarity of training data.
- *Reinforcement learning* In reinforcement learning the supervision is not given explicitly by expected outputs. Here the learning methods retrieve rewards as results of the actions. The assignment is to optimize the rewards by reinforcement learning techniques, by combining exploration and exploitation.

Another special category of machine learning methods is *semi-supervised learning*. The methods belonging to this category combine the strengths of both supervised and unsupervised approaches. These methods are beneficial in scenarios where labeled data is valuable, but only a small number of labeled points can be obtained. The next section contains more information about semi-supervised learning.

In anomaly detection, depending on the availability of labels we can use and extend machine learning methods from any of these categories to construct our anomaly detection approach.

Standard machine learning pipeline consists of multiple stages of data processing, also displayed in Figure 2.1:

- Data preprocessing

Standard machine learning methods process input data in a matrix form. Therefore we need to convert the data stored in a particular database to numerical matrices.

- Feature extraction and selection

  Before training, it is useful to reduce the dimensionality of the data using algebraic transformations to a lower-dimensional space or select features based on statistical significance.

- Training

  In training we estimate the parameter values that optimize our model with respect to a predefined objective function. Our objective function reflects the goal that we want to achieve with our machine learning model.

- Validation

  We can use a holdout set of validation data to test our model architecture. In case of poor performance of our model on this holdout set we can return to the previous step, change the model architecture and retrain the model.

- Testing

  In the testing phase we perform a final evaluation and deployment of our model. The performance of our model on the test set gives the best estimation of our success in learning the model from data.

2.3 Machine Learning Paradigms

In this section we describe the machine learning and statistics paradigms used in the next chapters. In particular, the Section 2.3.1 is important for Chapter 3, information from Section 2.3.2 is used in Chapters 3 and 5, while the introduction from Section 2.3.3 gives background to the Chapter 5. On the other hand, Section 2.3.4 is important for Chapter 3, while Section 2.3.5 serves as introduction necessary for Chapter 7.

2.3.1 Semi-Supervised Learning

The lack of proper data labeling has already been defined as an important problem in malware research [14]. Consequently, one would benefit from a method that offers maximum utilization of the available set of labels, as well as of unlabeled data. This setting is known in machine learning as semi-supervised learning and is halfway between supervised and unsupervised algorithms. While supervised learning is a paradigm that encompasses machine learning methods where the training data is labeled and the purpose of the algorithm is to optimize the classification of data on the validation and test datasets, unsupervised learning discovers the underlying structure in the data such as

clusters in the dataset. We use unsupervised learning when we do not have information about labels in the time of training. Since in semi-supervised setting we do have labeled data, but it is scarce, we combine the advantages of two separate methods to overcome this limitation. More specifically, in semi-supervised learning we leverage the property of data that it forms natural clusters. Even if we only have a small number of labeled data that identifies the clusters that exist in the dataset, we can propagate these labels in the neighborhood of the labeled data, by considering the clusters detected in the dataset. The book of Chapelle et al. [25] contains a breadth of information about methods of semi-supervised learning.

2.3.2 Nonparametric Learning

In many scenarios the parameter set of machine learning models is predefined and we only optimize the values of these parameters during training. However, in nonparametric methods the parameter set itself is adapted to the training data. For example, in kernel methods using a Gaussian Kernel, such as Kernel Density Estimation, the parameter matrix can contain new Gaussian parameters for every new training point. This enables a more precise function that determines classification border or predictions in general. In cases when we need to process an increasing number of data, it can also be difficult to predetermine the complexity of the model in terms of the size of the parameter set that is needed to deliver proper performance. For example, in topic models it is difficult to predict the number of topics. Nonparameteric methods can help to make more flexible learning of models from data.

2.3.3 Online Learning

In multiple application scenarios the training data does not appear in a single batch, but it is collected incrementally, while the system is working. This requires incremental training, where the model is updated while the system is working, by adding the information from the new data into the parameter fitting procedure.

In online learning [108], the training of a model is performed as the data arrives. Upon the arrival of an input data point, a model can provide the output based on this point. Based on an objective function criterion, such as the difference between the desired and obtained output, the model is adjusted using the new input data point. This means that the training of the model is not completed at once after obtaining a training set, as the training set grows with time. Instead, the training is performed while gathering the data and during the normal system functionality.

2.3.4 Variational Inference

In Bayesian machine learning methods it can be demanding to compute posterior densities for large-scale datasets, since we need to sum probabilities for the whole dataset to

find a normalization value. Variational inference is one of the methods to approximately compute probability densities. We introduce the techniques of variational inference needed for understanding the methodology used in the thesis, by summarizing concepts and using equations described in detail in recent surveys on this topic [18, 136].

For a general problem where we have joint density of observations (x) and latent variables (z) as:

$$p(x, z) = p(x \mid z)p(z) \tag{2.1}$$

we can retrieve the posterior density of the latent variable using the following equation:

$$p(z \mid x) = \frac{p(z, x)}{\int p(z, x)dz} \tag{2.2}$$

In case that the dataset is too large, we cannot compute the denominator exactly, because the integral is intractable. Therefore we need to find an approximation $q(z)$. We form the task for finding this approximation as an optimization problem, where we want to bring the approximation as close as possible to the original density:

$$q^*(z) = \arg\min_{q(z) \subset Q} D_{KL}(q(z) \mid\mid p(z \mid x)) \tag{2.3}$$

where Q is a family of approximate density functions. The operation D_{KL} in the formula is Kullback-Leibler divergence, which indicates how different one probability distribution is from another distribution. We need to define a family Q that will inherently bring us close to the original distribution, and then minimize the D_{KL} function output. The Kullback-Leibler divergence (D_{KL}) can in our case be defined as:

$$D_{KL}(q(z) \mid\mid p(z \mid x)) = \mathbb{E}[\log q(z)] - \mathbb{E}[\log p(z, x)] + \log p(x) \tag{2.4}$$

Since computing the Kullback-Leibler divergence includes the value $p(x)$, we cannot determine it directly. Instead we compute the value of the *Evidence Lower Bound* ($ELBO$):

$$ELBO(q) = \mathbb{E}[\log p(z, x)] - \mathbb{E}[\log q(z)] \tag{2.5}$$

This value is different from negation of KL divergence by a constant w.r.t. $q(z)$, therefore minimizing D_{KL} is equivalent to maximizing this lower bound.

An alternative to this method is to use a Markov Chain Monte Carlo approach, where we iteratively draw random samples of the latent variable, whose distribution would converge to the real posterior $p(z \mid x)$. Advantage of this method is that it contains convergence guarantees for a sufficient number of draws. However, in practice it could take a large number of draws to achieve this convergence, which is prohibitively expensive for large datasets [103].

2.3.4.1 Mean-Field Variational Family

A standardly used family of variational densities is the *Mean-Field Variational Family*. Its generic form is:

$$q(z_{1..m}) = \prod_{j=1}^{M} q_j(z_j) \tag{2.6}$$

This means that there is a separate independent variational parameter for each latent variable. We can assign different form of the function q for each type of latent variable. For example, for Bayesian mixture of Gaussians this function is:

$$q(\mu, s) = \prod_k q(\mu_k; m_k, \sigma_k^2) \prod_i q(s_i, \phi_i) \tag{2.7}$$

where the variational density is defined in two parts, a Gaussian distribution for each mixture and a distribution for mixture assignment.

2.3.4.2 Coordinate Ascent Method

One of the basic algorithms for solving mean-field variational inference is using coordinate ascent. In this algorithm, we modify each variational parameter separately, while holding other parameters constant. When fixing the other variational parameters, optimal value for q_j can be retrieved using the conditional probability of z_j given all other latent variables and observations:

$$q_j^*(z_j) \propto \exp(\mathbb{E}_{-j}[\log p(z_j \mid z_{-j}, x])) \tag{2.8}$$

We can use this or an equivalent expression to iteratively compute variational factors independently, based on the mean field assumption. The coordinate ascent procedure converges to a local maximum.

Using these updates, we can define the following steps:

1. Update all q_j iteratively
2. Compute $ELBO(q)$
3. If $ELBO(q)$ has not converged, return to step 1

2.3.4.3 Stochastic Variational Inference

Although we have defined a way to iteratively update variational parameters, we still need to run the iterative update through the entire dataset. In order to truly avoid this kind of update, we can introduce a procedure where we use a smaller part of the dataset at a time. We divide parameters into global variational parameters of the model and local

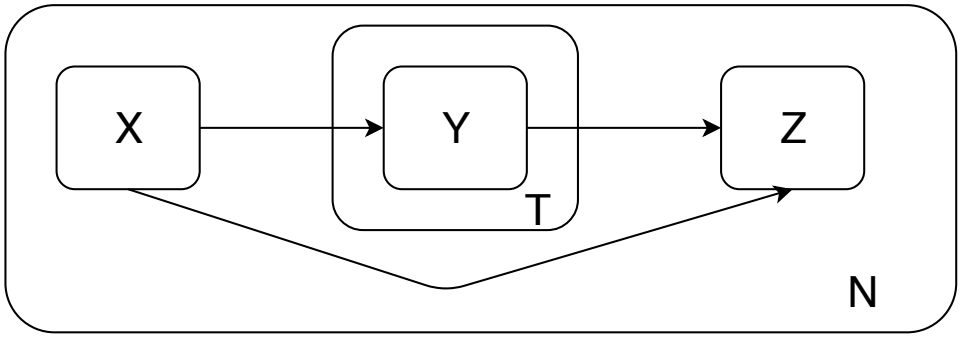


Figure 2.2: Graphical model for ground truth estimation from annotator labels [99]

parameters for each point. Then we alternately use global parameters to optimize the local variational parameters for subsampled data and adjust global parameters according to the results of this optimization. The optimization can be done by computing gradients of the $ELBO$ function w.r.t. the global and local parameters.

2.3.5 Learning from Multiple Annotators

The success of supervised learning procedures heavily depends on the quality of obtained labeled data. In some cases, labels for the input data are not obtained from one reliable source, but from multiple annotators with variable labeling expertise and reliability. In that case, we need to find a good method to use the *Wisdom of Crowds* to obtain correct labels out of these numerous data sources. For example, in case of Amazon Mechanical Turk, we could obtain labels for data from thousands of participants in exchange for money. However, how do we know if the labelers are competent and dedicated to this task? On the other hand, if we are retrieving labels from a machine, such as a measurement device, how do we know that this device gives reliable and useful labels?

Simple approaches to aggregate labels from multiple annotators is to use majority voting, or some averaging method. In case that most of the annotators are highly competent and diligent in labeling all of the data, their labels will outnumber the other, less adequate labelers. However, this may not be the case. Many times the labelers have variable expertise for different classes, different parts of the input space, and also many times annotators just give constant or random labels in an automated fashion, in order to acquire more labeling fees in a smaller amount of time. This motivates the need for more sophisticated statistical approaches.

In a general probabilistic approach, we are given a set of data points $X_i, i = 1..N$, labels $Y = y_{it}$ from annotators $t = 1..T$, and possibly ground truth $Z_i, i = 1..N$. Our goal is to find a way to estimate ground truth from data points and annotations. Basic probabilistic model is displayed on the Figure 2.2.

The joint probability for annotations (Y) and ground truth labels (Z) for input data

(X) can be expressed as:

$$p(Y, Z \mid X) = \prod_i p(z_i \mid x_i) \prod_t p(y_{it} \mid x_i, z_i) \tag{2.9}$$

In a simplified version, employed by Raykar et al. [94], the annotator labels are just noisy versions of ground truth labels, i.e. $p(y_i \mid x_i, z_i) = p(y_i \mid z_i)$. However, to get a more realistic model we need to incorporate the dependency on the input features. This is because each annotator could have different modeling expertise for different data. For example, we can express this with a Gaussian model [133]:

$$p(y_{it} \mid x_i, z_i) = \mathcal{N}(y_{it} \mid z_i, \sigma_t(x_i)) \tag{2.10}$$

The variance σ_t here depends on the input features, and can be parameterized depending on the output constraints. Furthermore, we can parameterize the dependency between ground truth and the input, for example with a logistic regression in order to get a classification model [133]:

$$p(z_i = 1 \mid x_i) = (1 + \exp(-\alpha^T x_i - \beta))^{-1} \tag{2.11}$$

The parameters of this kind of model can be estimated using the common maximum likelihood approach, in case that the ground truth is known in training time. This means that we search for parameters that maximize the probability of our data conditioned on the assigned class.

2.4 Baseline Machine Learning Methods

In this section we shortly describe baseline machine learning methods well-known in the literature, that we use and extend in our work. Knowledge of Section 2.4.1 is important for Chapter 3. Familiarity with the Section 2.4.2 is important for Chapters 4, 6 and 7, while Section 2.4.3 is important for Chapter 5.

2.4.1 Topic Modeling

In machine learning, data is very often organized as documents containing a series of tokens. Most explored examples of this kind of data are audio and video recordings, genetic sequences, and text documents. For instance, it has been determined that very often news articles belong to a smaller set of latent topics such as *Basketball*, *Tour De France*, *Hollywood*, *FBI Investigation*, etc [83]. On a higher level, topics could be *sports*, *culture*, and *finance*. Words belong to different topics with different probability, which is based on their appearance in documents belonging to corresponding topics. In other words, topics group documents based on the words that are contained in these documents. If topics have word distributions that are semantically interpretable, created

model also has a semantic meaning. This means that we can describe these topics based on the words that belong to them. The topic modeling approach to text modeling problem and vocabulary can be translated to problems with other types of data, in order to use the convenient features obtained with this method

Topic modeling methods are mostly constructed and described as generative methods in literature. In essence, the topics are constructed in such a way that the training documents can be generated with high probability using just the topics inferred from the model. If our documents can be confidently described by a smaller set of topics, this is convenient for document summarization, as well as feature extraction. It is important to know that the topics do not need to be known in advance, they are inferred from the set of documents and their contents.

Overall, topic modeling is a method to statistically explain a large set of documents using a small set of clusters (topics), based on frequency of different words in these documents. Typically, topic modeling methods are not sequence-aware, as they only count the appearance of words without taking account of their position. This approach has been often called *bag-of-words*.

One of the most adequate stochastic processes used for topic modeling is the *Dirichlet* process, where the results of draws are probability distributions. Furthermore, using the Dirichlet process is important in case that we need a model where only few latent topics can describe a large set of documents. Although it is not the first method for modeling structure of text documents, topic models were popularized with the development of *Latent Dirichlet Allocation* (LDA) method [19]. In this method the topic structure is sampled from a Dirichlet distribution as prior, which gives more flexibility in training the generative model. Although there exist related methods of topic modeling [37], LDA is the most used regarding document information retrieval because of its flexibility and modular structure. This method has been further adapted for document classification as well [93].

2.4.2 Neural Networks

Neural networks are popular machine learning constructs, designed with biological motivation from the structure of natural neural networks in the human brain [80]. From a mathematical standpoint, a neural network is a composition of multiple layers of nonlinear input feature transformations. These transformations are parameterized and the parameter values are determined based on input data during training using a gradient descent procedure. In feedforward networks the transformation looks like the following:

$$y = f(\sum_{j=1}^{M} w_j x_j + w_0) \tag{2.12}$$

where w_i are parameters of the neural network (weights), and f is a nonlinear function that adds complexity to the model. Nonlinearity enables neural networks to model more

complex relationship between input features and final outputs.

Multiple layers can be connected by using the output of one layer as an input to the next one. In case of classification this series typically ends with a *sigmoid* function (in binary classification) or *softmax* function (in multiclass classification). These functions transform activation values of the last layer into vectors of values between 0 and 1. These vectors indicate the classification labels. On the other hand, the outputs of the last layer could also be set to have continuous value, which is adequate for regression models.

Apart from supervised learning for classification or regression, neural networks can be used in unsupervised anomaly detection as well. In this case the architecture of the neural network is different. The input and output layers have the same dimensionality, while hidden layers have the same or smaller number of dimensions. This special kind of networks, called autoencoders, are trained to replicate the input during training, while compressing the data in the neural network parameters [51]. In case that the hidden layers have a smaller number of parameters than the input layers, the output of the hidden layers represent the compressed data representation. In the testing phase, we measure the difference in the test inputs and the output of the neural network. This difference is used to measure the ability of the network to replicate the test input. In case that the network is not able to replicate the test input using its hidden representation, the corresponding sample is treated as an anomaly. This way we can use autoencoders for unsupervised anomaly detection.

Neural networks have been studied for decades and many architectures have been proposed for modeling various types of data. The most common architecture type is a feedforward neural network, where the connections between layers are only directed towards the output layer. Some theoretical results show that even the 3-layer network (with one nonlinear layer) can be a universal approximator for smooth functions [32]. However, in practice, it has been shown that increasing the number of layers further can improve performance as it enables hierarchical feature extraction [15]. This paradigm of using a high number of layers has been called *deep learning* and it brought many improvements to classification accuracy according to recent publications, for example in image processing [64] and text classification [57].

2.4.2.1 Convolutional Neural Networks

Many successful approaches in deep learning come from designing neural networks that contain layers of specific structure, to accommodate the types of patterns expected to be present in data. For instance, one of the very often used architectures are convolutional neural networks, known for decades to be useful for image processing [67]. This type of networks have been successfully used in processing various signals as data - images, audio, video, as well as in text processing. As in typical feedforward networks, convolutional networks contain multiple connected layers. However, convolutional networks mostly consist of multiple layers of *convolution* and *pooling*. On the one hand, convolution layer uses a function to extract features from data samples by moving a parameterized

convolution filter in a predefined window [66]. In this case, discrete convolution with filter K in two dimensions (u and v) can be defined with the following transformation on input I [44]:

$$(K * I)_{r,s} = \sum_u \sum_v I_{r-u,s-v} K_{u,v} \tag{2.13}$$

where the filter K is given as a matrix of predefined size.

The output of a convolutional layer consists of a series of feature maps, where each map is a result of the aforementioned convolution. This kind of maps are computed in each convolutional layer.

On the other hand, pooling layer takes the results of a convolutional layer as input and extracts the most salient features using simple reduction functions, such as average or maximum. This way the pooling layer effectively executes subsampling of results from the feature maps. A sequence of convolutional and pooling layers alternates feature extraction and dimension reduction. The parameters of these transformations are trained using numerical optimization, by executing gradient descent on filter parameters based on the difference between the expected and retrieved outputs. Gradient is computed using backpropagation, i.e., by propagating the mentioned difference from the output layer back to the hidden layers.

In image processing, the convolutional filter can be used to recognize features in the image, which is very useful for object recognition or image classification. This way the objects can be detected almost invariant from their position. Apart from processing images, the convolutional networks have been found useful in other domains as well. In text processing applications (e.g., sentence classification, search, recommendation) we can extract information and detect high level features for short texts using the flexible convolutional filters.

2.4.3 Support Vector Machine

Basic idea of Support Vector Machine [30] is that the optimal hyperplane for separation of points into two classes can be obtained by optimizing for maximum margin between the two groups. If the groups of training points for two different classes are linearly separable, we can optimize the margin by setting the following optimization problem:

$$\frac{1}{2} \min \|w\|^2 \tag{2.14}$$

$$y_i(x_i w - b) = 1 \tag{2.15}$$

where y_i are the class labels for training points (1 or -1), x_i are input points and w and b define the hyperplane position and orientation. This problem can be solved using the method of Langrangian multipliers:

$$L_p = \frac{1}{2}|| \ w \ ||^2 - \sum_{i=1}^{l} \alpha_i y_i (x_i w + b) + \sum_{i=1}^{l} \alpha_i, \qquad \alpha_i > 0, \forall i \in N \tag{2.16}$$

An equivalent dual form can also be derived:

$$\max L = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j x_i x_j \tag{2.17}$$

subject to

$$\sum_i \alpha_i y_i = 0, \alpha_i \geq 0 \tag{2.18}$$

This form is especially convenient in case that the points are not linearly separable. The reason for this is that it includes the scalar product of pairs of points, which we can transform using a distance function of our choice. This function is called a kernel and the most well-known form is the Gaussian Radial Basis Kernel Function (RBF):

$$K(x_i, x_j) = \exp\left(\frac{- \ || \ x_i - x_j \ ||^2}{2\sigma^2}\right) \tag{2.19}$$

2.5 Constraints in Machine Learning

Machine learning approaches to solving problems of anomaly detection have been successful in the past. These approaches enable us to leverage a large amount of data in order to construct a statistical model of normal and anomalous behavior. However, leveraging such a vast amount of data can be difficult in case that constraints are present. Some of the main constraints come from limited resources in data and computational power.

Limitations in computational resources cause problems with scaling the machine learning systems to datasets that become larger as the availability of data is increased. This can be solved either by investing in larger computational capabilities of the data processing hardware or adapting the machine learning methods to an environment with resource constraints. Since many times the application scenario requires resource-constrained environment (mobile devices, network routers, virtual machines), in these cases it is very important to find a way for machine learning methods to work under the conditions of limited memory, CPU and bandwidth.

Another possible problem is that the assumptions we have about the data, reflected in the train model, can change in time. For example, in case of the data distribution changes, the parameters of the model need to be updated, i.e. model needs to be retrained to include those changes. Noise in the data can be increased, caused by natural changes of the observed process, or attacker's obfuscation or making deliberate

misleading manipulation of the input dataset. In order to preserve the reliability of our machine learning system, we need to detect these changes and adapt our model to these conditions.

2.5.1 Adversarial Learning

Machine learning methods are constructed to handle random statistical noise. Most of them use the idea of Gaussian noise in order to examine the ability of the methods to capture the essential information. However, in their baseline version the standard methods do not consider possible adversarial contamination of the input data.

While in case of a reliable data source, the Gaussian noise assumption has been proven as good enough, there are scenarios where a malicious actor can contaminate the data with a purpose of changing the results, thereby changing the noise distribution. These adversarial attacks [53] can be divided into:

- Exploratory attacks, where the attacker manipulates the test data in order to change the classification or regression result
- Causative attacks, where the attacker manipulates a small part of training data and changes the functionality of the subsequently trained machine learning system.

Furthermore, in examining the vulnerability of machine learning systems we can consider the following attacker knowledge:

- Perfect knowledge, where the attacker knows all the training data and the machine learning system in detail
- Limited knowledge, where the attacker has some knowledge, such as the architecture of the machine learning method used
- Zero knowledge, where the attacker has no knowledge

In many publications the assumption of perfect knowledge is used, although it is in many cases unrealistic. However, in many cases we can get close to perfect knowledge, for example if a standard classifier is used and if we have a good idea of the distribution of training data. Furthermore, the principle of transferability can apply among machine learning methods [85]. This means that adversarial modifications made on one machine learning system in many cases are effective in another, previously unknown system.

Behavior of machine learning methods under data manipulation can provide clues about what the system really learns and how vulnerable the learning mechanism is for a particular system and in general when using a certain method.

2.5.2 Budgeted Learning

Baseline machine learning methods do not contain the option to take account of budget constraints for feature acquisition in training or test time. Furthermore, they do not contain the possibility to control the size of the model, which has consequences on the need for larger computational resources.

This is especially important in online learning, where it is important to find approaches that limit the model complexity in order to maintain retraining speed. Online learning methods on a budget limit the number of support vectors in support vector machines to bind model growth. They are characterized based on their strategy to stay within budget size limits. Model performance is highly affected by the selection strategy based on which it is determined which support vector will be removed. Multiple previous works suggest various strategies:

- stop updating after reaching the support vector budget ([84])
- remove the oldest support vector ([23])
- remove a random support vector ([23])
- remove support vector that increases the loss the most([31])

In addition, some of the algorithms also tune the weights of the support vectors to give higher weight to the most recent data.

2.5.3 Communication-Efficient Learning

There are multiple variants of distributed learning where communication constraint is present. For example, training with a large-scale dataset stored on multiple machines can be distributed, with a communication between training machines and a parameter server [71]. This incurs a need for a large bandwidth for communicating the results, with possibly also frequent synchronization among worker machines, if the model is sequential.

Another example is federated learning [75], where training is partially done in resource-constrained clients, and the results are aggregated on a central server on the cloud. Here the devices where the training is executed could have limited reliability, higher latency and lower throughput.

2.6 Performance Measures

Measure of performance is a very important part of development of anomaly detection methods. There are multiple values that need to be taken into account when evaluating anomaly detection systems. They are defined in terms of true positive (TP), true negative (TN), false positive (FP) and false negative (FN) points [80]:

1. Accuracy - proportion of accurately classified data points

$$ACC = \frac{TP + TN}{all\,points} \tag{2.20}$$

2. Precision - number of correctly detected positive examples, relative to all the examples considered positive

$$PR = \frac{TP}{TP + FP} \tag{2.21}$$

3. Recall - frequency of positive samples that were correctly labeled (true positive rate)

$$RC = \frac{TP}{TP + FN} \tag{2.22}$$

4. F_1-score - combination of precision and recall

$$F_1 = \frac{2PR \cdot RC}{PR + RC} \tag{2.23}$$

5. Specificity (true negative rate)

$$Sp = \frac{TN}{TN + FP} \tag{2.24}$$

6. Area under ROC Curve (AUC)

   ROC (Receiver Operating Characteristic) curve is a plot of true positive rate (sensitivity) against false positive rate at various values of threshold parameters, for example threshold value between an anomaly and a "normal" point. This curve shows how stable the system is in terms of performance. As a derived quantitative performance measure we compute the area under ROC curve as a sum or an integral.

While it is not necessary to measure all of these values to have a good idea of the anomaly detection performance, more performance results give a better illustration of the functionality for the anomaly detector.

2.7 Summary

In this chapter we defined the basic terms needed to understand the rest of the thesis text. First we described the anomaly detection problem and various approaches for its solution. Next, we in particular presented machine learning as a very popular and effective approach. We introduced specific concepts and baseline methods from machine learning that will be used throughout the thesis text. Furthermore, we stated the constraints that can be imposed on machine learning systems. Finally, we described the standard performance measures used to evaluate various machine learning systems, including anomaly detection solutions.

In the next chapters we describe the scenarios where anomaly detection is applied in constrained environments. Each chapter contains a description of a specific scenario and our contribution to solving the anomaly detection problems. Furthermore, we explain how our methods generalize to other similar scenarios as well.

Chapter 3

Online Topic Models

This chapter contains the first scenario for anomaly detection with constraints. In particular, we examine methods for malware variant detection and classification using behavioral traces consisting of system calls. We evaluate and improve the use of statistical topic modeling to solve this problem, with taking into account the curse of dimensionality of long execution sequences and an online learning setting.

Parts of this chapter are published in the paper of Kolosnjaji et al. [62] about adaptive semantics-aware malware classification.

3.1 Related Work

This section contains the description of the research efforts that precede our work. These efforts are mostly divided into research dedicated to (*i*) application of machine learning methods in malware analysis and (*ii*) designing systems to support the malware analysis process. Therefore we explain the evolution and current state of those two groups of methods separately. Furthermore, we explain how the methodology used in our approach takes into account the related papers and builds a new approach upon this work.

3.1.1 Machine Learning Methods for Malware Detection

Machine learning has been used in multiple research efforts as a malware detection and classification method. Various features that characterize program behavior have been used as input data for the machine learning-based procedures: system calls [126], registry accesses [50], and network packets [116]. These event sequences are analyzed using unsupervised (e.g., clustering), semi-supervised, or supervised learning (classification) methods. Static program code features have also been deployed for malware classification [107]. The classification methods can be further divided into one-class anomaly detection [50], binary classification [90], and multiclass learning [97]. One-class classification is used in case that we want to create a model for normal behavior (benign

samples) and detect malware as a deviation from that model. In binary classification we optimize the classification boundary between benign and malicious samples. Multiclass classification methods are able to differentiate between different, previously known and defined classes of malware instead of, or in addition to differentiating between benign and malicious samples.

Researchers that perform malware detection usually maintain a sample set from different malware families with their static and behavioral patterns, and use them as a baseline to properly classify the suspicious applications. For instance, in the case of sequential data, automatic methods for extraction of relevant features can be used to cope with the possibly noisy and high-dimensional data. An example of this is given by recent application of statistical topic modeling approaches to the classification of system call sequences [130]. This approach could be extended by taking system call arguments as additional information and including memory allocation patterns and other traceable operations [129]. Support vector machines with string kernels represent another novel methodology, where a standard classification scheme is augmented to work robustly with system call sequences of variable length [90]. However, most of these approaches only consider malware detection, and do not focus on classifying malware samples into families. Another example of sequential data is the network traffic. Towards this direction, the network traffic produced by the analyzed samples can be classified by taking into account the frequency and length of different types of packets or generating n-gram features out of packet payloads. As a matter of fact, researchers have already proposed various approaches to model the network data and design anomaly detection procedures for network infrastructures with purpose of network security [41, 69, 112, 135].

Previous works have considered many potential solutions for semantics-aware malware classification and analysis, including topic modeling. However, they have not dealt with the typical setting in malware analysis systems where a high number of samples is acquired online and models must be updated to give an accurate result. Therefore their methodology is only adequate in a scenario of offline malware analysis.

3.1.2 Big Data Malware Analysis Systems

Since security companies get overwhelmed with hundreds of thousands of malware samples on a daily basis, the problem of malware classification can be defined as a *Big Data* problem. Recently, there have been many efforts to create Big Data platforms for malware analysis. Examples of such systems are BinaryPig [48], Polonium [26], BitShred [56], and WINE [38]. BinaryPig is a system for distributed processing of data obtained by static malware analysis, leveraging the recent advances in tools for Big Data domain. It uses Hadoop File System, MapReduce, and ElasticSearch as building blocks for scalable processing of static analysis data. Polonium is another system for large-scale mining of malware. It leverages graph mining approaches to build a reputation-based system to identify malware among terabytes of anonymously submitted suspicious files. BitShred, on the other hand, is an attempt to design and build a scalable malware

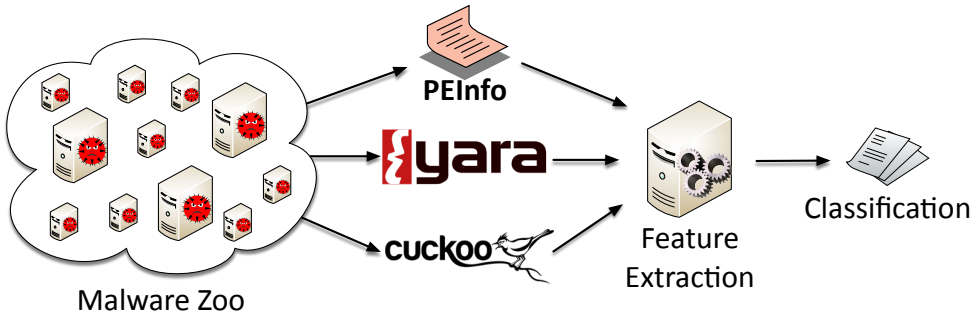


Figure 3.1: Malware classification architecture.

analysis system. It focuses on increasing efficiency of similarity analysis with feature hashing and uses similarity information for clustering. Finally, WINE is an approach that leverages Big Data and creates a scalable reputation-based security intelligence system, which also includes intrusion detection for network-based attacks.

These systems use machine learning-based technology and represent advances in scalability of malware detection and feature extraction. However, they do not emphasize on the development of statistical methods and do not consider semantic interpretability of the statistical models. Machine learning models very often need tuning and the absence of semantics can make such efforts extremely difficult for malware analysts. It is very important for analysts to be able to interpret the model in order to focus their efforts properly. In our approach, we do not only consider advanced topic modeling methodology for semantics-aware modeling, but we also take into account the scenario that a high influx of malware induces changes in the dataset and requires adaptation of the classification model. We automate this adaptation in order to maintain topic modeling feature extraction, using the nonparametric modeling methodology. Furthermore, our approach joins results of static and dynamic malware analysis and acknowledges the case where labeled examples are scarce.

3.2 Methodology

We propose a classification scheme based on a topic modeling feature extractor. In particular, we want to be able to discover semantic features of malware classes, maintain an adaptive topic model, and maximize the utilization of a semi-labeled dataset from heterogeneous data sources. To do so, we first emphasize on extracting semantic features from high-dimensional and noisy data. Second, we optimize the classification mechanism under the setting where low number of labeled samples is available. To this end, we join results of static and dynamic malware analysis to unify these different views on properties of malware samples. Finally, we design an architecture that is adaptive in the online training setting. In summary, our malware classification architecture complies with the scheme displayed on Figure 3.1.

3.2.1 Data Acquisition

The malware samples were collected over multiple months from three primary sources: Virus Share [98], Maltrieve [74], and private collections. We chose these sources to provide a large and diverse volume of samples for evaluation.

Data acquisition is done using widely available tools for the static and dynamic malware analysis. On the one hand, static analysis provides us with features extracted from the code of the malware samples. For this purpose, we use two sources aimed for static analysis: PEInfo [29, 128] and Yara [8]. We leverage PEInfo to extract entropy, size of different PE sections, and the collection of imported libraries. Similarly, Yara provides us with a list of used function calls to the Windows kernel API and other custom signatures extracted from the code.

On the other hand, dynamic analysis enables us to gather reliable behavioral data without the need for deobfuscation. There exist various tools that enable tracing the execution of malware and gather logs of execution sequences [46, 68]. We select the Cuckoo Sandbox, which provides a controlled environment for executing malware. During the execution of malware samples we record calls to the kernel API that we later use to characterize malware activity. For each sample we obtain a sequence of API calls, which is preprocessed by removing subsequences where one API call is repeated multiple times in a row. We cut these subsequences by using only one kernel API call instance as representative in the resulting sequence. In multiple samples we have noticed the repetition of one API call; for example, when malware repeatedly tries to open a file.

In addition, we leverage VirusTotal [3] by extracting antivirus signatures from its web service, for each malware sample we use. Users can upload MD5 hashes of malware executables to VirusTotal and retrieve results from multiple antivirus engines through the VirusTotal API. These engines are signature-based and compare the submitted hash to the data in their own database. By using the VirusTotal services we access malware analysis results and signatures, out of which we are mostly interested in retrieving ground truth labels for our classification. In a lack of other label sources, we use antivirus signatures in label construction for training and testing our classification scheme. Since antivirus programs use customized strategies for signature generation, we need to find a way to extract one numerical training label per unique sample using the diverse antivirus signatures. We use signature clustering to achieve this goal.

3.2.2 Signature Clustering

To get more confident training and testing labels, we perform a selection process that uses a simplified version of signature clustering method introduced in VAMO [89]. Specifically, we create signature vectors for every malware sample that contains signatures given by different antivirus engines. We use boolean features to generate these vectors, where each feature reveals presence or absence of a certain antivirus signature. Our assumption is that the malware samples of the same family will have the same or sim-

ilar boolean feature vector. Next, we use a variant of *cosine distance* as a measure of difference among signature vectors for our clustering process. We cluster the samples using DBSCAN [39], as we do not know their number in advance. Finally, we select ten clusters with the highest number of members as classes for classification. This way we cover most of our labeled dataset. Since the classes assigned to our malware resemble the families defined by antivirus engines, we use the terms *class* and *family* interchangeably.

3.2.3 Feature Selection

Static analysis tools provide us with a high number of features. In detail, we retrieve 23,060 features from PEInfo and 3805 features from Yara extracted from the malware binary files. Using a high number of features makes the classification problem ill-posed and therefore we choose to utilize feature selection methods to obtain an optimal feature set. This selection is based on the correlation of features and the ground truth class. We use *univariate feature selection* approach and perform a χ^2 test for all training samples. This way we can extract the features that are most relevant to our classification problem and reduce the computational effort needed for the training process. For our purpose we achieve best results by selecting 10,000 features for PEInfo and 1000 features for Yara.

3.2.4 Topic Modeling Algorithms

To extract features from the kernel API call sequences we utilize the topic modeling approach, which includes a well-developed set of methods already heavily used for automatic information retrieval from text and image data.

General Approach. As we already mentioned, topic modeling is a method based on the fact that a collection of tokens (words) from documents can be grouped to a limited set of topics. More specifically, we apply topic modeling to process data from dynamic malware analysis, as we consider that a list of API calls can be divided into a smaller number of latent activities. In our case, documents are malware execution logs and words are elements of malware execution sequences—calls to the Windows Kernel API. Additionally, topics are groups of these elements that constitute an elementary operation, for instance, registry access and modification, file manipulation, process creation and invocation.

This analogy justifies the attempt to adapt the topic modeling approach for the malware classification problem. The general topic modeling scheme can be represented with the following formulas:

$$G \sim DP(\alpha, H) \tag{3.1}$$

$$\theta_i \mid G \sim G \tag{3.2}$$

$$x_{j,i} \mid \theta_i \sim F(\theta_i) \tag{3.3}$$

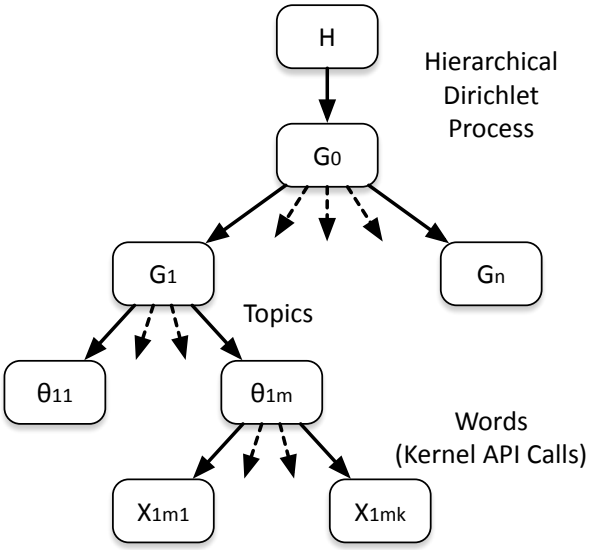


Figure 3.2: Graphical model for our Hierarchical Dirichlet Process.

where parameter G (a Dirichlet distribution) controls the topics and generates the parameter θ_i. Words ($x_{j,i}$) are generated based on this parameter. Dirichlet process is actually a distribution of distributions. The draws from Dirichlet processes are probability distributions, which are inferred for the next parameter in the chain. This parameter controls the word distribution for single topics. Topic modeling based on a Dirichlet process enables us to define a generative model, where each document is a mixture of a small number of topics. It is important to note that topics are not known in advance, but are inferred by the topic modeling methods. This enables us to uncover previously unknown semantics from the malware execution logs. Parameters are approximately determined using variational inference and Markov Chain Monte Carlo methods [122], as exact inference is not tractable. This also enables fast retraining in case of need for online update of the model. Figure 3.2 contains a graphical model used for topic inference, where the directed edges show the process of word generation.

Topic models can be essential for classification performance, as important latent structure is inferred and noise canceling is implicitly executed by extracting the important topics. However, even more crucial is the possibility of semantic interpretation. Although malware analysts are able to get a rich set of information from the dynamic malware analysis tools, this information needs to be further analyzed and significant expert knowledge is required to extract the important information out of the logs retrieved from these tools. Thus, it would be extremely useful to automate this procedure and to extract relevant data about the malware activity. This would enable analysts to achieve their goals faster and with statistically confident results. Therefore, we develop a more efficient alternative to the cumbersome deterministic manual analysis procedure. Even if the topics do not have an obvious semantic meaning, comparing the topic structure

among different malware families can enhance the classification process and provide new knowledge about the dataset in use.

Previous work demonstrated the utility of topic modeling for extracting semantics out of kernel API call logs by using LDA, where topic parameters are drawn out of the Dirichlet distribution [19]. Furthermore, this method was adapted from a bag-of-words method to a new scheme that takes account of sequential data ordering [129].

In its standard form, LDA uses a bag-of-words assumption, which means that it does not capture the sequential nature of the document: it only counts words independently. The LDA method is characterized by the following word generation hierarchy:

$$\theta \sim \mathrm{Dir}(\alpha) \tag{3.4}$$

$$z_n \sim \mathrm{Multinomial}(\theta) \tag{3.5}$$

$$w_n \sim \mathrm{Multinomial}(z_n, \beta) \tag{3.6}$$

The parameter θ is drawn from a *Dirichlet* distribution, hence the name *Latent Dirichlet Allocation*. This parameter determines the process of drawing topic assignments to different words (z_n). In the next level of hierarchy, word is generated from a multinomial distribution, depending on the previously drawn topic. At the inference time we compute the posterior distribution of the hidden variables (θ, z) given the document contents and hyperparameters.

However, this approach is not scalable on large sets of malware and online learning, and is sensitive to noisy sequences. It also requires a predefined number of topics, which would need to be manually updated by the malware analyst as new data is acquired. In case of an organization that maintains its own malware dataset and receives a high amount of submissions on a daily basis, this kind of setting may not be satisfactory.

Hierarchical Dirichlet Processes (HDP). Given the limitations of LDA, we take a different approach, using methods that bring the required improvement to online incremental learning. More precisely, we utilize an adaptive method called *Hierarchical Dirichlet Process* (HDP) [117]. In this method the topic distributions are also determined by Dirichlet processes, yet there exist different processes for each document. These processes, however, are not independent. They are drawn from a prior Dirichlet process, which depends on parameters that control the growth of topics and their distribution as the dataset grows in time:

$$G_0 \sim DP(\alpha_0, H) \tag{3.7}$$

$$G_j \mid G_0 \sim DP(\alpha_j, G_0) \tag{3.8}$$

where Dirichlet processes G_j are conditioned by the prior G_0.

Overall, the general setting of the topic modeling remains the same: documents belong to multiple topics and words depend on topic distributions. HDP is an instance

of nonparametric machine learning methods. As a difference from parametric methods, like LDA, nonparametric methods are used when we want the parameter set to change with the dataset. HDP introduces a more flexible approach, which is also more computationally demanding. Actually, this is the case with all the nonparametric machine learning methods. Nevertheless, there exist modifications that trade the accuracy of the method for performance in an online setting [123]. We use these modifications to create a scalable approach with respect to the computational demand. Our implementation is based on extending the *GenSim* library [95], developed for the estimation of text document similarity.

Semi-Supervised HDP Semi-supervised learning is an approach where we utilize both labeled and unlabeled data. We consider two approaches in semi-supervised learning for our test case. One approach is to use unlabeled data for pretraining our topic models, and continue further with supervised HDP. Another approach is to include the unlabeled data into the iterative optimization procedure. In this case, our parameter update procedure includes labeling the unlabeled documents, as well as changing the label assignments to labeled documents. Since our proposed model is semi-supervised, where we don't have labels for all the documents, we use a downstream approach. This means that we create a model where we generate labels after the topics are generated.

We denote y as our response variable, also named label or class, which is a distribution over $y_1, ..., y_C$ where C denotes the number of classes. We implemented this distribution according to [137] with the following softmax function:

$$p(y_j|\bar{\theta}_j, \mu) = \frac{exp(\mu_{yj}^T \bar{\theta}_j)}{\sum_{l=1}^{C} exp(\mu_l^T \bar{\theta}_j)}, \tag{3.9}$$

Equation 3.9 shows that the labels are dependent on two variables, μ and $\bar{\theta}_j$, μ is a new parameter introduced by [137] for the proposed supervised HDP model. The parameter $\bar{\theta}_j$ represents the cumulative topic count and can be approximated as follows:

$$\bar{\theta}_j = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T} \rho_{jt} \zeta_{jnt} \tag{3.10}$$

In the following section we will describe how the latent variables and parameters of the model are inferred from the observable data.

Inference As we already mentioned, our model is based on the supervised HDP established by Zhang et al. [137] and this model is based on the online HDP with variational inference introduced by Wang et al. [123]. Here we shortly describe the supervised model, which is important for enabling classification. Given the described model above it is necessary to compute or estimate all the latent variables in order to infer the latent topics as well as the labels for the documents. There are multiple latent variables that

need be to inferred: the word distribution of each topic ϕ, the corpus-level sticks β', the document-level sticks π', the indicator variable which maps document-level topics to the corpus-level topics $\boldsymbol{c}$, the indicator variable which maps document-level topics to words $\boldsymbol{z}$, and the softmax parameter μ. The inference is done by following the standard mean field variational steps [123]. We factorize the variables by making an assumption that the above mentioned latent variables are independent and that they factorize as stated below:

$$q(\beta', \pi', \boldsymbol{c}, \boldsymbol{z}, \phi) = q(\beta')q(\pi')q(\boldsymbol{c})q(\boldsymbol{z})q(\phi) \tag{3.11}$$

where according to [137] each factor is computed given certain variational parameters as described below:

$$q(\beta') = \prod_{k=1}^{K} q(\beta'_k | v_{1k}, v_{2k}), \tag{3.12}$$

$$q(\pi') = \prod_{j=1}^{M} \prod_{t=1}^{T} q(\pi_{jt} | a_{1jt}, a_{2jt}), \tag{3.13}$$

$$q(\boldsymbol{c}) = \prod_{j} \prod_{t} q(c_{jt} | \rho_{jt}), \tag{3.14}$$

$$q(\boldsymbol{z}) = \prod_{j} \prod_{n} q(z_{jn} | \zeta_{jn}), \tag{3.15}$$

$$q(\phi) = \prod_{k} q(\phi_k | \lambda_k). \tag{3.16}$$

In Equation 3.12 and 3.13 the variational parameters for the beta distribution are stated, v_{1k}, v_{2k} (corpus-level) and a_{1jt}, a_{2jt} (document-level). This means that they control the stick-breaking on the corpus and document-level, thus they also govern the weights. The two variational parameters for the multinomial distributions which represent the per-document topic and per-topic word proportions are ρ_{jt} and ζ_{jn} and can be found in Equation 3.14 and 3.15. ρ_{jt} is a multinomial parameter which determines the probability to select corpus-level topic k for the j^{th} document's t^{th} topic. ζ_{jn} is a multinomial parameter as well and it governs the probability to select the t^{th} topic for the j^{th} document's n^{th} word. In the last Equation 3.16, λ_k is the variational parameter for a Dirichlet distribution. Furthermore, μ is the softmax parameter that determines the proportions of corpus-level topics for each class.

For the labeled data in our case we use the updates, defined by [137], for the following parameters: ρ, ζ and μ. However, there is no closed form solution to compute ρ and μ, therefore we will use the partial derivative and use coordinate descent to update the variables. For the sake of clarity, we review the partial derivatives for ρ and μ, and the update for ζ as well [137].

Let

$$\mathcal{F} = \sum_{l=1}^{C} \prod_{n=1}^{N} \left(\sum_{e=1}^{K} \sum_{i=1}^{T} \rho_{jie} \zeta_{jni} exp(\frac{1}{N} \mu_{le}) \right), \tag{3.17}$$

and let

$$h_i = \sum_{l=1}^{C} \Big(\prod_{n=1}^{N} (\sum_{e=1}^{K} \sum_{i=1}^{T} \rho_{jie} \zeta_{jni} \exp(\frac{1}{N} \mu_{le})) \cdot (\sum_{e=1}^{K} \rho_{jie} \exp(\frac{1}{N} \mu_{le})) \Big), \tag{3.18}$$

where C is the number of classes, N is the number of words, K is the maximal number of corpus-level topics and T is the maximal number of document-level topics.

The partial derivative for the ρ update is given below [137]:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \rho_{jtk}} =& \sum_{n=1}^{N} \zeta_{jnt} \mathop{\mathbb{E}}_{q}[\log p(w_{jn}|\phi_k)] + \mathop{\mathbb{E}}_{q}[\log \beta_k] - 1 - \log \rho_{jtk} + \mu_{yjk} (\frac{1}{N} \sum_{n=1}^{N} \zeta_{jnt}) \\ & - \mathcal{F}^{-1} \Big(\sum_{l=1}^{C} \Big(\Big(\prod_{m=1}^{N} (\sum_{e=1}^{K} \sum_{i=1}^{T} \rho_{jie} \zeta_{jmi} \exp(\frac{1}{N} \mu_{le})) \Big) \\ & \cdot \sum_{n=1}^{N} (\frac{\zeta_{jnt} \exp(\frac{1}{N} \mu_{lk})}{\sum_{e=1}^{K} \sum_{i=1}^{T} \rho_{jie} \zeta_{jni} \exp(\frac{1}{N} \mu_{le})}) \Big) \Big). \end{aligned} \tag{3.19}$$

The partial derivative for the μ update is given below [137]:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mu_{jk}} =& \sum_{j=1}^{M} \Big([y_j = y] \frac{1}{N} \sum_{n=t}^{N} \sum_{t=1}^{T} \rho_{jtk} \zeta_{jnt} - \mathcal{F}^{-1} \prod_{m=1}^{N} \Big(\sum_{e=1}^{K} \sum_{i=1}^{T} \rho_{jie} \zeta_{jmi} \exp(\frac{1}{N} \mu_{ye}) \Big) \\ & \cdot \sum_{n=1}^{N} \Big(\frac{(\sum_{i=1}^{T} \rho_{jik} \zeta_{jni}) \cdot \frac{1}{N} \cdot \exp(\frac{1}{N} \mu_{yk})}{\sum_{e=1}^{K} \sum_{i=1}^{T} \rho_{jie} \zeta_{jni} \exp(\frac{1}{N} \mu_{ye})} \Big) \Big). \end{aligned} \tag{3.20}$$

ζ update is given below [137]:

$$\zeta_{jnt} \propto \exp \Big(\sum_{k=1}^{K} \rho_{jtk} \mathop{\mathbb{E}}_{q}[\log p(w_{jn}|\phi_k)] + \mathop{\mathbb{E}}_{q}[\log \pi_{jt}] + \frac{1}{N} \mu_{yj}^{T} \rho_{jt} - (h^T \zeta_{jn}^{old})^{-1} h_t \Big). \tag{3.21}$$

After taking the supervised updates into account we can now define our approach more precisely. Our model combines the two methods: on the one hand the unsupervised HDP [123] and on the other hand the supervised HDP [137]. We implemented and explored two approaches for combining the two methods. In the first approach we use the unsupervised HDP for pretraining the model and determining the corpus-level parameters and continue afterwards with the supervised method. On the other hand, we also attempt the second approach, to combine the semi-supervised and supervised

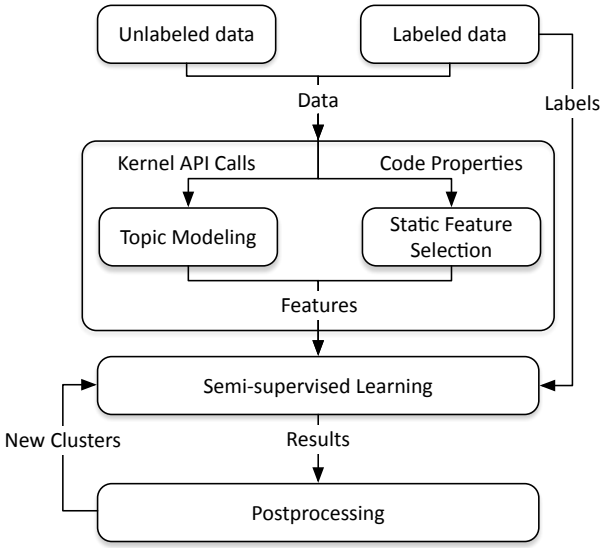


Figure 3.3: Semi-supervised learning scheme.

method in an alternating fashion, similar to expectation-maximization. This means that we alternate between document-level updates and corpus-level updates. After each corpus-level update we label the documents using the most probable class:

$$p(\hat{y}_j) = argmax_{y_j \in \{1,\dots,C\}} p(y_j|\bar{\theta}_j, \mu) \tag{3.22}$$

This entire algorithm is repeated until the maximum number of *epochs* (training iterations) is reached.

The updates for the unsupervised part can remain the same as they are already implemented by [95] based on [123], while we implement the supervised algorithms.

Although this algorithm continuously updates both labeled and unlabeled documents, we got better results with the pretraining approach. Therefore in further text we will show results obtained with this method.

3.2.5 Semi-Supervised Malware Classification

Accurate malware classification is often difficult due to lack of confident label sources. We can find proper signatures only for a small subset of malware samples, even by utilizing services such as VirusTotal. One approach to tackle this problem is to use strategies from semi-supervised learning, where we influence the usual malware clustering procedure with high-confidence labels. Figure 3.3 displays our semi-supervised classification scheme. Our system unifies advantages of topic modeling and semi-supervised learning. To this end, data retrieved from static and dynamic analysis tools are run through feature extraction and forwarded to the classification stage.

To achieve an effective semi-supervised learning model, we take two separate approaches to classify the static and dynamic analysis results. For results retrieved from static analysis we use label propagation. This method uses labeled data and density-based clustering to propagate the given labels through the dataset. The propagation of labels is conditioned by the similarity structure between data samples. In particular, we use a regularized variant of label propagation, to take account of the possible noise in labeling [139].

For dynamic analysis results we use another alternative. In a semi-supervised setting we can use unlabeled data for initial pretraining of topic models before using the actual labeled data or use a fully semi-supervised training approach. To discriminate between classes of malware, we make use of a *maximum-a-posteriori* (MAP) approach. This approach is used in machine learning very often when estimating distributions and parameters of a model. As a result, when classifying, we assign the class label to the data that is inferred with a highest probability.

We create a topic model for every existing class, based on the available logs of API calls. For each new log we evaluate the likelihood that its API call sequence would be generated from each topic model ($P(D = x \mid y = c_i)$). We also estimate independent prior probability of a certain class ($P(y = c_i)$) by simply calculating the share of certain class in the labeled dataset. Using the MAP approach, we evaluate the conditional probability of a certain sample belonging to the class i:

$$P(y = c_i \mid D = x) = \frac{P(D = x \mid y = c_i)P(y = c_i)}{\sum_i P(D = x \mid y = c_i)P(y = c_i)} \tag{3.23}$$

After computing the conditional probabilities for all classes, we find the most probable class by maximization:

$$CLASS(x) = \underset{i}{\operatorname{argmax}}(P(y = c_i \mid D = x)) \tag{3.24}$$

Malware sample is classified to the class to which it belongs with the highest probability.

Once the separate classification procedures finish for the outputs of available static and dynamic malware analysis tools, we forward the classification results to the aggregation and postprocessing stage.

3.2.6 Result Aggregation and Postprocessing

Our semi-supervised learning method returns probabilities of malware belonging to the predefined classes. These probabilities are results of separate classification using our three data sources (i.e., PEInfo, Yara, and Cuckoo). We combine these results to get a reliable class probability estimation. In machine learning-based classification it is often beneficial to combine multiple data sources and different classifiers to reduce model overfitting and use advantages of different methods in one system [65]. This approach is called *ensemble learning*. Multiple methods of various sophistication exist for combining

different classifiers. We argue that, since we do not have a large set of classifiers, there is no need for complicated ensemble learning approaches. In case of a larger number of data sources, an approach such as mixture of experts can be used, however, we did not notice any advantage of this approach in our case. For our experiments we use median and average of probability values, and majority voting of class assignments resulting from the three data sources. By aggregating the classification results, we get a more robust classifier. In fact, we combine the advantages of the static and dynamic analysis, to get a better classification performance. The combination of multiple views on data makes our results more reliable.

During the online classification procedure our system can detect the appearance of a new cluster. This can happen in one of the following cases: (*i*) new data has been put in the learning algorithm that contains a previously unknown label, or (*ii*) there is a new region of high local density that is detected during the execution of the learning algorithm. With the postprocessing algorithm, it is determined if the new sample can be confidently assigned to one of the existing classes, or a new class needs to be defined. Introduction of new classes can be done automatically by tuning the machine learning model, and in addition the new labels can be approved by a malware analyst. If we do not expect the new classes to appear very often, we can assign this job to the analyst, who can give a reliable estimation and help avoid possible mistakes in labeling. If indeed a new cluster is confirmed, the algorithm must be retrained in order to include this new fact into the machine learning model.

3.3 Evaluation

In this section we evaluate our approach. The extracted results prove advantages of topic models, semi-supervised methods, and combining results of static and dynamic malware analysis into a unified classification procedure.

For this purpose, we took ten recurring malware families from our labeled dataset of 2000 malware samples, as they correspond to clusters that contain more than ten members, covering more than 90% of our labeled dataset. The class titles were directly extracted from VirusTotal, where we manually chose signatures from multiple antivirus programs that were most prominent in our dataset. In addition to the labeled samples, we had 15,000 samples that we used as unlabeled, as the results of VirusTotal did not provide us with signatures for them. We then divided the dataset into training and test sets using a variant of three-fold crossvalidation. More precisely, the dataset is divided randomly in three parts, where two parts are used for training and the last part for evaluation. This division and accuracy experiment were repeated ten times and we took the average of the results. The distribution of samples in our dataset is mostly uniform, except for one significantly bigger and three smaller families (see Figure 3.4). However, we take this into account in crossvalidation when determining the training and test set, as well as during the evaluation of our approach.

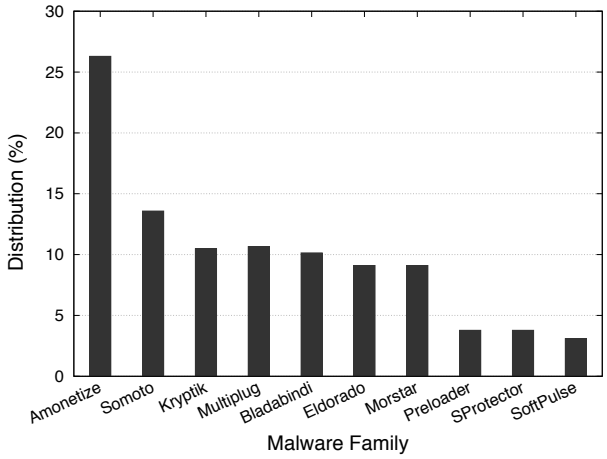


Figure 3.4: Samples distribution by family.

Table 3.1: Accuracy evaluation of LDA for different number of topics.

Family	LDA for a different number of topics						HDP
	1(%)	5(%)	10(%)	20(%)	40(%)	80(%)	
Amonetize	0.0	0.0	10.0	100.0	100.0	100.0	100.0
Somoto	0.0	0.0	0.1	30.3	20.4	30.0	99.8
Kryptik	0.0	18.0	30.0	70.0	60.0	30.5	91.5
Multiplug	0.0	57.4	80.0	30.0	40.0	69.4	80.0
Bladabindi	0.0	1.7	5.7	4.0	7.0	10.3	93.0
Eldorado	0.0	0.0	0.0	0.0	0.0	0.0	54.4
Morstar	0.0	0.0	0.0	0.0	0.0	0.0	100.0
Preloader	0.0	0.0	7.5	71.0	50.0	60.0	100.0
SProtector	100.0	100.0	100.0	100.0	100.0	100.0	100.0
SoftPulse	0.0	4.2	4.1	6.7	5.0	6.9	86.2

3.3.1 Topic Models

Using the training set, we created a topic model for each class using HDP. We computed the statistical likelihood of drawing each particular sample from the model. Based on this likelihood we classified the samples using the already described MAP approach. Next, we executed ten crossvalidation tests with a random division into training and test set and averaged the obtained results. We also executed the equivalent tests for LDA with different number of topics in order to compare our work with this approach. Table 3.1 displays the averaged results for different malware families. These results are obtained using the supervised learning approach, however, the distribution is similar in the semi-supervised case. The outcomes justify the use of Hierarchical Dirichlet Processes over the Latent Dirichlet Allocation. More specifically, the classification accuracy is higher for most classes in case of using HDP, and in the worst case the performance is equal. This result along with the property that the HDP can automatically optimize the number of topics, gives us an adaptive and accurate classification component.

Table 3.2: Overview of main semantically relevant topics.

Registry manipulation	Memory management	File manipulation	Process Handling
NtWriteFile	VirtualAllocEx	NtReadFile	OpenProcess
RegOpenKeyExW	VirtualQueryEx	NtWriteFile	ReadProcessMemory
RegCloseKey	VirtualQuery	NtDelayExecution	WriteProcessMemory
RegEnumValueW	VirtualFreeEx	LdrGetProcedureAddress	CloseHandle
RegQueryValueExW	VirtualFree	NtSetInformationFile	LocalAlloc
LdrGetProcedureAddress	LdrGetProcedureAddress	NtCreateFile	LocalFree
RegOpenKeyExA		NtQueryDirectoryFile	

An interesting aspect of using LDA is that depending on the malware family we want to detect, we should apply different number of topics. Although the overall results reveal a significant advantage when using a higher number of topics on average, the correlation is not clear for all the families we tested. An example of this is Multiplug which exhibits better detection accuracy by selecting just ten topics, while Amonetize offers better accuracy when selecting 20 or more topics. Unfortunately, we could not detect any samples that belong to Eldorado and Morstar families using LDA. One possible reason is that we did not find the optimal number of topics for these samples.

In our experiments we noticed that the topics that were results of our topic modeling experiment often have an obvious semantic meaning. This makes our classification approach semantics-aware. Some examples of semantically meaningful topics are presented in Table 3.2. It is worth to mention that some kernel API calls belong to different topics simultaneously, which is a useful property of topic models, since activities represented by topics can consist of overlapping sets of operations.

3.3.2 Static and Dynamic Analysis Combination

Table 3.3 illustrates a comparison of classification accuracy of our three data sources, determined by executing crossvalidation with these sources separately, using a semi-supervised procedure. More specifically, we combined on the one hand the Cuckoo sandbox with HDP, and on the other hand Yara and PEInfo with label propagation. It is evident from the results that even in cases with a small number of labeled samples we can achieve a sufficient accuracy. Furthermore, we notice that each separate data source is significant for the overall performance, as none of the data sources gives maximal classification accuracy for all classes. The maximal accuracy is, however, achieved when combining the three data sources. All the methods of combining results give a high accuracy for most of the families, with slight advantage for median and majority voting. These results justify our motivation for combining multiple data sources in order to get a better performance.

Table 3.3: Comparative accuracy test using results from static and dynamic malware analysis data, separately and combined.

Family	[Cuckoo + HDP](%)	[Yara + LP](%)	[PEInfo + LP](%)	Average(%)	Median(%)	Majority(%)
Amonetize	100.0	99.6	100.0	100.0	100.0	100.0
Somoto	99.0	100.0	51.0	100.0	100.0	100.0
Kryptik	100.0	100.0	100.0	100.0	100.0	100.0
Multiplug	99.2	100.0	100.0	100.0	100.0	100.0
Bladabindi	93.2	96.6	100.0	96.6	96.6	99.0
Eldorado	56.6	80.2	83.0	84.9	86.8	81.0
Morstar	100.0	40.0	100.0	40.0	100.0	100.0
Preloader	100.0	100.0	100.0	100.0	100.0	100.0
SProtector	100.0	100.0	100.0	100.0	100.0	100.0
SoftPulse	86.1	88.8	0.0	88.8	88.8	77.8
Average	93.4	90.5	83.4	91.0	97.2	95.8

3.3.3 Comparing Supervised and Semi-Supervised Learning

We gathered results from semi-supervised and fully supervised learning. Table 3.4 shows the comparison of results of both approaches, when taking median class probability from all available data sources as classification criterion. The two colons for semi-supervised learning represent two separate experiments that we executed in order to evaluate the advantages and disadvantages of a partially labeled sample set.

The first experiment for supervised and semi-supervised methods is done using the same set of labeled examples, with the difference that in the semi-supervised case two thirds of the labeled data are used as unlabeled. We can notice that despite of using only a small number of labeled examples, we can get an adequate performance in classification. This performance is provided by our label propagation procedure, where we used the local density around the labeled points to propagate the class affiliation through the affinity matrix.

For the second experiment we used the samples for which we do not have antivirus labels as unlabeled samples and attempt to improve the classification performance. This approach shows that in most classes we can obtain a marginal improvement in the classification performance, as the unlabeled data helps in inferring the high density regions in the dataset.

Finally, we compared our classification performance with the results from related papers. Our results on average represent a significant improvement with respect to the related work in terms of average accuracy.

3.3.4 Open World vs. Closed World

We measured the performance of our closed world experiment compared to an open world situation, where not all classes are known in advance. We did this by executing the crossvalidation test, always leaving out one class from the training set. In the

Table 3.4: Performance experiment with fully supervised and semi-supervised classification models regarding the accuracy, precision, and recall.

Family	Supervised(%)			Semi-supervised(%)					
				1^{st} Experiment			2^{nd} Experiment		
	ACC	PR	RC	ACC	PR	RC	ACC	PR	RC
Amonetize	100.0	100.0	100.0	100.0	88.3	100.0	100.0	98.4	100.0
Somoto	100.0	100.0	100.0	93.6	72.2	93.3	100.0	96.8	100.0
Kryptik	100.0	100.0	100.0	100.0	86.4	100.0	100.0	100.0	100.0
Multiplug	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Bladabindi	99.4	98.1	96.0	83.5	95.4	82.9	96.6	95.8	96.6
Eldorado	75.6	26.3	86.0	31.4	98.1	31.6	86.8	98.9	86.8
Morstar	100.0	98.5	100.0	99.2	97.5	99.2	100.0	99.0	100.0
Preloader	100.0	100.0	100.0	57.1	100.0	55.4	100.0	100.0	100.0
SProtector	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
SoftPulse	64.4	75.4	87.0	49.5	51.1	50.8	88.9	86.5	88.9
Average	93.9	89.8	96.9	81.4	88.9	81.3	**97.2**	**97.5**	**97.2**
Rieck et al. [97]	**88.0**	-	-	-	-	-	-	-	-
Dahl et al. [33]	**90.5**	-	-	-	-	-	-	-	-

test phase, we classified the samples that belonged to one of the training classes with probability higher than 50% into the appropriate training class. We put the samples which did not belong to any classes with such a high probability into the "outlier" class. Our hypothesis is that the "outlier" samples will be the ones belonging to the class that is missing from the training set. This method was previously used by Rieck et al. [97], where the drop in accuracy in the open world was around 20 %. Our experiments showed that in our case, for most of the families, the performance dropped by 10 % or less. However, our system could not reliably detect the family *Eldorado* in the open setting, as the performance drop was over 40 %. This may be due to the comparatively shorter system call sequences, which makes the discrimination against other classes more difficult.

3.3.5 Time of Training

In our last experiment we wanted to measure the time of training of our approach. Therefore, we executed experiments with various number of samples and measured the time frame in which the training was complete. Figure 3.5 illustrates the distribution of the time it takes to retrain the topic models on arrival of new data points. It is noticeable that training time growth is linear, which is acceptable in online setting, considering that usually computational complexity of topic models grows not only with the number of documents, but also with the number of topics [19]. This is in part thanks to the parallelization of document-level updates.

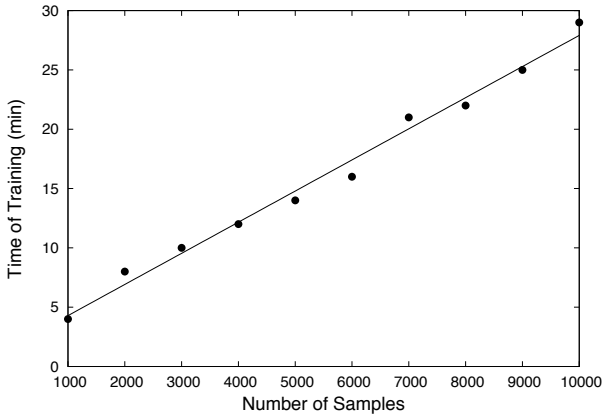


Figure 3.5: Time of training.

3.3.6 Summary of Findings

The outcomes of our experiments reveal that our methodology is adaptive, as our topic model can deal with varying number of topics and with this outperforms the standard LDA approach. Additionally, we show the semantic awareness of our method by displaying topics retrieved from system call sequences. Furthermore, we justify our approach by showing performance advantages of semi-supervised learning and joining static and dynamic analysis results in terms of classification precision and recall. Finally, we compare the performance of our approach to previous works and show improvement in classification accuracy. Overall, our approach can assist analysts by offering them a more accurate malware classification.

3.4 Limitations and Future Work

The experiments provide an insight into the performance of the methods used in our classification system. Our classification accuracy experiment on topic models shows a comparison of HDP and LDA, where HDP outperforms LDA in most classes. Nevertheless, it is also noticeable that the overall accuracy varies between different classes. For most of the classes the accuracy is high, yet there exist outliers. An explanation for this would be the overall limited reliability of the ground truth labels based on the antivirus signatures and lack of possibility of explicit evaluation of label confidence. The results could be more reliable if a more trustworthy source of labels was available. For instance, it would be useful to initially perform unsupervised learning with unknown number of clusters and in addition enhance the results with custom labeling using the analyst domain knowledge. As a difference from work done by Xiao et al. [130], we use the sequence of kernel API calls as a bag-of-words (i.e., we ignore the information about order between the calls). This gives flexibility to our model, however, it may reduce accuracy. A further study is needed to experimentally compare these two approaches.

Aside from the obtained accuracy of our classification, we are able to add another feature to our approach. This feature is the ability to extract semantics out of kernel API logs using inferred topics of Hierarchical Dirichlet Processes. Although a minority of the extracted topics has such an obvious semantic interpretation as in the presented examples, it can be very useful for a malware analyst to have such an insight.

In our evaluation, we compared results of static and dynamic malware analysis. While both static and dynamic analysis data were very useful for malware classification, the combination of the two methods proves to be the best of both worlds. Unfortunately, we had two data sources for static analysis data and only one source of dynamic analysis results. Therefore, the utilization of more data sources that provide additional data related to the program execution path, such as Drakvuf [68], would enhance the inference capability of our method.

Finally, we evaluated precision and recall of our classification and compared it with related work. Overall our system achieves a significant improvement over the previously published work in terms of classification performance, while retaining semantic model interpretation.

3.5 Summary

In this chapter, we presented an improved semi-supervised malware classification approach that joins the results from static and dynamic malware analysis to give an optimal classification performance. It uses separate algorithms for classification of static and dynamic analysis results: static analysis results are classified using a semi-supervised label propagation procedure, while the results from dynamic malware analysis are preprocessed by statistical topic modeling, which uncovers the latent semantically interpretable topics that capture the important properties of malware families. The method used for topic modeling is flexible and offers automatic adjustment of the topic set in case of online learning. Overall, our nonparametric approach creates an adaptive online system for malware classification that outperforms previous approaches.

Chapter 4

Robustness to Random and Adversarial Noise

While in the last chapter we argument using topic modeling for malware detection and classification, in order to extract semantics-aware information, in case of instruction sequences we are dealing with a different type of data. Previous work [81] shows that instruction sequences and byte sequences consist of small patches of data, whose existence helps differentiate malicious from benign binaries. Based on this result, there is a tendency to use machine learning methods that can detect these structures, such as neural networks. We examine closely this kind of approach in terms of performance and robustness to noise, especially adversarial noise potentially inserted by an attacker, trying to subvert the decision of the detector.

Parts of this chapter are published in the work of Kolosnjaji et al. about malware classification [60] and its vulnerability [59].

4.1 Related Work

Analysis of binaries statically on a byte level has been used in the work of Nataraj et al [81]. In this work they use image processing methods to extract features important for malware classification, through detecting patches of bytes that are characteristic for particular malware families. In essence, they convert malicious binary into an image in order to use methods already developed for images processing. As neural networks have been employed for image classification in recent years, there is a motivation to use them for malware analysis as well. Albeit neural networks' increasing popularity in machine learning applications, as they have caused performance improvements in many areas, these approaches have not been extensively used in malware analysis. Nevertheless, there exist efforts trying to introduce neural networks to this application area. For instance, feedforward networks have been recently utilized to analyze malware code [104], while recurrent and convolutional networks have been employed for modeling system call sequences in order to construct a "*language model*" for malware [63, 87]. Furthermore, in a publication of Raff et al. [92] the authors claim that it's possible to detect malware

by only using raw bytes as inputs and extracting features using a deep convolutional network.

The aforementioned works contain interesting efforts to automate malware detection and analysis using neural networks and show that this method has potential to help in increasing the precision and recall. However, they do not analyze how robust these methods are.

4.2 PE Files

In malware analysis and classification, it is important to know the *PE* format of binaries to properly parse and interpret the malware samples. The *PE* file format prescribes the structure of the Windows executable files, including malware [91]. More precisely, this format prescribes the organization of the *PE* File Header, which contains metainformation about the executable files. For example, in the *PE* File Header we can find which *Dynamic Link Libraries* (DLLs) are used by the considered executable and which functions are imported. This can tell us more about the intended functionality of a binary. In addition, we can see the timestamp from the compilation, names of different code sections, their sizes on disk and in memory.

After the *PE* File Header we can find the actual program. This program is divided into multiple sections, for example `.text`, `.rdata`, `.data`, and `.rsrc` section. The `.text` section contains the executable code of the program. Using data from this section can provide us a more in-depth insight on how a program operates. Next, the `.rdata` section holds the import and export information, whereas the `.data` section stores the data that is globally accessible throughout the executable. Finally, the `.rsrc` section contains the resources required by the executable, such as images, strings, icons, and menus. This can give us even more information about the purpose of the program.

It is possible to use the data from the aforementioned sections to characterize the program and enrich this with the additional data. For instance, we can use tools like *objdump* to retrieve a series of assembly language instructions, which could give us more explainable features. Furthermore, in some cases, it is also possible that this code can be further decompiled to get the source code in a higher level programming language. Although this could simplify the analysis, it does come with its own limitations. For instance, frequently there is a lack of information regarding all the compilation requirements of the binary and therefore an attempt to decompile the binary in a higher level programming language can lead to a loss of valuable metainformation. In principle, using raw binaries as input would enable us to use all the data. However, the results would be less interpretable and could contain noise that hinders the training of a malware detector.

4.3 Methodology

Our methodology for malware detection consists of gathering and preprocessing appropriate input data, feature extraction, and classification using a neural network. Furthermore, we show a method for evasion of malware detection based on these neural networks.

4.3.1 Data Acquisition

We collected a set of malware samples over multiple months from three primary sources: Virusshare, Maltrieve and private collections, similarly as in the previous chapter. However, for the work in this chapter we utilize different data acquisition tools: *objdump* [1] to retrieve the opcodes and *PEInfo* [2] to extract information from the *PE* Header.

4.3.2 Preprocessing on the Instruction-Level

We extract features of *PE* files by preprocessing the fields of the *PE* Header and opcodes from the code section. In order to use this data in the classification process, we need to create numerical feature vectors out of the already existent structure.

PE Metadata. The metadata contained in the *PE* Header offers some potentially useful input. This data is often used in malware analysis, as it is easy to just extract it from the beginning part of the binary. In this work we used the following metadata: (*i*) compile time stamp, (*ii*) address offsets to image resources (dialog boxes, menus etc), and (*iii*) size of image resources. In addition, for each code section we take (*i*) the entropy, (*ii*) the virtual address, (*iii*) the virtual size, and (*iv*) the size on disk. In total, we extract 27 numerical features.

Our motivation for extracting these features is that the *PE* metadata may help the neural network to detect suspicious aspects of a given *PE* file. For instance, if the virtual size of the `.text` section is much larger than its raw size, then it is implied that the code is packed (i.e., it has been modified using a runtime compression or encryption program). Furthermore, the set of image resources may help the neural network to identify the resources required by malware samples belonging to the same malware family.

PE Import Features. Next, we look at the list of imported functions and their DLL files. We can extract these lists from the *PE* files using the *PE* Header. To vectorize these lists, we encode the presence of unique functions along with their associated DLLs using the *one-hot* encoding approach. In total, we counted 488 unique functions from the *PE* files in our collection.

Our motivation for extracting these features is that imported functions may help the neural network to identify the purpose of a particular malware sample. For instance, functions imported from *kernel32.dll*, such as *OpenProcess*, *GetCurrentProcess*, and *GetProcessHeap*, imply that malware opens and manipulates processes. This DLL

file provides functions for most of the Win32 APIs. Many GUI manipulation imported functions, such as *RegisterClassEx*, *SetWindowText*, and *ShowWindow*, indicate that the malware has a GUI and possibly imitates the appearance of a benign GUI application. Functions imported from *shell32.dll* imply that the malware launches other programs. Subsequently, the neural network classifies malware samples with a semantically similar combination of imported functions to the same family.

Assembly Opcode Features. The last set of features is derived from the assembly instructions of the *PE* files. For each *PE* file, we first disassemble it to generate the assembly instructions with opcode information. Second, we convert the generated sequence of opcodes into a matrix representation, where each row of the matrix is a vector representation of one opcode in the sequence. To convert an opcode into a vector, we represent it by the *one-hot* encoding scheme. Our motivation for extracting these features is that opcodes may help the neural network to capture the semantics of the instruction usage patterns that a particular malware sample relies upon and thus exhibit the exact behavior of the sample. Consequently, the neural network can learn the discriminative instruction usage patterns among different benign and malware families.

Signature Clustering. We prepare the dataset for malware detection and classification using the essentially same procedure as in the Chapter 3. We select 13 clusters with the highest number of members as families for classification, since they cover most of our dataset. This gives us a different set of clusters than in Chapter 3, since more samples gave us useful data than in case of dynamic analysis. Namely, many samples only exhibit significant activity if they are ran in a specific operating system configuration. Table 4.1 illustrates the most prominent antivirus signatures included in each of the 13 clusters that we use as classes in this chapter.

4.3.3 Neural Network Architectures

We introduce two neural network architectures, used for byte-level [92] and instruction-level [60] feature extraction and further analysis.

4.3.3.1 Byte-Level Analysis

There are multiple papers describing use of raw byte data as input to a malware detector or classifier. The latest approach is the work of Raff et al [92], where they use a network with two parallel convolutional layers and a temporal max-pooling layer. We show a sketch of this neural network on Figure 4.1 and give a short description.

Let us denote with $X = \{0, \dots, 255\}$ the set of possible integer values corresponding to a byte. Then, the aforementioned network works as follows. The k bytes $(x_1, \dots, x_k) \in X^k$ extracted from the input file are padded with zeros to form an input vector x of d elements (if $k < d$, otherwise the first d bytes are only considered without padding). This ensures that the input vector provided to the network has a fixed dimensionality

Table 4.1: The most frequent malware signatures included in each of the 13 clusters.

Cluster	Signatures
0	Neurevt, QQPass, Keylogger and IRCbot
1	Zapchast and Banload
2	Artemis and IRCbot
3	Neurevt, QQPass and IRCbot
4	Dridex
5	QQPass and Keylogger
6	Banload
7	Generic
8	Genbl
9	Keylogger
10	IRCbot
11	QQPass and IRCbot
12	Androm

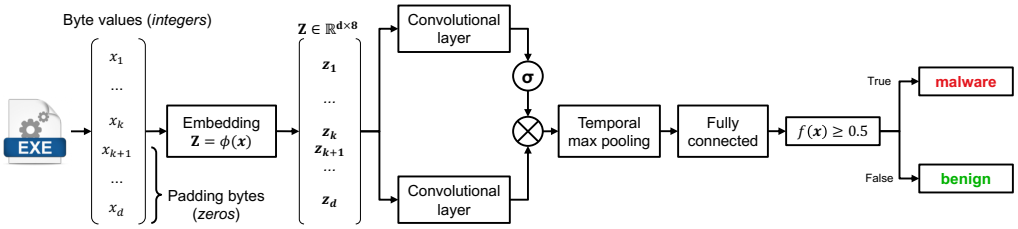


Figure 4.1: Architecture of the *MalConv* deep network for malware binary detection [92].

regardless of the length of the input file. Each byte x_j is then embedded as a vector $z_j = \phi(x_j)$ of 8 elements (through a fixed mapping ϕ learned by the network during training). This amounts to encoding x as a matrix $Z \in \mathbb{R}^{d\times 8}$. This matrix is then fed to two convolutional layers, respectively using Rectified Linear Unit (ReLU) and sigmoidal activation functions, which are subsequently combined through *gating* [34]. This mechanism multiplies element-wise the matrices that are retrieved from the two layers, to avoid the vanishing gradient problem caused by sigmoidal activation functions. The obtained values are then fed to a temporal max pooling layer which performs a 1-dimensional max pooling, followed by a fully-connected layer with ReLU activations. To avoid overfitting, Raff et al. [92] use DeCov regularization [27], which encourages a non-redundant data representation by minimizing the cross-covariance of the fully-connected layer outputs. The deep network eventually outputs the probability of x being malware, denoted in the following with $f(x)$. If $f(x) \geq 0.5$, the input file is thus classified as malware (and as benign, otherwise).

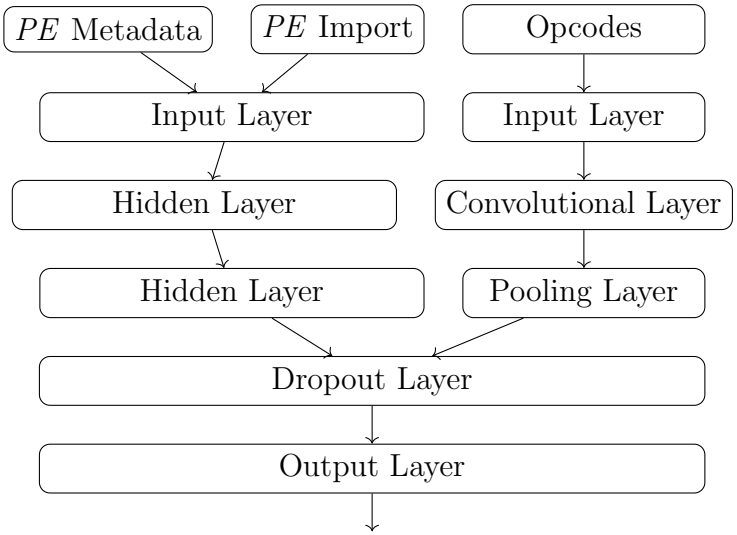


Figure 4.2: Overview of our neural network architecture.

4.3.3.2 Instruction-Level Analysis

We use a hybrid approach to design our neural network to get an adequate structure for hierarchical feature extraction. Figure 4.2 illustrates an overview of our architecture. Our intention is to use both malware opcode sequences and metadata from the PE Header. While we transform the *PE* metadata and *PE* Import features using feedforward neurons, in parallel, we leverage convolutional network layers for the malware opcode sequences.

As we mentioned, in order to construct our final neural network-based malware classifier, we combine the feedforward and convolutional neural network architectures along with their corresponding features into a single network. The second hidden layer from the feedforward neural network and the pooling layer from the convolutional neural network are aggregated into one layer. Then, a fully-connected output layer, of 13 *softmax* units is added to generate the final classification output. In the following paragraphs we provide more details about the feedforward and convolutional layers.

Feedforward Layers. We use feedforward sublayers to process the data that does not have the structure which would benefit from convolutional filters. For our inputs we are able to extract 27 features from *PE* metadata and 488 boolean features from the *PE* Import list. This data is processed by two fully-connected hidden layers, where each of the layers has 500 units, with a *Parametric Rectifier Linear Unit* (PReLU) [49] activation function. Both of the hidden layers apply 20% dropout to the hidden units. The weights of the hidden layers are initialized using Glorot's scheme [43]. Finally, after the last dropout layer, we use a fully connected output layer of 13 *softmax* units to generate the final classification output.

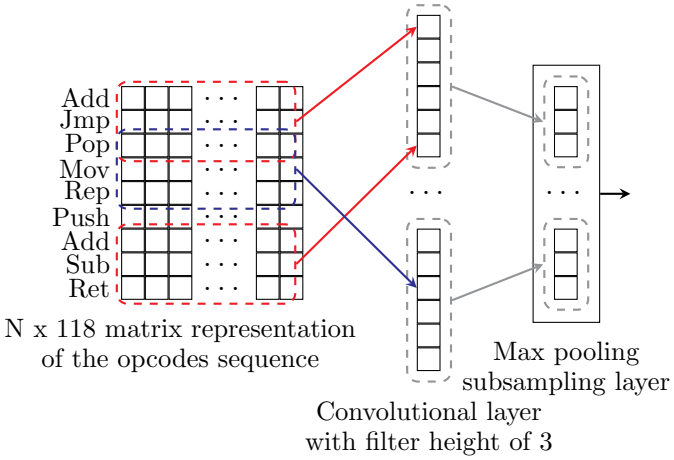


Figure 4.3: Convolutional layers.

Convolutional Layers. For the assembly opcode sequences we use different type of layers. As we want to capture the n-gram information from the instructions sequences, we employ convolutional layers to make the neural network learn this kind of high level features out of the input data. We intend to use convolution as a way of capturing the semantics of the instruction usage patterns. Figure 4.3 reveals the structure of convolutional layers. We expect that as a consequence of the good feature extraction, our convolutional architecture in the neural network will help us to discriminate usage patterns among benign and malware families. Indeed, the convolutional filter helps to discover the higher-order local features that are invariant to small changes in data. This means that our method can also help with small obfuscation done by the malware authors. Although it is not a provably optimal tool against obfuscation, we expect that this kind of architecture will help with modifications such as reordering of instructions and adding an amount of unreachable code.

Since the rows of the input matrix represent discrete opcodes, it is reasonable for filters to slide over full rows of the matrix, similar to the applications in Natural Language Processing. Thus, we choose the width of the filters to be 118, which is equal to the dimensionality of the opcode vectors. However, the height of the filters, which represents the number of adjacent opcodes, varies among different variants of our model. We mainly experimented with filter heights of 2, 3, 4 and 5, with 10 feature maps each. The dimensions of the filters are chosen by examining Natural Language Processing (NLP) application papers, where convolutional filters can detect specific patterns in the assembly instructions that are quite similar to opcode n-grams. Finally, we choose the height 3 as it gives best results in our experiments.

The convolutional layers are followed by a max-pooling subsampling layer with factor 2 in one dimension, which reduces the output dimensionality while keeping the significant global information captured by the filters. If a specific opcode sequence pattern is

reallocated in the code, information regarding whether this pattern occurred in the code will be kept, but information regarding its order in the code will be lost. Finally, we use a fully-connected output layer of 13 *softmax* units to generate the final classification output. Similar to the feedforward neural network design, PReLU was chosen as the activation function, and Glorot's scheme was selected to initialize the network weights.

The neural network is trained over shuffled mini-batches using backpropagation and stochastic gradient descent with Nesterov momentum [82].

4.4 Adversarial Attacks

In this section we describe our attack on byte-level malware detection. Furthermore, we describe some considerations about a possible attack on a neural network using the instruction-level features.

4.4.1 Attack on the Architecture with Byte-Level Features

We discuss here how to manipulate a source malware binary x_0 into an *adversarial* malware binary x by appending a set of carefully-selected bytes after the end of file. As in previous work on evasion of machine-learning algorithms [17], our attack aims to minimize the confidence associated to the malicious class (i.e., it maximizes the probability of the adversarial malware sample being classified as benign), under the constraint that $q_{\max}$ bytes can be injected. Note that, to append q_{max} bytes to x_0, we have to ensure that $k + q_{\max} \leq d$, where k is the size of x_0 (i.e., the number of informative bytes it contains) without considering the padding zeros. This means that the maximum number of bytes that can be injected by the attack is $q = \min(k + q_{\max}, d) - k$.[1] The attack can then be characterized as the following constrained optimization problem:

$$\min_{x} \quad f(x)\,, \tag{4.1}$$

$$\text{s.t.} \quad d(x, x_0) \leq q \tag{4.2}$$

where the distance function $d(x, x_0)$ counts the number of padding bytes in x_0 that are modified in x.

We solve this problem with a gradient-descent algorithm similar to that originally proposed in [17], by optimizing the padding bytes one at a time. However, the structure of data and the neural network architecture that we use for the attack imposes additional constraints. Ideally, we would like to compute the gradient of the objective function f with respect to the padding byte under optimization. However, the *MalConv* architecture is not differentiable in an end-to-end manner, as the embedding layer is essentially a lookup table that maps each input byte x_j to an 8-dimensional vector $z_j = \phi(x_j)$. We denote the embedding matrix containing all bytes with $M \in \mathbb{R}^{256\times 8}$, where the

[1] Note that $q \leq 0$ if $k \geq d$, meaning that in this case no byte can be manipulated by this attack.

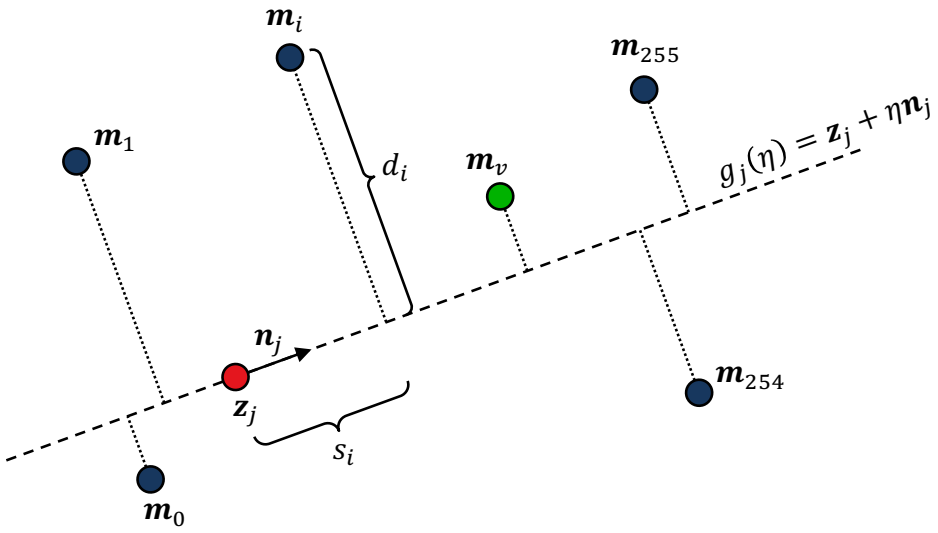


Figure 4.4: Representation of an exemplary two-dimensional byte embedding space, showing the distance d_i and the projection length s_i of each byte m_i with respect to the line $g_j(\eta)$. In this case, the padding byte z_j will be modified by the attack algorithm to m_v, as $d_v = \min_{i:s_i>0} d_i$, i.e., m_v is the closest byte with a projection on $g_j(\eta)$ aligned with n_j.

row $m_i \in \mathbb{R}^8$ represents the embedding of byte i, for $i = 0, \ldots, 255$. To overcome the non-differentiability issue of the embedding layer, we first compute the (negative) gradient of f (as we aim to minimize its value) with respect to embedded representation z_j, denoted with $w_j = -\nabla_\phi(x_j) \in \mathbb{R}^8$. We then define a line $g_j(\eta) = z_j + \eta n_j$, where $n_j = w_j/\|w_j\|_2$ is the normalized (negative) gradient direction. This line is parallel to w_j and passes through z_j. The parameter η characterizes its geometric locus, i.e., by varying $\eta \in (-\infty, \infty)$ one obtains all the points belonging to this line. Ideally, assuming that the gradient remains constant, the point z_j will be gradually shifted towards the direction n_j while minimizing f. We thus consider a good heuristic to replace the padding byte x_j with that corresponding to the embedded byte m_i closest to the line g_j, provided that its projection on the line is aligned with n_j, i.e., that $s_i = n_j^\top(m_i - z_j) > 0$. Recall that the distance of each embedded byte m_i to the line g_j can be computed as $d_i = \|m_i - (z_j + s_i \cdot n_j)\|_2$. A conceptual representation of this discretization process is shown in Fig. 4.4. This procedure is then repeated for each modifiable padding byte (starting from a random initialization), and up to a maximum number of iterations T, as described in Algorithm 1.

Although the padding bytes are generated by manipulating the input vector x, creating the corresponding executable file without corrupting the malicious functionality of the source file is quite easy, as also explained in the introductory sections and in [11]. It is however worth mentioning that our attack is general, i.e., it can be used to manipulate

Algorithm 1 Adversarial Malware Binaries

Input: x_0, the input malware (with k informative bytes, and $d-k$ padding bytes); q, the maximum number of padding bytes that can be injected (such that $k+q \leq d$); T, the maximum number of attack iterations.
Output: x': the adversarial malware example.

Set $x = x_0$.
Randomly set the first q padding bytes in x.
Initialize the iteration counter $t = 0$.
repeat
    Increase the iteration counter $t \leftarrow t + 1$.
    for $p = 1, \ldots, q$ **do**
        Set $j = p + k$ to index the padding bytes.
        Compute the gradient $w_j = -\nabla_\phi(x_j)$.
        Set $n_j = w_j / \|w_j\|_2$.
        for $i = 0, \ldots, 255$ **do**
            Compute $s_i = n_j^\top (m_i - z_j)$.
            Compute $d_i = \|m_i - (z_j + s_i \cdot n_j)\|_2$.
        Set x_j to $\arg\min_{i:s_i>0} d_i$.
until $f(x) < 0.5$ or $t \geq T$
return x'

any byte within the input file. To this end, it suffices to identify which bytes can be manipulated without affecting the file functionality, and optimize them (instead of optimizing only the padding bytes).

4.4.2 Attack on the Architecture with Instruction-Level Features

We also consider an attack on the classifier based on the instruction sequences as inputs, described in the previous section. Similarly to the byte-level attack, we can define the attacker's power as the ability to add arbitrary instructions at the end of the instruction sequences. These instruction sequences do not endanger the existing functionality of the binary. However, they may be adding some unnecessary execution sequence. Furthermore, there may be considerable additional effort necessary to recreate the binary after these modifications.

We can define the gradient-based attack procedure similarly to the byte-level experiment. However, since in the instruction-level experiment there is no embedding layer, we imagine the attacker optimizing the attack by updating the instruction sequences using direct gradient-based update with one additional step. This step determines the highest value in the encoding vector for each instruction and sets it to 1, while all the other values are set to zero. With this assignment we get an updated one-hot encoded vector.

The attack with this heuristics enables us to optimize the instruction sequences to increase the softmax component for the benign class. We plan to conduct the evaluation of this attack as future work.

4.5 Evaluation

In this section we show and comment on the performance of our neural networks and the results of our attack approach.

4.5.1 Evaluation of Byte-Level Analysis

We practically reproduced the deep neural network proposed in [92], and performed the evasion attacks according to the algorithm described in Sect. 4.4. In the following, we first describe the employed setup, and then we discuss the results obtained by comparing the efficiency of the proposed gradient-based method with trivial random byte addition.

Dataset. We employed a dataset composed of 9,195 malware samples, which were retrieved from a number of sources including VirusShare, Citadel and APT1. Additionally, to evaluate the performance of the network we employed 4,000 benign samples, randomly retrieved and downloaded from popular search engines.

Network Performance. We evaluated the performance of the deep neural network by splitting our dataset into a training and a test set, each of them containing 50% of the samples of the initial dataset. To avoid results that could be biased by a specific training-test division, we repeated this process three times and averaged the results. Under this setting, we obtained an average precision of $92.83 \pm 5.56\%$ and an average recall of $84.68 \pm 11.71\%$ (mean and standard deviation).

We performed our tests by modifying 200 randomly-chosen malicious test samples with Algorithm 1 to generate the corresponding *adversarial malware binaries*. As for Algorithm 1, we set the maximum number of attack iterations as $T = 20$, and the maximum number of injected bytes $q_{max} = 10,000$. As a result, we chose all malware samples that satisfied the condition $k + q_{max} \leq d$, where k is the file size and $d = 10^6$. The attack was performed by appending, at the end of each file, bytes that were chosen according to two different strategies: a *random* attack injecting random byte values, and our *gradient-based* attack strategy. To verify the efficacy of the attack, we measured for each amount of added bytes the average *evasion rate*, i.e., the percentage of malicious samples that managed to evade the network. Fig. 4.5 provides the attained results as the number of bytes progressively increases, averaged on the three aforementioned training-test splits. Notably, adding random bytes is not really effective to evade the network. Conversely, our gradient-based attack allows evading *MalConv* in 60% of the cases when 10,000 padding bytes are modified, even if this amounts to manipulating less than 1% of the input bytes.

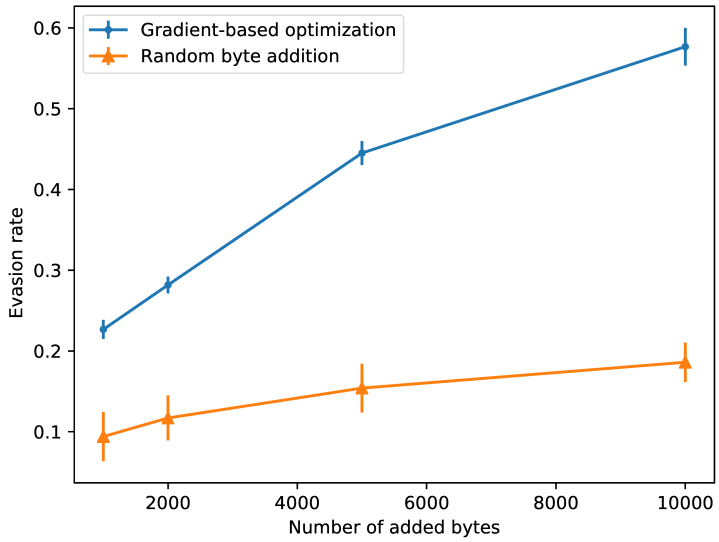


Figure 4.5: Evasion rate against number of injected bytes.

The success of our gradient-based approach relies on the fact that it guides the decision of which bytes to add, thus creating an organized *padding byte pattern* specific to each sample. To better clarify this concept, in Fig. 4.6 we consider a sample that successfully evaded the network, and show the distribution of the $10,000$ bytes added by the two attacks. Note how, in the optimized case, only a small group of byte values is consistently injected. This shows that the gradient guides the choice of specific byte values that are repeatedly injected, identifying a clear *padding byte pattern* for evasion.

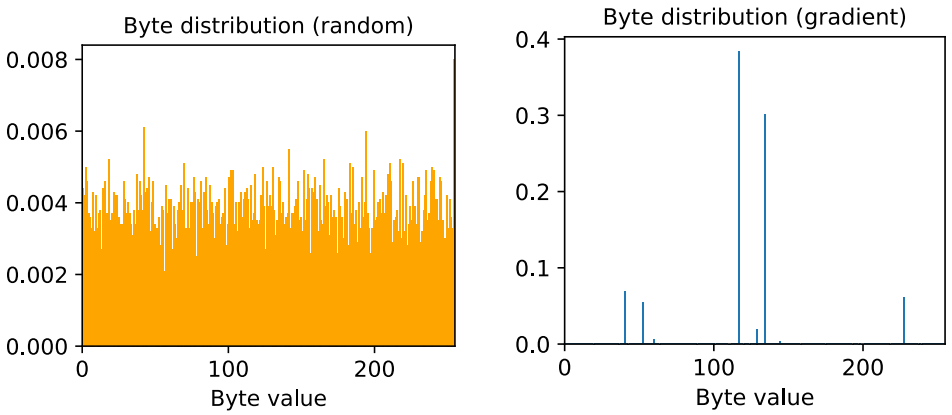


Figure 4.6: Distribution of the $10,000$ padding byte values injected by the random (*left*) and gradient-based (*right*) attacks into a randomly-picked malware sample.

Table 4.2: Classification performance of the feedforward, convolutional, and hybrid neural networks as well as the SVM for instruction sequences as inputs.

Model	Precision	Recall	F1-score
FFNN	0.90	0.92	0.90
CNN	0.92	0.92	0.91
SVM	0.92	0.92	0.92
Hybrid NN	0.93	0.93	0.92

4.5.2 Evaluation of Instruction-Level Analysis

Next, we first discuss about our crossvalidation experiments for instruction-level analysis and then present our classification results. Furthermore, we show how our approach is robust against simple instruction reordering.

4.5.3 Crossvalidation Experiments

We conducted a set of crossvalidation experiments to assess how well each of the feedforward and convolutional neural networks individually perform, and then we evaluated the performance of the hybrid neural network. We ran the experiments on a machine with a *CUDA* graphical processing unit. In addition, we built and trained the neural network models using the Tensorflow [6] library. Our final dataset after noise filtering contains 22,757 executables, with 22,694 malicious executables and 63 benign executables from ZDNet's download list of most popular programs [5] ("*benign*" class). One may notice that we ended up with unbalanced datasets. However, since we use multiclass classification, where we have one benign class and multiple classes of malicious executables, this reduces the problem with balance among class sizes.

In detail, we conducted 3-fold crossvalidation experiments to estimate the results over new data. In each experiment, we randomly split the dataset into three equally sized partitions, where we trained against two partitions and tested against the remaining one. This process was repeated for three times while a different partition is left out for testing each time. Finally, we computed the average results of the three tests to obtain a reliable measure of how well the proposed neural network architecture performs over the entire dataset. We quantitatively assessed the performance of the neural network architecture using three metrics: *precision*, *recall*, and *F1-score*.

To measure the performance gain brought by our combined deep neural network, we trained a reference support vector machine (SVM) classifier using the *PE* metadata, *PE* import, and assembly opcode features. In detail, we used the same set of *PE* metadata and import bag-of words features. However, in the case of the assembly opcodes of a *PE* file we used counts of 3-grams of opcode instructions to get a methodology for

comparison with convolutional neural networks.

4.5.4 Classification Results

We compared the averages of the classification performance of the fully feedforward, convolutional, and our neural network as well as the support vector machine, as shown in Table 4.2. Our hybrid neural network provides a performance improvement over the feedforward and convolutional neural networks as well as the support vector machine. The hybrid neural network achieves an F1-score value of 0.92 as well as precision and recall values of 0.93.

The experiments demonstrated that our approach accurately classifies malicious executables that behave similarly into the same family. Table 4.4 depicts the averages of the classification performance for each of the classes of the hybrid neural network. Table 4.3, on the other hand, shows the confusion matrix of the hybrid neural network, which illustrates how malware samples of a particular family are correctly classified, or misclassified into different families. In detail, Table 4.4 reveals that samples belonging to classes 1, 4, 5, and 7 as well as the benign samples are approximately correctly classified. In contrast, samples belonging to classes 0, 2, 10, and 11 are misclassified. Furthermore, the confusion matrix of the deep neural network, shown on Table 4.3, indicates that malware samples belonging to classes 0, 2, and 11 are particularly confused with class 3. The reason behind this is that these classes share common malware signatures, as demonstrated in Table 4.1.

Next, we examined the activation values of the input neurons at the input layers in order to determine which features have higher activation values after training. For instance, Figure 4.7 presents the average activation values of the input features for ten malware samples of class 1. It is interesting to see that the most active neurons at the input layer are the assembly opcode features. This indicates that in our tests the assembly opcode features make the most significant contribution to the malware classification.

Furthermore, the high-dimensional activations of the neurons at the last hidden layer were projected to a lower dimensional space for visualization and analysis. As shown in Figure 4.8, the feature points for classes 1, 3, 4, 5, 8, 12, and 13 are well separated. However, there is confusion among classes 6, 7, 9, and 10. To keep the distances between activation values of different samples while reducing the vectors to 2D we used the T-SNE [119] method for dimension reduction.

4.5.5 Robustness Against Instruction Reordering

Attackers often create malware variants by reordering the instructions to bypass signature detection. The instructions are reordered in such a way that preserves the inter-instruction dependencies. Hence, in this experiment, we attempt to illustrate the robustness of the neural network against instruction reordering. Given 150 test samples, we

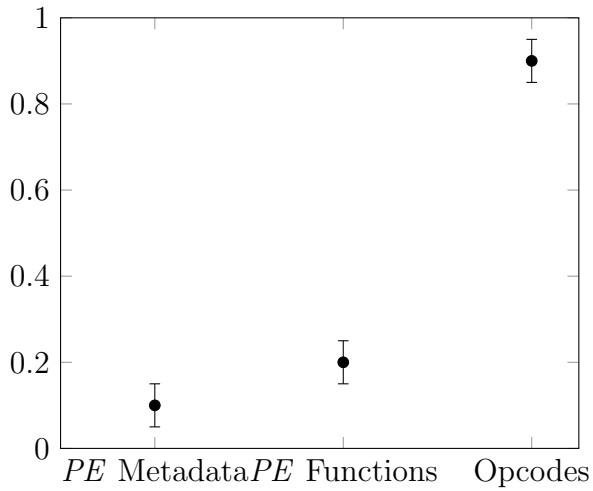


Figure 4.7: Visualizing the average activation values of the input features for ten malware samples of class 1.

randomly shuffled their corresponding sequences of opcode and measured the neural network performance. Our neural network was still able to accurately classify the malware and benign samples with F1-score values above 0.90 until 50% rate of shuffling. Additionally, we randomly increased or decreased the input frequencies of 3-gram opcodes to the support vector machine for these 150 test samples and measured the support vector machine performance. The support vector machine performance has dramatically decreased to an F1-score of value 0.82. Figure 4.9 shows the small decrease in the F1-score of the hybrid neural network while increasing the rate of instruction shuffling. In addition, it shows the significant decrease in the F1-score of the support vector machine while increasing the rate of 3-grams' distribution alteration. This shows that our methodology contains a defense mechanism against the above mentioned obfuscation approaches.

4.5.6 Saliency Maps

In order to visualize and interpret the classification process of our neural network, we use the methodology developed by Symonian et al. [110]. In particular, we use Taylor expansion and compute first partial derivative of the classification results before putting them through the softmax function. This gives us a detector of salient features that contribute the most to attributing samples with a certain class. Out of a saliency map we can draw conclusions about the reasons for classification results. For example, in Figure 4.10 we can see that the chosen sample should be given a label 7, according to most of the *PEInfo* features. Yet, there is a significant peak at the feature with number 34 that indicates that this sample also has some characteristics of the class 10 (i.e., IRCBot). This feature is the size of a certain *PE RESOURCE* field in the *PE Header*.

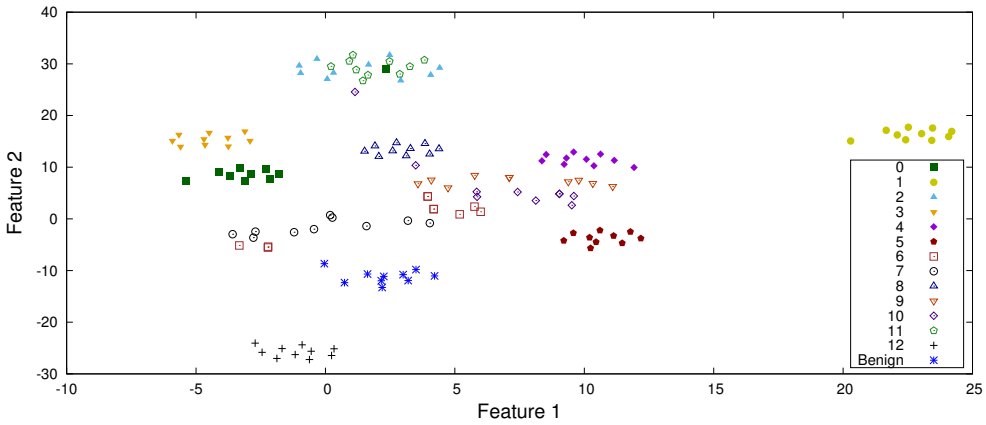


Figure 4.8: Visualizing ten samples from each class with their corresponding activation values of the last hidden layer neurons.

4.6 Limitations and Future Work

Although we have, in principle, achieved our goal of leveraging convolutional networks to improve malware classification, there are some limitations to our methodology. Regarding the dataset, we only use results of static malware analysis. Although we show that our convolutional layers are somewhat robust to code obfuscation, there still are ways to obfuscate the code to an extent that our methodology will not work. For example, the malware samples may be packed with a custom packer or contain code that gets decrypted in execution. In dynamic analysis, malware samples are executed in a protected environment and traces of actual malware behavior can be extracted. Optimal methodology would be to use both static features and behavioral traces to gain knowledge from both views. Since multiple views on malware provide different aspects of malware characteristics, one machine learning method cannot be optimal for this heterogeneous data. Previous work has shown a multiple kernel learning approach to unifying models based on different aspects of malware characteristics [10].

The second limitation is that we use a closed world assumption, i.e., we do not consider how appearance of malware from a family unknown to our system will affect the performance. Finally, in our methodology we use convolutional filters to capture information from opcode sequences. However, we do not create an explicitly sequential model. One possible improvement would be to create such a model using fully recurrent networks. This may be difficult because recurrent networks are hard to train efficiently. However, future work can be directed to tackle this problem.

We also discuss here some limitations related to our adversarial attack analysis. First, in comparison to [92], we employed a smaller dataset, and we considered an input file size d of 10^6 rather than $2 \cdot 10^6$. These are both factors that may facilitate evasion of *MalConv*. Conversely, we found that appending bytes to the end of the file reduces the effectiveness of the gradient-based approach. To better realize this, in Fig. 4.11 we

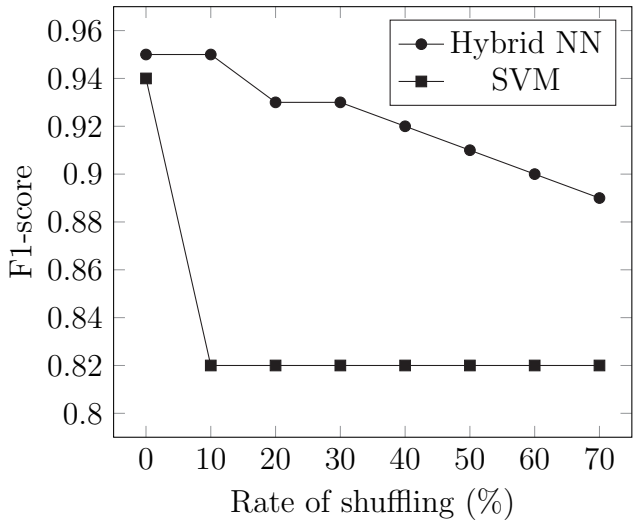


Figure 4.9: Demonstrating the decrease in the F1-score for the hybrid neural network and support vector machine while increasing the rate of instruction reordering.

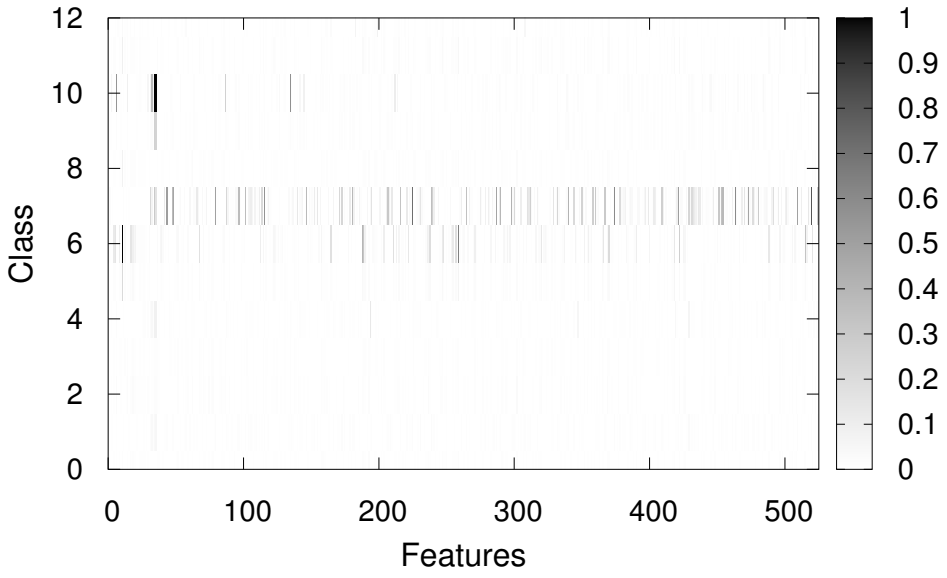


Figure 4.10: Saliency map for a chosen sample and PEInfo features.

show that the average norm of the gradient w computed over all attack samples is much higher for the *first bytes* in the file. This is reasonable, as files have different lengths, and the probability of finding informative (non-padding) bytes for discriminating malware and benign files decreases as we move away from the first bytes. From the attacker's perspective, this also means that modifying the first bytes may cause a much larger decrease of $f(x)$ and, consequently, a much higher probability of evasion. However, as described in the introductory sections, modifying bytes within the file may be quite

complex, depending on the specific file and the content of its sections. This is definitely an interesting avenue for future research in this area.

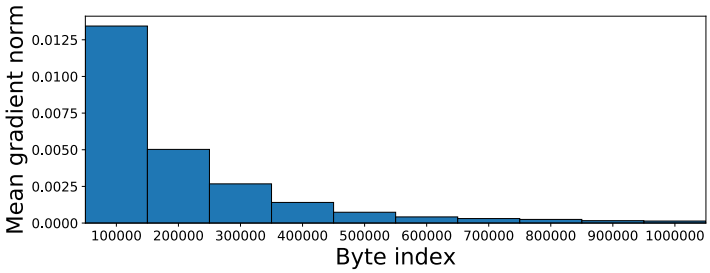


Figure 4.11: Mean gradient norm (per byte) over all attack samples.

Furthermore, the adversarial attack that we describe in this chapter opens up new issues. These results question the adequateness of byte-based analysis from an adversarial perspective. In particular, the use of deep learning on raw byte sequences may give rise to novel security vulnerabilities. Binary-based approaches are usually based on the hypothesis that all sections have the same importance from the *learning perspective*. However, such claim is challenged by the fact that there are typically strong semantic differences between sections containing instructions (e.g., `text`) and those containing, for example, debug information. Hence, performing manipulations directly on the targeted files might be easier than expected.

In future work, we plan to particularly investigate this issue, by exploring fine-grained, automatic changes to executables that may be more difficult to counter than the injection of padding bytes at the end of file. We also plan to repeat the assessment of this paper on a larger dataset, more representative of recent malware trends (as advocated by Rossow et al. [101]). We anyway believe that our work highlights a severe vulnerability of deep learning-based malware detectors trained on raw bytes, highlighting the need for developing more robust and principled detection methods. Notably, recent research on the interpretability of machine-learning algorithms may also offer interesting insights towards this goal [36, 72].

4.7 Summary

In this work, we investigate neural network-based approaches for malware detection and classification based on static malware features. At first, we construct a neural network based on the work of Raff et al. [92] that works with raw byte values as input and evaluate its work in an adversarial environment. We proposed a general gradient-based approach that chooses which bytes should be modified in order to change the classifier decision. We applied it by injecting a small number of optimized bytes at the end of a set of malicious samples, and we used them to attack the *MalConv* network architecture, attaining a maximum evasion rate of 60%. Second, we design and evaluate a neural

network that uses metadata of *PE* files, imported functions, and series of opcodes to separate malicious executables from benign programs and classify malware into 13 predefined classes. Furthermore, it differentiates benign executable files from the predefined malicious classes. Its architecture that combines convolutional and feedforward layers provides improvements in classification performance in comparison to baseline neural networks and support vector machines. We also provide preliminary considerations on robustness of this network, which we intend to investigate further in the future.

Table 4.3: The confusion matrix of the deep neural network.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**Benign**
0	26	0	0	119	0	0	0	0	0	0	0	0	0	0
1	0	100	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	267	0	0	0	0	0	0	0	0	0	0
3	0	0	0	504	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	157	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	869	0	4	0	2	0	0	0	0
6	0	0	0	0	0	0	70	170	14	0	0	0	12	0
7	0	0	0	200	25	2	14	18572	245	16	10	0	180	0
8	0	0	0	0	0	0	0	0	430	0	0	0	0	0
9	0	0	0	0	4	2	2	20	0	98	0	0	4	0
10	0	0	0	0	0	0	0	112	19	5	12	0	0	0
11	0	0	0	129	0	65	0	0	1	0	0	0	0	0
12	0	0	0	0	0	0	0	66	0	0	0	0	145	0
Benign	0	0	0	0	0	0	0	4	0	0	0	0	0	59

Table 4.4: Classification performance of our neural network for instruction sequences, for each of the classes.

Class	**Precision**	**Recall**	**F1-score**
0	1.00	0.16	0.27
1	1.00	1.00	1.00
2	0.00	0.00	0.00
3	0.41	1.00	0.59
4	0.93	1.00	0.96
5	0.92	0.99	0.95
6	0.87	0.31	0.46
7	0.98	0.96	0.97
8	0.56	1.00	0.72
9	0.78	0.75	0.77
10	0.67	0.10	0.18
11	0.00	0.00	0.00
12	0.50	0.80	0.61
Benign	1.00	0.92	0.96
Average	0.93	0.93	0.92

Chapter 5

Low-Budget Authentication

While the Chapters 3 and 4 describe problems of dealing with maintaining anomaly detectors in case of using large-scale machine learning for malware detection and analysis, in this chapter we introduce the problem of resource constraints on small devices. In particular, we consider online training of a resource-constrained model on a mobile device. We use this model to improve behavior-based authentication using sensor data.

We leverage a well-known method: One-Class Support Vector Machine [106] and adapt it to our scenario of learning on a budget. Parts of this chapter have been published in the paper of Kolosnjaji et al. about budgeted learning for sensor-based user authentication [61]. Our budgeted One-Class SVM approach also extends for other anomaly detection scenarios, such as fault detection, in environments where we have constraints on memory and processing.

5.1 Related Work

Anomaly detection methods often need to be designed for devices with constrained memory and processing power, for example mobile devices. This especially affects methods where the number of parameters grows with data. One-Class Support Vector Machine with a Gaussian Kernel is a very common example as a method of choice for anomaly detection. One instance of resource-constrained anomaly detection problem is user authentication for mobile devices. This section will put our work in context of published related papers on behavior-based user authentication, focusing on mobile devices.

For instance, Gascon et al. [42] study classification of users by determining keystroke dynamics based on sensor data only. They obtain promising results with a false acceptance rate (FAR) of only 1% for some users. However, they encounter that other users show a barely characteristic behavior, making it difficult to model their user profile. On the other hand, Buschek et al. [22] show improvements in classification accuracy based on combining temporal and spatial features for keystroke dynamics. Their work focused on password entries only, and it would be interesting to extend it on multiple types of

inputs. Interestingly, the same paper contains evaluation of model performance obtained within and across temporal varying sessions. Results prove that user behavior varies over time, which confirms the need for a continuously updated model. Xu et al. [131] execute a 21-day experiment in order to measure long-term performance of sensor-based feature extraction. Furthermore, their study is not limited to one touch gesture, but includes keystroke, swipe, handwriting and pinch gestures. They execute an experiment where they modeled the user behavior based only on each user's first day, and they observed the change in performance when testing for the following days. Offline training of a model can therefore be assumed to have limited reliability, as the behavior patterns can change. Frank et al. [40] have investigated the swipe gestures as basis for classification of users of mobile devices. They separate their setup in an enrollment phase to train their model before utilizing it for classification in the continuous authentication phase. Velten et al. [120] were able to correctly classify users with both FAR and false rejection rate (FRR) lower than 1%, based on 14 swipes only. Swipe data had been gathered whenever using the device's browser. Their work has confirmed helpfulness of acceleration features for classification. While their model setup relied on user data being sent to a remote web server, our approach focuses on an authentication model working on the phone only. Murmuria et al. [79] model user behavior similarly to previous approaches, with an additional consideration for the application context. Similarly, in this chapter our work will also evaluate influence of differing application contexts on swipe gestures. The feature set of Murmuria et al. has been based on heterogeneous data: power modalities, sensor and touch data.

Previous work has not considered the size of the parameter set when implementing online training for mobile sensor-based authentication, and we attempt to optimize our modeling strategy for this condition. This would enable an anomaly detection model with a smaller memory footprint and faster updates in an online learning scenario, while maintaining anomaly detection performance.

5.2 Methodology

In our methodology we consider the situation where we gather sensor data and train our model incrementally using a valid user and then, in the test time, have both a valid user and an attacker trying to use the device. Both the attacker and the valid user leave a trace of their activity in the form of touchscreen interaction, accelerometer, and rotation sensor values. The goal of our system is to detect the attacker's behavior as a significant deviation from the model based on the valid user, while letting the valid user use the device without unnecessary interruption. Therefore, we consider our problem as an example of unsupervised anomaly detection problems. Our anomaly detection system is displayed on Figure 5.1.

This scheme consists of data collection, feature selection and an anomaly detection model. Upon detecting an anomaly, we foresee the restrictive policy of locking the phone

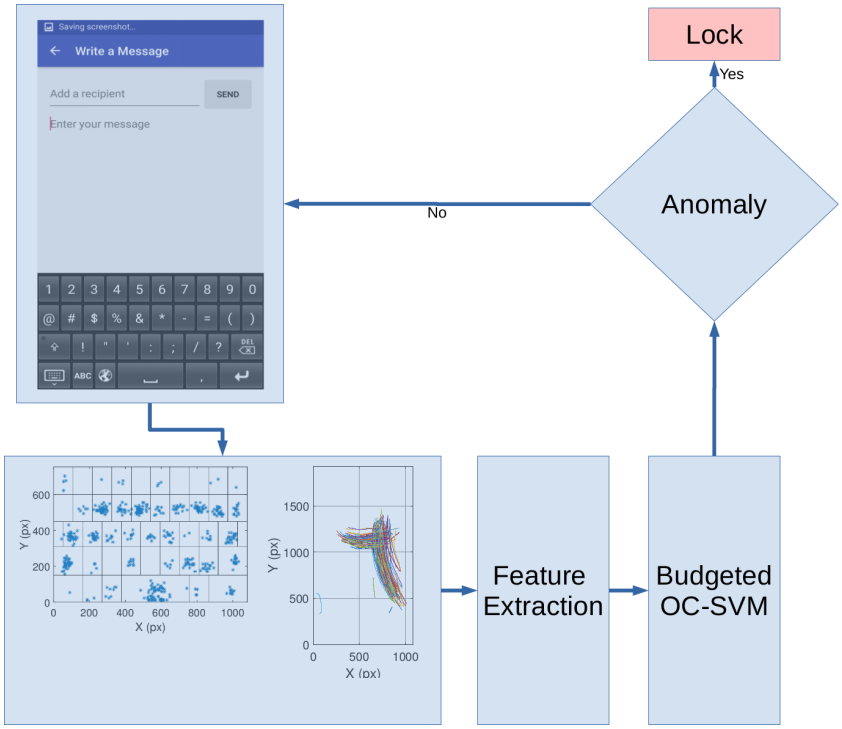


Figure 5.1: Scheme for low budget sensor-based authentication

and turning to other, more deterministic authentication methods.

5.2.1 Experiment Design

To gather user-specific data, we design a two-part experiment focusing on different typical activities on the smartphone.

Requested Tasks in Our Mobile App. We request users to perform a series of tasks in order to gather data for training of our model. A potential future app, delivered directly from the smartphone vendor, could offer the option for users to initially pretrain their smartphones with their behavior, if they wish to enable the continuous authentication feature. Another possibility is for the user to perform the tasks presented in our app under the hood. This means that the phone does not need to execute an application, where it requests input from the user, but it can stealthily collect data and train the models. The tasks that we use for collecting continuous authentication data in our app are the following: *Image Finding*, *Text Reading*, *Zooming*, *Message Composition*, *Entering a PIN* and *Phone Call*. We design these tasks taking into account the most often performed activities on a smartphone. Each task is repeated ten times per participant, which lowers the impact of sudden change of behavior because of external conditions (e.g., stress).

Tasks with Built-in Apps. We execute the next series of tasks using the apps provided by the mobile devices themselves. This makes the behavioral traces of users more natural, as they regularly use this kind of apps and can perform the tasks more smoothly. The tasks in this part were: *Browsing*, where a user needs to find answers to three questions using Google and Wikipedia, and *Taking a Selfie* using the smartphone's camera.

5.2.2 Data Collection

We used the *Samsung Galaxy S4* smartphones with Android 5.0 to execute the above described experiments with 28 participants in order to capture the discriminative features in user activity. The Android API delivers a sequence of motion events to the currently active application each time a user touches the screen. The touch sequence is started by a down event followed by a variable number of move events and an up event closing the sequence. Each motion event has an event ID, followed by its timestamp and its x and y center coordinate, as well as major and minor axis of the ellipse making up the touch contact. Apart from the standard motion event data, for keyboard typing events we can record the keycode of each key hit. For security reasons our version of Android does not allow to capture keyboard input of the standard keyboard. For this purpose we develop a keyboard that resembled the Samsung keyboard, and run it as an Android service. Users are instructed to choose the custom keyboard when opening the app such that no other keyboard would accidentally be used.

In addition to touch data, sensor data from built-in accelerometer and gyroscope (rotation sensor) are harvested. Data are polled at a rate of five times a second. The accelerometer measures acceleration in m/s^2, while the gyroscope measures rate of rotation in rad/s along x, y and z axis of the phone. Gathering sensor data is done by running an Android service, same as in case of the custom keyboard.

Android only allows capturing touch events from within a custom application. Therefore, it is not possible to gather motion events from the Android API while using standard applications for the second part of the experiment. Nevertheless, it is possible to directly read from the input device driver under /dev/input/eventX, where X is a number identifying the input device.

5.2.3 Feature Engineering and Selection

Based on the ideas from related work, we capture a large range of spatio-temporal features to determine which subset is the most useful. Initially there are 59 features from swipe gestures and 51 features from keystrokes. The swipe feature set contains data that characterize the trajectory of the swipe motion, the velocity of movement, and size of the trace. On the other hand, the keystroke features contain the similar velocity and stroke size patterns as well as keycode and offsets to the key center. Our feature set is based on the works of Velten et al [120], Frank et al. [73], and Buschek

et al. [22]. Out of this feature set we select most relevant ones based on the sequential feature selection procedure of Ruckstiess et al. [102].

5.2.4 Budgeted Anomaly Detection

Our goal is to detect invalid users based on the gathered data. Therefore we leverage a method of anomaly detection called One-Class Support Vector Machine (OC-SVM) [106]. OC-SVM is an anomaly detection method similar to the well-known Support Vector Machine for classification. This method enables us to determine a region where most of the data points lie, giving them the label `+1`. On the other hand, the rest of the points are considered anomalies (labeled as `-1`). We optimize the margin between these two regions of points in order to have the most accurate boundary. In the primal form this method can be represented with the following formula:

$$\min \frac{1}{2}\|w\|^2 + \frac{1}{\nu N}\sum_i \xi_i - \rho \tag{5.1}$$

subject to:

$$(w\Phi(x_i)) \geq \rho - \xi_i, \xi_i > 0 \tag{5.2}$$

The parameter w defines the position and orientation of the classification boundary, while the value of ξ is tuned to increase robustness to possible noise in the labeling process. The value of N is the size of the sample set and the constant ν is used for tuning the percentage of outliers.

In order to easier define the budget saving strategy, we use a dual form of the decision function:

$$f(x) = sgn(\sum_i \alpha_i K(x_i, x) - \rho) \tag{5.3}$$

where the value of ρ is tuned to get the best detection performance on a certain data distribution. The dual form enables us to use a kernelized version of the distance value ($K(x_i, x)$). This means that we can freely select the distance measure between points (K). If we use a linear distance measure, we can only optimize a linear decision boundary. Since our samples are not linearly separable, we use the Gaussian RBF kernel, in order to have a more precise separation of points.

When using a dual form of One-Class SVM, we solve the following optimization problem:

$$min_\alpha \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j k(x_i, x_j) \tag{5.4}$$

subject to:

$$0 \leq \alpha_i \leq \frac{1}{vl}, \sum_i \alpha_i = 1 \tag{5.5}$$

This is a quadratic problem with linear constraints. Instead of solving this problem with a usual quadratic programming procedure, for online learning we use gradient descent, where the gradient looks like the following:

$$\frac{\partial g(x)}{\partial \alpha_i} = \frac{1}{2} K_i \alpha_i \tag{5.6}$$

For each new point that arrives, we execute 100 gradient descent steps to update our model, with learning rate $\lambda = 0.0001$. After executing the model update, we execute the maintenance to make sure that the parameter set does not increase beyond the budget limits. There are multiple possible strategies for this maintenance. In the work of Wang et al. [125] three strategies are defined: deletion, projection, and merging. After testing all of them, we decided to use the deletion method, as it provides approximately the same results as other methods, but with faster execution, which is very important in online learning. In the deletion method, we simply remove smallest values of α_i in order to fit to the budget limit.

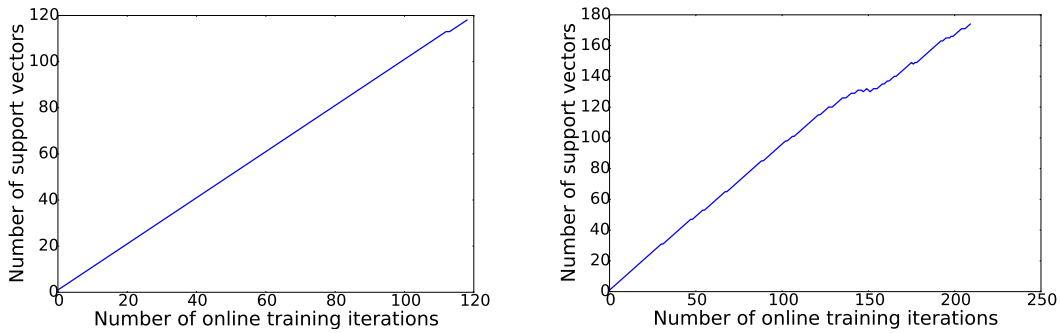
Since we have two broad feature sets (i.e. swipe gestures and keyboard writing), we build two separate models using the previously described procedures. For each new relevant event we first detect the type of event based on the values that we can read from the sensors and then run the authentication process using the appropriate model.

5.3 Evaluation

We tested our procedure using the data gathered from our experiments, with a type of crossvalidation. All the gathered data from querying touch screen, keyboard, acceleration, and gyroscope sensors were turned into feature points and the order of actions from the experiment was randomized. The randomized sets of points were divided into training (70%) and test sets (30%). The results had been averaged over 5 runs to make them more reliable. At first we do this test separately for swipe and keyboard gestures in order to investigate the model behavior for the two sets of features. We test the model performance in an unconstrained scenario, followed by tests with different budget size limits. Afterwards, we execute a joint test to simulate the real-world behavior and analyze the model performance for different test users.

5.3.1 Model Growth in an Unconstrained Scenario

First, we show a graph of the model growth suitable to our scenario of using One-Class Support Vector Machine. Figure 5.2 shows that in an unconstrained scenario the model grows almost linearly. While we can only show growth in the support vector set for the limited sequence of training samples we used in crossvalidation, the model would keep receiving a continuous stream of samples in reality, thereby growing unlimitedly. This clearly shows demand for a limit on model size as we would otherwise gradually reserve



(a) Swipe model **(b)** Keyboard model

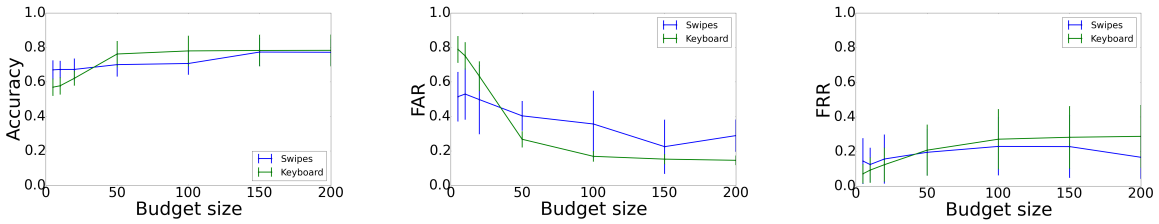
Figure 5.2: Increase in the model size for increase in training samples presented to the model for keyboard features.

more and more memory on users' phones. Not only is this inconvenient in terms of memory footprint, but it would also build up a very complex model, thereby increasing computational cost, which might in turn yield a reduced operating speed. Besides, old support vectors kept in the model may not be reflecting current user behavior. While they unnecessarily reserve memory, they may also degrade performance of the model.

5.3.2 Budget Size

The next group of plots, displayed on Figure 5.3 shows the change of results for swipe and keyboard features with the increase in the available support vector budget. On the first plot the accuracy grows very fast with the number of available support vectors, achieving 95% of the maximum accuracy with only a half of maximum support vectors as a limitation. This shows that we can place a demanding limit on the size of the model and still achieve high accuracy. Furthermore, the next plot shows fast decrease in false acceptance rate with the number of available support vectors. This means that even for a small budget we can get an authentication system where attacks will have a high detection rate. On the other hand, false rejection rate is actually growing with the number of available support vectors. This means that in terms of this particular criteria, the performance does not improve with the available budgets. Possible reason for that is that more complex model tends to have high bias towards the training set. This means that more initial training data may be necessary for more robust models.

We also show similar results for the keyboard features. However, in the case of keyboard features the growth of accuracy is not as fast and we need around 150 vectors to achieve the maximum performance, as a difference from the swipe features, where we only need around 100. In terms of FAR and FRR, the test shows similar trends to the results with swipe features.



(a) Accuracy **(b)** FAR **(c)** FRR

Figure 5.3: Results for different budget constraints with swipe and keyboard features

5.3.3 Confusion Matrix

The confusion matrix on Figure 5.4 contains the information on the mean accuracy when using various pairs of training users and attackers. This figure shows that for the most pairs we can achieve high accuracy in recognizing the attack. However, for some users the performance is still problematic. In particular, for the users 8, 13, and 26 we cannot properly differentiate the attackers, because the models are not discriminative enough. By gathering more data this performance can be improved.

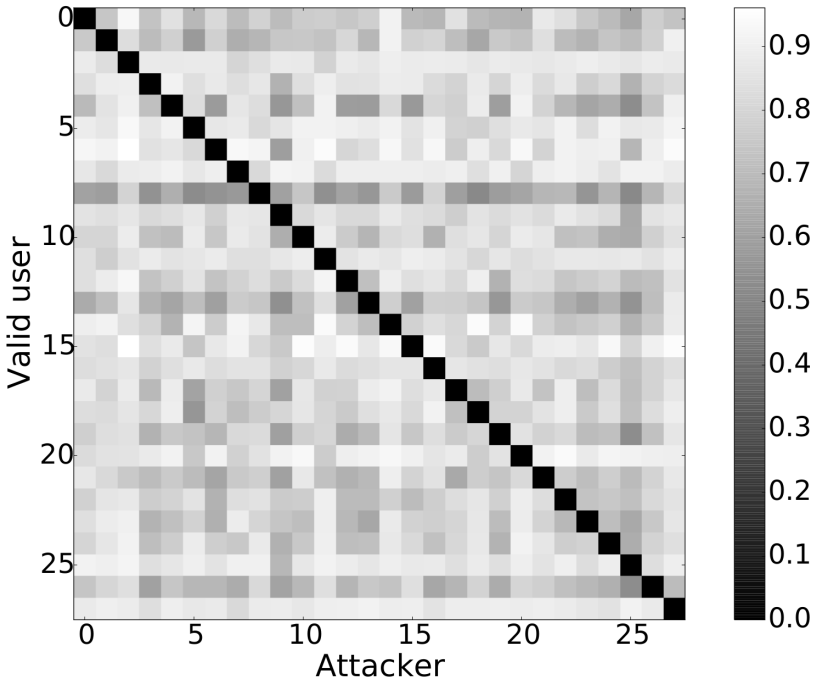


Figure 5.4: Confusion Matrix.

5.4 Summary

In this chapter we propose a budget-efficient model for continuous authentication on mobile devices, using the data from touchscreen interaction as well as the values from accelerometer and rotation sensor. We develop an online budgeted version of the well-known OC-SVM algorithm and train it using gradient descent in an online manner. We execute the training and testing experiment on a dataset from behavioral traces that we gather from 28 subjects. This enables us to achieve performance highly comparable to the results using unlimited models, while reducing the model size by more than 50%.

While we use the resource-constrained One-Class SVM on a particular scenario of sensor-based authentication, it can also be used in any other anomaly detection scenario where we need to maintain the model size.

Chapter 6

Constrained Data Collection

Previous chapter considers resource-constrained environment on a mobile device and maintaining the size of the nonparametric model. While this method solves the problem of constraining the model itself, it does not consider resource constraints in the form of acquisition of a large feature set. This problem can, in principle, be solved with standard feature selection. However, in case of more structured data, such as event sequences, the problem is more complex, as the features have different importance in the different part of the structure. This problem is very prominent in malware detection based on dynamic malware analysis, where we need to gather a large set of security-related event sequences and use them for anomaly detection.

This chapter contains a solution to the problem of gathering actual sensor data, while having a limited sensing budget. In particular, we consider the problem where data is gathered as a sequence of events, and we want to minimize the number of events that we track over time.

Parts of this chapter are published as concept in the position paper of Taubmann and Kolosnjaji [114] about an architecture for resource-aware malware analysis. We show more details about this concept including evaluation.

6.1 Related Work

In this section we describe the context of our paper by referring to previous works with similar topics. There is a long track of research publications describing attempts to use machine learning for malware detection and analysis. Many of them use behavioral traces as input data, where the features are obtained by tracing various events caused by malware execution such as system calls [126], registry accesses [50], and network traffic [116]. Paradigms used as baseline machine learning methodology range from topic modeling [62], kernel methods [9] to neural networks and deep learning [63, 134]. In particular, deep neural networks have been used in multiple recent papers in order to benefit from hierarchical feature extraction possibility. For example, Huang et al. [55]

use up to four hidden layers of feedforward neural network and focus on evaluating multi-task learning ideas. More recent papers [92, 134] exacerbate the need to use deep learning to obtain top performance in malware detection and classification for multiple architectures, where the systems using this approach largely outperform *shallow* machine learning systems and the existing antivirus programs.

Pascanu et al. [87] use recurrent networks for modeling system call sequences to construct a *language model* for malware. They test Long Short-Term Memory and Gated Recurrent Units and report very high classification performance. Further work [63, 96] also benefits from the use of recurrent networks for improving sequence modeling for malware detection. Although they obtain a model with high performance, in this work the authors do not consider the potential resource problems when obtaining execution traces in a production environment.

We also base our work on papers from machine learning literature that deals with the attention model for classification and feature acquisition. For instance, Contardo et al. [28] create a cost-sensitive recurrent neural network for adaptive feature acquisition. We use this approach and extend it for our anomaly detection purpose. Furthermore, we take the approach of Shen and Lee [109] into account when designing the classification variant of our recurrent network.

6.2 Methodology

In this section, we describe the methodology used in our paper, from data collection and preprocessing to the machine learning model. Figure 6.1 depicts the high-level view of our system's architecture. More specifically, it consists of a virtualized system for data collection and an ElasticSearch database, a feature extraction module, and a machine learning model trained for the detection of malicious executables.

6.2.1 Data Collection

Our method is based on the data collected from virtual machine introspection on Linux and Windows virtual machines. Because of different available infrastructure and malware sample sets, we design and implement data collection processes for Linux and Windows platforms separately. We use OpenNebula [78] to create a virtualization environment where we can host the Linux virtual machines. More precisely, we create a host virtual machine and afterwards a guest VM for every malware experiment. On the host Virtual Machine we run the *libvmtrace* tool [115] for collecting system calls, their arguments, and return values. This data comes from tracing the guest VMs that we create for each malware sample. The collected traces are written into an *ElasticSearch* database. While we also tested our approach on Linux malware, as we did not collect enough malware samples for conclusive results, we only further describe and show evaluation for Windows malware. To have a stealthy data collection mechanism and effective malware detection

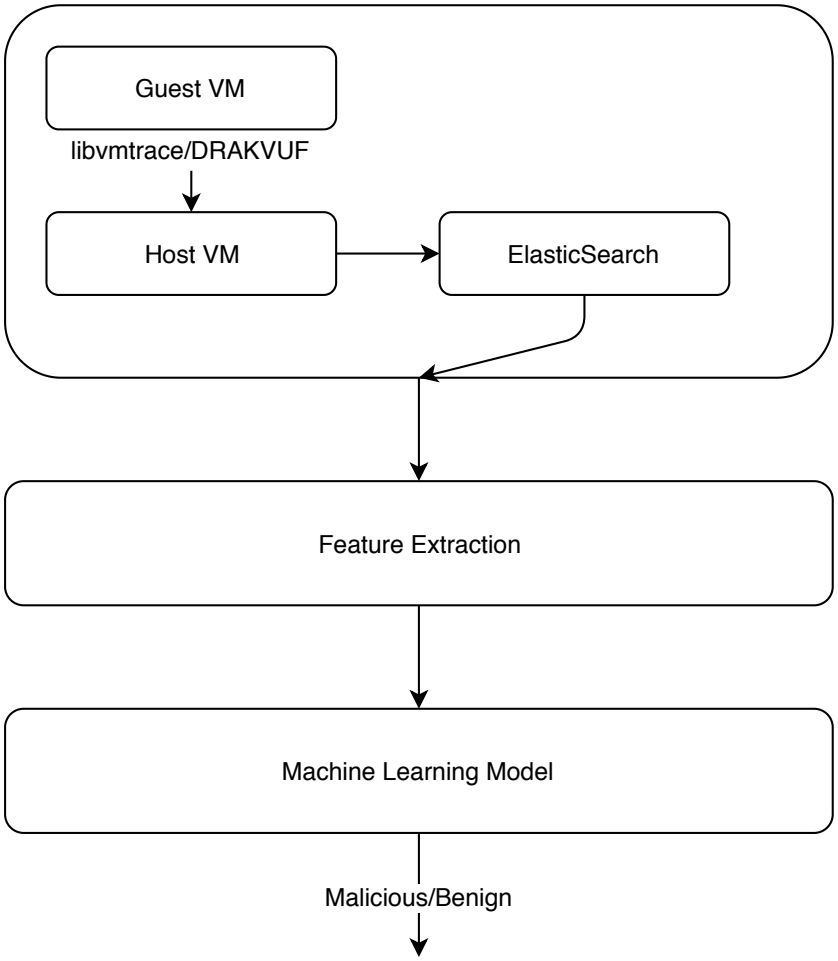


Figure 6.1: Scheme for our system architecture.

for Windows, we use Virtual Machine Introspection for tracing Windows malware as well. We use Xen on Linux as host to create *Windows 7* virtual machines. To trace the activity on the virtual machines we use the *DRAKVUF* malware analysis system [68], which we extend in order to have a document database, containing system calls for each malware sample. For collecting ground truth benign data we trace benign Windows processes running in the background. For Windows traces we use the following *DRAKVUF* event sets: *syscall*, *poolmon*, and *filetracer*. As syscall we consider a Windows system call, whose name always starts with *Nt*. The *poolmon* option enables tracing of calls to the `ExAllocatePoolWithTag` function, which is responsible for allocating objects in the Windows kernel heap. Finally, the filetracer collects traces whenever the *FILE OBJECT* is used, which gives overview of file accesses. Traces collected included side information: process name, trap name (for system calls), and parameters passed. This enables us to capture full information about the behavior of the executing application.

6.2.2 Preprocessing

We use one-hot encoding to transform the data from the system call logs into sequential feature matrices. Furthermore, system call arguments are included using concatenation. In order to equalize the length of system call traces, we use simple zero padding. This gives us tensors with three dimensions: dataset size, sequence length, and the encoding dimension. We forward data in this form to a machine learning model created in Tensorflow [6]. While it is common nowadays to use an embedding of high-dimensional data to have a trained representation with lower dimensionality, we do not use this approach. The reason for this is that the one-hot representation gives us the option to easily incorporate the resource-constraints into our model and track down which events contribute more to the malware detection performance.

6.2.3 Machine Learning Model

Our model for anomaly detection is based on a recurrent neural network, which is a standard method for sequence modeling [77]. The recurrent neural network is fed with input in form of a sequence of feature vectors (x_t). The input is forwarded to a recurrent unit, which produces a state value for each timestep (h_t). The state units comprise a memory of the network which outputs the latent representation of the training sequences. The value h_t in each timestep depends on the input in the current timestep as well as the recurrent unit state from the previous timestep:

$$h_t = f(x_t, h_{t-1}, \theta) \tag{6.1}$$

where θ incorporates the trainable parameters of our neural network. The parameters of this network are also trained using gradient backpropagation. However, since the backpropagation from the output to the input is done through multiple timesteps, the gradient also needs to be propagated through these timesteps. This method is called *backpropagation through time* and is a standard way to train recurrent neural networks.

Connecting multiple layers of the latent representation could in principle enable capturing more complex relationship between data samples and features. However, as we assume operation under resource constraints, we only use one such layer.

To detect an anomaly we need to collect information given by the states of our model (h_t) from all the timesteps t. Our unsupervised learning model is based on an autoencoder, where the dimensionality of input and output layer are the same, and the model learns to compress the traces into a set of parameters in the hidden layers.

We improve the control over the propagation of error using the concept of Long Short-Term Memory from Hochreiter and Schmidhuber [52]. Long Short-Term Memory (LSTM) is a type of recurrent neural network where units contain additional parameterized gates that help to train the propagation of input features through the input sequence. In case of the baseline neural network the gradient is propagated by multiplication operations, meaning that small values multiply through different timesteps,

causing the gradient to vanish through time. On the other hand, the additive update in LSTM does not contain this issue, which enables more effective learning for longer sequences.

The convenience of this method is that LSTM cells can easily be added to replace the common recurrent network units, without changing the overall neural network architecture. Although this requires increase of the number of parameters, which can in turn cause overfitting, we use regularization measures to counter this effect. Most important such measure is Dropout [111], where we only probabilistically use particular units of LSTM cells. In the evaluation section we show results using this kind of configuration.

In case of supervised learning we can produce an output in a form of probability (y) as a result of the state values during execution (h_t). For this we can use *average-pooling* or *max-pooling*, where we compute the average or maximum of values of h_t and convert it to an appropriate value range using the softmax function. This simple strategy of using recurrent neural network with *max-pooling* was used successfully in the paper of Pascanu et al. [87] for a malware classification task. Although, according to this work, this model offers high performance, in our work we are more concerned with the overhead of tracing all the events. Therefore, we extend this model with an attention mechanism in order to limit the overhead and offer a cost-sensitive prediction. In developing this model we build upon strategies mentioned in the position paper of Taubmann et al. [114], which is in turn based on the work of Contardo et al. [28].

6.2.4 Attention Layer

The attention layer contains units that enable control of importance of features in sequence modeling. This has already been used in processing videos [121] and image caption generation [132]. Standard feature selection methods, such as *Lasso* [47], allow for determination of features to use in the modeling along with the optimization for accuracy w.r.t the ground truth data. However, these methods do not allow for sequence-aware feature selection and control of event tracing.

In our use case, the reason for this additional layer is that we want to programatically select the statistically significant events to be traced after neural network training. This would enable us to execute large-scale analysis with acceptable resource demands. We hypothesize that there is a reduced set of events that carry most of the discriminative information for determining the anomaly score of a program under test. Furthermore, we include a functionality where the trained model enables changing of the set of traced events for each timestep.

The attention layer consists of attention vectors (α_t), containing event weights depending on the time step. Figure 6.2 displays the recurrent neural network scheme for supervised learning including the attention vectors. The scheme shows a layered structure of the recurrent neural network, where apart from the standard set of input, hidden, and output layers, we can find an attention layer with vectors α_t. This layer is used for determining the weights of events, thereby enabling cost-sensitive tracing.

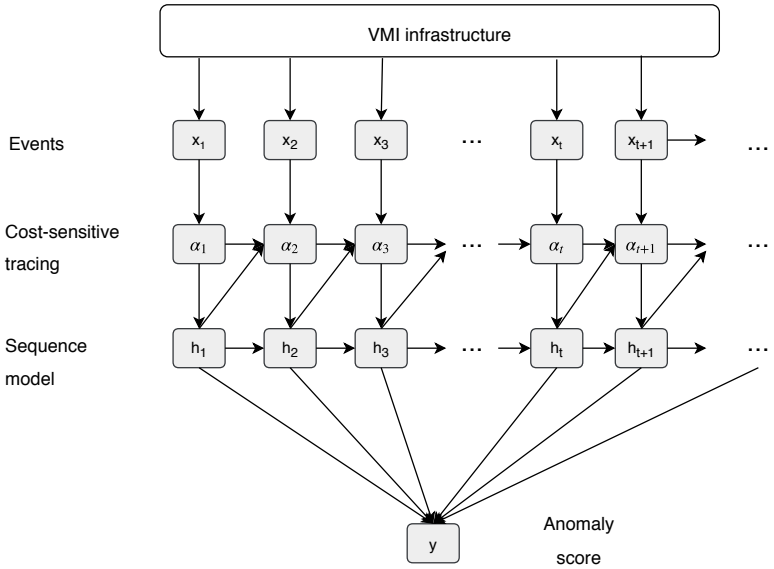


Figure 6.2: Neural network architecture with the attention model.

Similarly, in case of unsupervised learning we have an architecture with the attention layer. However, the output layer (o_t) should contain the reproduction of the inputs (x_t). This is displayed on Figure 6.3. In this case, the training process minimizes the squared distance between the inputs and the outputs. We also introduce Gaussian noise to the inputs in the training set in order to reduce overfitting.

In the attention layer, the updates are done sequentially. In particular, the current attention vector depends on the attention vector (α_{t-1}) and the state vector (h_{t-1}) of the neural network from the previous time step as well as from the current input (x_t):

$$\alpha_t = s(h_{t-1}, \alpha_{t-1}, x_t) \tag{6.2}$$

The attention update function s defines how the attention vector is reassigned. We use the following update equation, similar to standard neural network unit updates:

$$\alpha_t = \sigma(Wh_{t-1} + Vx_t + M\alpha_{t-1} + b) \tag{6.3}$$

where we train the weight matrices W,V and M, as well as the bias vector b. The sigmoid function (σ) is used to normalize the values in the attention vector. This means that the weights of events are scaled between zero and one.

The update equation of α_t shows that the weight of different events in a certain timestep potentially depends on multiple previously known values. First, it depends on the state of the network from the previous timestep, as the state vector propagates information throughout the time of execution. Second, it depends on the input in the current timestep, as the change in the current input affects the importance of current measurements. Last, it depends on the weights of events in the previous timestep, as we want to provide information about how important an event was in the previous time.

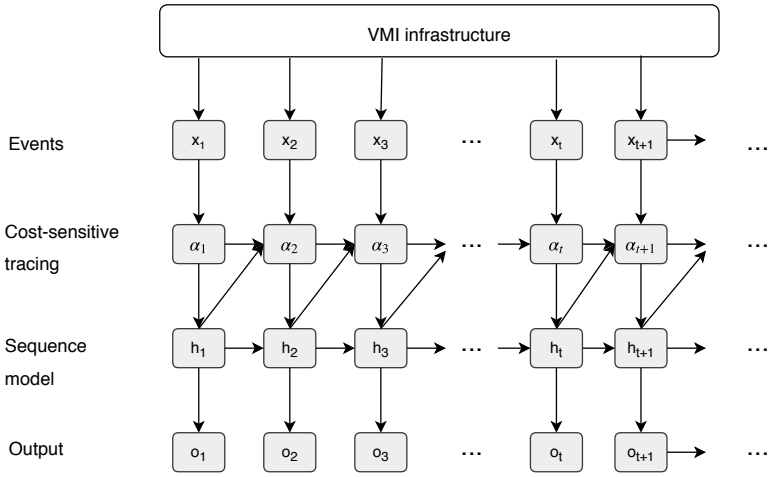


Figure 6.3: Autoencoder architecture with the attention model.

Next, we want to force sparsity in this updated vector α_t, taking special attention at the values already close to zero. In order to bring the low values of α_t to zero, we use the following update:

$$\alpha_t = ReLU(\alpha_t - \epsilon) \tag{6.4}$$

The function *ReLU* is called a *rectifier* linear unit and is linear for the inputs higher than zero, while it results with zero for input of zero and negative input values. The *ReLU* unit is very often used as an activation function in neural networks, especially in case of using a large number of layers (*deep learning*). The value ϵ is a preassigned constant. Essentially, this function assigns zero to all values lower than ϵ.

The resulting attention vector is used to filter the input in each timestep. The filtered vector is the value g_t:

$$g_t = \alpha_t x_t \tag{6.5}$$

Furthermore, we can keep the standard RNN update for our sequence modeling:

$$h_t = f(g_t, h_{t-1}, \theta) \tag{6.6}$$

The function f employed in the last equation depends on the type of neural network we are using.

In case of supervised learning, we use *average-pooling* and softmax function in order to get the labels for our sequences:

$$o_t = \sigma(f_{average}(h_{1..T} + b)) \tag{6.7}$$

Otherwise, in case of unsupervised learning, we gather the outputs from the recurrent layer:

$$o_t = g(h_t) \tag{6.8}$$

The attention vectors define the weights for the inputs. In order to save on the tracing budget, it is in our best interest to reduce the weight vector components to zero for as many events as possible. We can, in principle, achieve this by adding a sparsity component into our loss function [28]:

$$loss = \sum_{t=1..T} (y_t - \hat{y}_t)^2 + \lambda \sum_{i=1..N} (\sum_{t=1..T} a_{nt} c) \tag{6.9}$$

Our loss function consists of two parts, indicating the aim to balance performance and resource budget. In the first part, it counts the difference between the produced output y_t and desired output $\hat{y}_t$. In case of supervised learning, the produced output is the value between zero and one, representing the probability of the program under test being malicious or benign. The desired output is the binary label for this program used for training. On the other hand, in case of unsupervised learning, the produced output is the reproduced input given to the autoencoder, while the desired output is the entire input trace.

The second part contains the expression for minimizing the necessary event budget, represented with α_t. We train our model to minimize the quadratic loss described in the previous paragraph, as well as to minimize the overhead of acquiring input features. The parameter λ, which we call a *sparsity constant*, is used to control the tradeoff in optimization, while the value c enables tuning the weights in budget constraints between different events. The sparsity is also controlled on the feature (event) level and not just for particular sequence timesteps. This enables us to select events that are important throughout the sequence model.

Since our set of malicious executables is much larger than the benign sample set, we use random oversampling during training to counter the bias in the results.

The optimization of this loss function can be done using a gradient-based method, such as simple gradient descent, or Adam [58]. Since the joint optimization is prone to local optima, we also consider an alternative, to alternately optimize the two parts of the loss function by a small number of gradient-descent steps. We actually achieve better results using this method, while repurposing the sparsity constant to the learning rate for the second part of the loss function.

6.3 Evaluation

In this section we show the performance of our methodology. Although we tested the functionality of our framework on Linux and Windows data, due to a small number of Linux malware samples we only show the quantitative evaluation for Windows. Our main goal is to evaluate the performance of our malware detection system under various amount of budget constraint. We execute the evaluation in both unsupervised and supervised anomaly detection scenarios. The evaluation is conducted using a 3-fold crossvalidation, while solving the unbalanced dataset problem using oversampling.

As our malware sample set is not as large as it could be in practice, our results are preliminary. However, they show promising performance for our approach.

6.3.1 Experiment Setup

We use a collection of 1000 malware samples from Virusshare [98] to gather malicious traces. In addition, we record typical benign activity by recording the behavior of background processes in a Windows Virtual Machine. We repeat this activity tracing multiple times to generate multiple benign samples for training purposes. We implement neural networks using Tensorflow 1.11 and execute the code on the NVIDIA TitanX GPU. The experiments are conducted using a 3-fold crossvalidation. We measure average performance in our tests by computing the following values for each experiment instance: accuracy, F1-score, precision and recall.

6.3.2 Sensitivity to Sparsity Constant

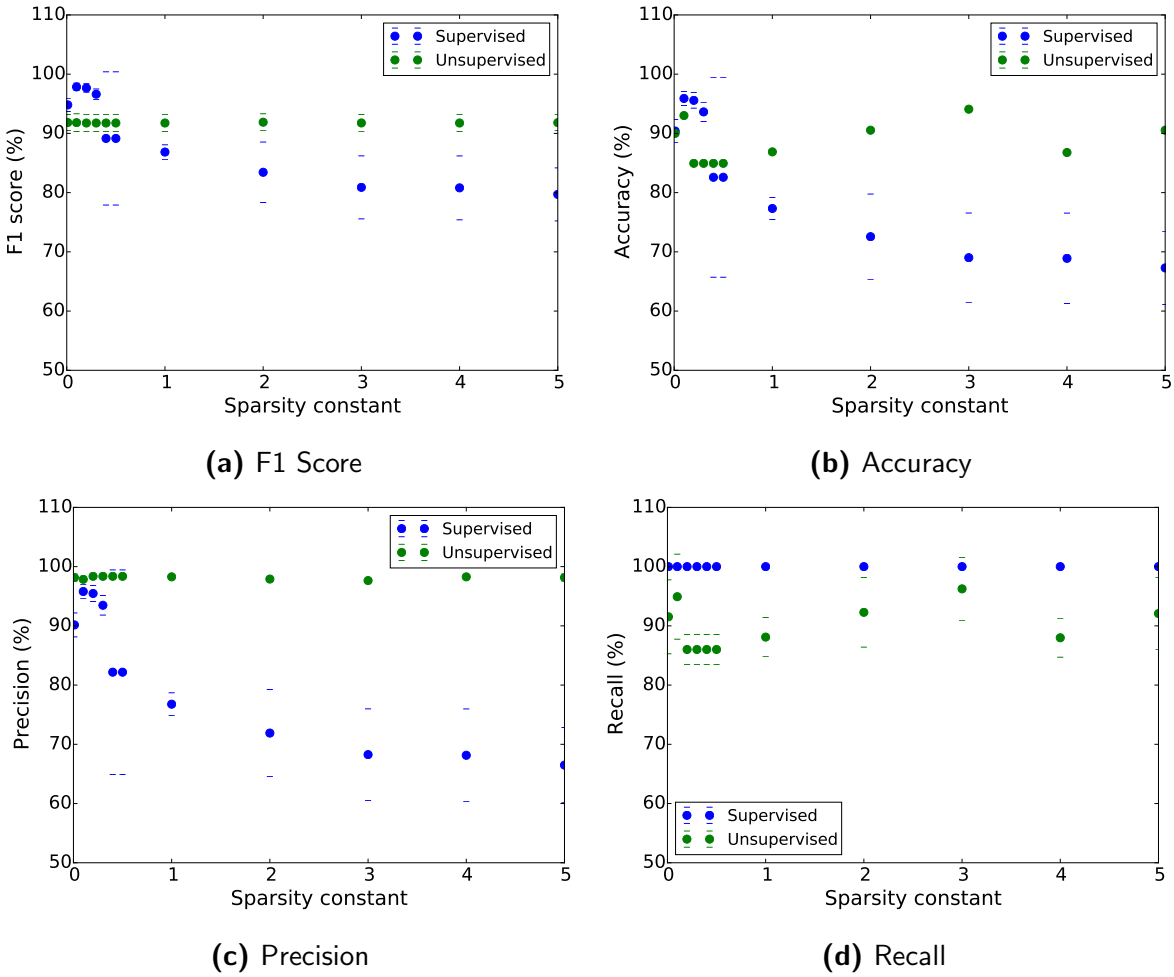
In the next experiment we train the malware detection system with a varying value of the sparsity constant (λ), which controls the weight of the part in the loss function for sparsity enforcement. The higher value of this constant means that we consider sparsity more important, which results with a model that uses a lower number of events to obtain the results. The results are displayed on Figure 6.4a (F1 score), Figure 6.4b (accuracy), Figure 6.4c (precision) and Figure 6.4d (recall).

These figures show that multiple metrics are stable under different amount of weight given to the sparsity component in the optimization. Especially for the autoencoder, we see very stable results, even with a very high accent on sparsity in the optimization. The most representative single performance measure is the F1 score. For the supervised recurrent network there is a significant reduction in this value. Although we still maintain the score of over 70% for high values of sparsity constant, for higher performance it is still important to tune this value in order to achieve the best results.

6.3.3 Performance Under Constraint

In the next figures we show the actual performance of the system under the budget constraints, using multiple standard metrics. Due to a different sparsity constraint, we have a model with various budget usage in terms of the percentage of events with non-zero weights (α_t). The results are displayed on Figure 6.5a (F1 score), Figure 6.5b (accuracy), Figure 6.5c (precision) and Figure 6.5d (recall).

The figures show that the results have mostly stable values, despite the fact that we only use a small portion of the budget. We do not see a smooth performance reduction trend, which could be expected for small changes in the budget. Similar to the previous test, the reduction of precision and subsequently F1 score for supervised LSTM in particular is high once we reach a high enough value of sparsity constant. This means



(a) F1 Score **(b)** Accuracy

(c) Precision **(d)** Recall

Figure 6.4: Plot showing the change in results with the adjustment of tradeoff between performance and model sparsity

that, according to these results, we have reached the lower limit of budget that gives us an acceptable result. In the following, we show the relation of sparsity constant and budget usage.

6.3.4 Budget Usage Optimization

The previous plots give us a hint that the change in the budget usage is not smooth and that it would be interesting to see how the sparsity constant influences the percentage of events used for training our model. Figure 6.6 displays the shape of this trend. More precisely, it reveals that the budget usage can change very fast with the sparsity constant and upon finding a minimal budget it remains constant. This happens around the value of $\lambda = 2.0$. The consequence of this is that the sparsity constant needs careful tuning in

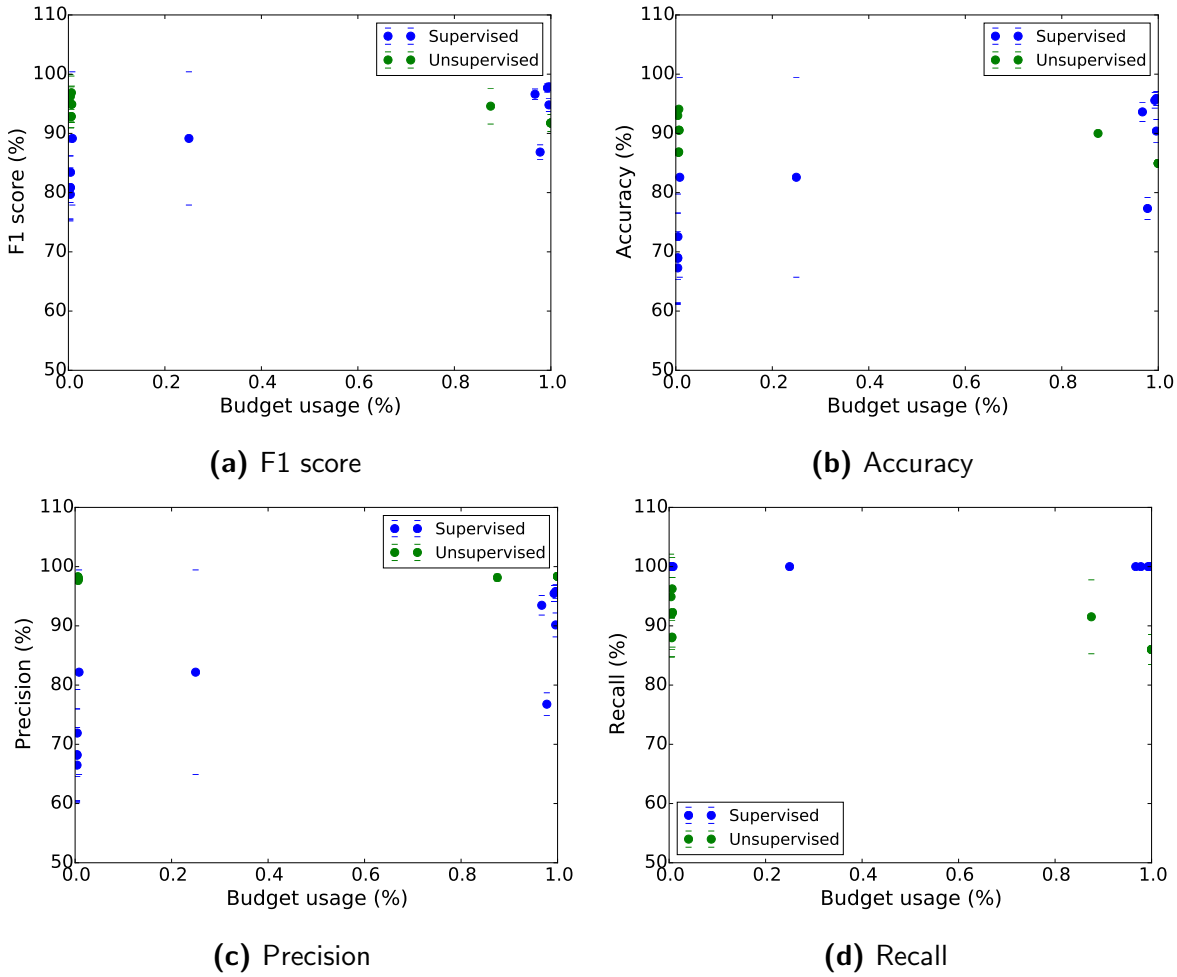


Figure 6.5: Plot showing the performance of our malware detection system for different amount of resulting budget usage, for both unsupervised and supervised learning

order to achieve this minimal budget, where we find all events that can be ignored with minimal effect on the malware detection performance.

6.3.5 Windows API Function Importance

In this subsection we show the importance of different functions in Windows API (mostly system calls) that we compute using our framework. After training our detection system, we can measure the value α_t, which contains the attention vector, showing the importance for particular API functions in timestep t. One-hot encoding enables us to directly get a scalar weight value per function.

In Table 6.1 we display the API calls that have the average value of α_t over 50%, for tests with $\lambda = 2.0$. These are 13 most important API functions out of 201 functions that

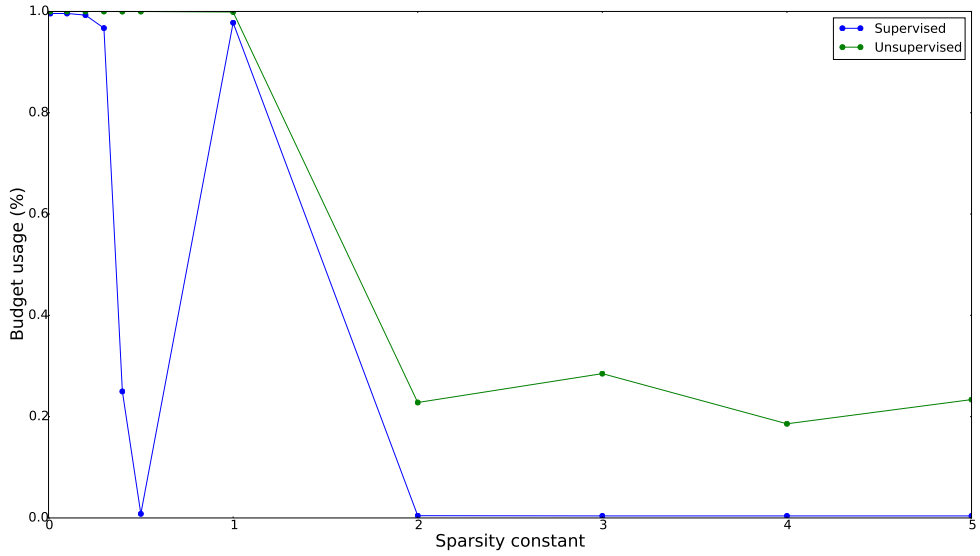


Figure 6.6: Budget usage vs. sparsity constraint

we trace. This table shows that the most important API functions are spread around the most common activities which happen in malware, such as manipulating the registry, opening a network connection, opening new threads. However, these activities are present in benign programs as well, therefore monitoring of arguments is also important for interpretation.

Table 6.2 shows the Windows API Functions that are most often set to zero in our tests. This means that these are the functions that are least important to trace, according to our evaluation. This table also shows a list of API functions of variable functionality. It is noticeable that out of ten least used functions, three are queries for information from the operating system (NtQuery). This could indicate that this kind of queries are not important for malware detection.

In the last table we show how the set of most important functions changes with the amount of sparsity that we enforce. Table 6.3 contains 10 API functions with highest average weights for different values of sparsity constant (λ). This table reveals that, for small sparsity constant, the list of most important system calls remains stable. However, as the budget gets lower, this list starts changing, especially as λ approaches the value 1.0. Afterwards, the list is different for each following increase of λ. However there are some functions that have a more stable place in the list, such as RegDeleteKeyA and partially NtSetValueKey.

Table 6.1: List of most API calls with highest average weight with their descriptions

Windows API Function	Mean attention value
LdrGetProcedureAddress	0.9004
InternetOpenW	0.6028
NtOpenFile	0.6016
NtAlpcDeleteSectionView	0.6016
RegCreateKeyExW	0.6016
URLDownloadToFileW	0.6016
NtOpenThreadToken	0.6016
RegDeleteKeyW	0.6016
InternetReadFile	0.6016
RegDeleteKeyA	0.6016
NtContinue	0.6016
NtSetInformationProcess	0.6016
DeleteFileA	0.6016

6.4 Discussion

In this section we briefly discuss the impact of our methodology and research results.

6.4.1 Generality of Our Approach

In this paper, we use neural networks based on LSTM, for both supervised and unsupervised learning, as they are the state-of-the-art in machine learning for sequential data. However, our approach could also be adapted for other methods for sequence learning such as Hidden Markov Models or methods using String Kernels. All these methods are based on numerical encoding of events and use the event traces as a source of features. By imposing sparsity on event traces, we do not make high changes on the particular baseline machine learning model (in our case recurrent network), only on the feature selection.

6.4.2 Possible Applications

We test our approach in a particular environment of malware detection in a cloud computing system. However, we argue that the low-budget sequence learning is a problem that appears in many other settings, for instance, the *Internet of Things*. Furthermore, the approach can be used in various different crowdsourcing scenarios where budget for data is limited.

Table 6.2: List of least used Windows API functions

Windows API Function	Mean attention value
NtCancelWaitCompletionPacket	0.0040
NtQueryInformationToken	0.0040
NtAlpcConnectPortEx	0.0040
NtAlpcAcceptConnectPort	0.0040
NtUnmapViewOfSectionEx	0.0040
NtProtectVirtualMemory	0.0040
NtUnmapViewOfSection	0.0040
NtQuerySecurityAttributesToken	0.0040
StartServiceW	0.0040
NtQueryInformationProcess	0.0040

6.5 Limitation and Future Work

This work shows the potential of using budget-efficient neural network architectures to address the constraint of feature acquisition in learning detectors of malicious behavior. However, there are limitations to this approach. We plan to address these limitations in our future work.

6.5.1 Dataset Improvements

Our dataset contains a diverse set of malicious traces. However we expect that more reliable results would still be obtained with a larger set of malicious and, especially, benign traces. Currently, the results are biased toward the data that we have collected, which is most obvious in our benign dataset. This occurs because the experiments with benign data are artificially designed. Measurements with a real-life production system would be a very interesting extension for this paper.

Furthermore, due to hardware limitation, sequences longer than 250 system calls could not be processed. Longer sequences were cropped and this might have a negative effect on the results. On the other hand, shorter sequences were padded with zeros. Other padding techniques could be explored.

6.5.2 Feature Acquisition Cost

We present a cost-aware recurrent neural network architecture. In our case the unit feature-specific cost (unit cost for a single event) is uniform across all features. More complex cost functions could be explored. For instance, monitoring network traffic could

Table 6.3: List of most important Windows API functions for varying λ

λ=0.0001	λ=0.01	λ=0.1	λ=1.0
CreateServiceW	CreateServiceW	NtTestAlert	NtQueryValueKey
NtCreateIoCompletion	NtCreateIoCompletion	NtWaitForMultipleObjects	NtSetInformationProcess
NtAlpcConnectPortEx	NtAlpcConnectPortEx	RegEnumKeyExA	NtProtectVirtualMemory
NtQueryInformationToken	NtQueryInformationToken	RegSetValueExW	NtCreateThreadEx
NtCancelWaitCompletionPacket	NtCancelWaitCompletionPacket	DeleteFileW	RegCreateKeyExW
NtReleaseSemaphore	NtReleaseSemaphore	NtOpenMutant	NtQueryVolumeInformationFile
NtSetValueKey	NtSetValueKey	CreateServiceW	NtCreateMutant
NtCreateSemaphore	NtCreateSemaphore	NtQuerySystemInformation	SetWindowsHookExA
NtOpenSemaphore	NtOpenSemaphore	NtCreateMutant	RegEnumKeyExW
NtCreateMutant	NtCreateMutant	NtQueryInformationThread	NtSetValueKey
λ=2.0	λ=3.0	λ=4.0	λ=5.0
LdrGetProcedureAddress	OpenSCManagerW	NtCancelTimer2	IsDebuggerPresent
InternetOpenW	NtResumeThread	StartServiceW	CreateDirectoryW
NtOpenFile	NtSetValueKey	ExitThread	NtOpenFile
NtAlpcDeleteSectionView	NtCreateSemaphore	NtSetValueKey	NtAlpcConnectPortEx
RegCreateKeyExW	NtQuerySecurityObject	RegDeleteValueA	FindWindowExA
URLDownloadToFileW	RegDeleteKeyA	NtOpenThreadTokenEx	MoveFileWithProgressW
NtOpenThreadToken	LdrGetDllHandle	InternetReadFile	OpenSCManagerA
RegDeleteKeyW	WriteConsoleA	InternetOpenW	RegEnumValueW
InternetReadFile	CreateThread	NtCreateNamedPipeFile	RegCloseKey
RegDeleteKeyA	NtTerminateThread	RegDeleteKeyA	NtAlpcCreateSecurityContext

cost less than monitoring other system calls and there could be a variety in tracing cost within system calls themselves. To determine a more complex cost function, we need to measure repeatedly and characterize the cost for the traced events. This would enable us to find a relative cost weight vector. Another interesting topic would be to include online learning concepts in our methodology. This would enable us to update the model while gathering data on a longer time scale.

6.5.3 Obfuscation

Although dynamic analysis partly solves the problem of code obfuscation, there exists an issue of behavioral obfuscation as well. Malware authors can use mimicry, i.e. insert sequences of system calls characteristic for benign programs into malware behavior in order to circumvent the detector. Recurrent neural networks have been proven susceptible to such attacks by Papernot et al. [86] who have shown a procedure for creating adversarial examples for recurrent neural networks by introducing small changes that cause misclassification. If the attacker is able to make an unlimited number of queries to the system, this can enable a gradient-based attack that would gradually change the output of the classification system until the desired result is retrieved. Even in case of black-box attacks, where the attacker does not have any knowledge about the classification methods used by the system, the attack transferability could enable successful evasion [85].

6.6 Summary

In this paper, we present a recurrent neural network model for anomaly detection that is cost-aware and can perform well compared to a baseline recurrent neural network with infinite budget. When the budget is too small, the performance of the network degrades according to the metrics we used, as fewer features are taken into consideration. Nonetheless, in our tests with datasets from Virtual Machine Introspection consisting of examples of malicious and benign behavior, the performance with our resource-aware architecture can be almost as good as the recurrent neural network with no resource limitations. Our work motivates the budget-efficient approach, since tracing all possible system events is too cumbersome for virtual machine introspection.

The use of our approach is not limited to the use case of virtual machine tracing, but can be used in other scenarios where resource availability is limited. For example, in wireless sensor networks or various Internet of Things scenarios, the tradeoff between performance and sensor usage is an important goal that needs to be optimized.

Chapter 7

Communication-Efficient Learning

In the previous chapters we considered the resource constraints in data collection and modeling. Especially, the Chapter 6 considers data acquisition as a possible bottleneck for machine learning model functionality. We also look further in this direction and consider the problem of gathering labels from external entities. The goal of this chapter is to describe our method of achieving optimal classification accuracy for minimal expenses in terms of needed labeling budget.

7.1 Related Work

There are multiple papers that deal with estimating the expertise of labelers when using *wisdom of crowds*. For example, Welinder et al. [127] determine groups of annotators with similar expertise, and find particular phenomena such as "schools of thought" and specialist experts in certain types of image labeling tasks. Tian and Zhu [118] investigate the annotator behavior in executing more complex labeling tasks, such as answering if an image is beautiful or if it contains a car. Zhang et al. [138] propose a method to filter out novice labelers. However, their method is specific to Gaussian Mixture Models. Rodrigues et al. [100] determine a similar procedure specific to Gaussian Processes. Bi et al. [16] argue that the factors that determine the annotators' labeling performance, apart from the expertise, can also be the worker's dedication to the task, his/her default labeling judgment and sample difficulty. However, they do not go further in determining the optimal subset of annotators. On the other hand, Li and Liu [70] propose a combinatorial procedure to find this optimal subset. However, such combinatorial procedures are very resource-intensive, especially for a large initial set of annotators.

On the other hand, there are are also multiple more theoretical frameworks. For example, Wang et al. [124] propose a selection of sensors based on Directed Acyclic Graphs (DAGs) in resource-constrained scenarios. However, they do not deal with annotating data, so their ideas do not completely fit to our problem. We investigate the possibility of designing a procedure for selection of a small number of highly competent annotators

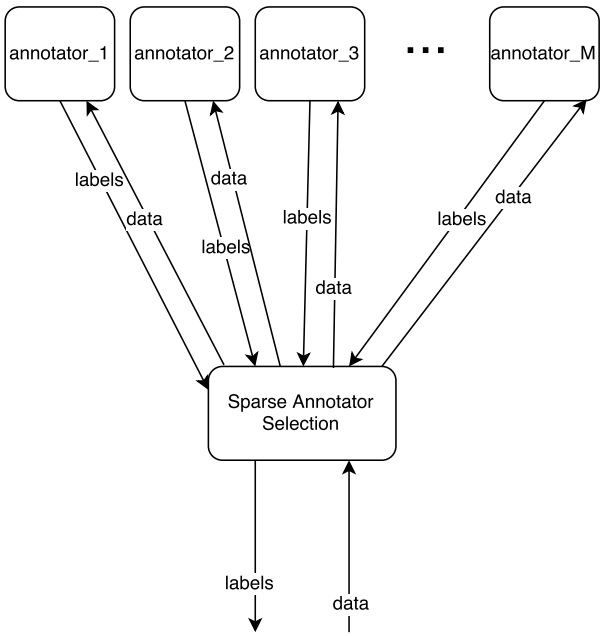


Figure 7.1: Label aggregation scheme

whose combined expertise enables acceptable performance in anomaly detection. This way we save on communication cost when retrieving label information, while minimizing the impact on performance due to a limited communication budget.

7.2 Methodology

Our methodology is based on the scenario drawn on the Figure 7.1. This scheme shows a common situation when a server acquires labels from multiple annotators.

In training time, server forwards the training inputs to all the annotators in order to retrieve labels. Using the data labeled by a large set of annotators, the server trains a machine learning model that jointly learns the expertise of all the annotators while minimizing the quadratic loss with respect to the ground truth. We assume that during the training the server has ground truth available and uses it to adapt the weights of the annotators. Moreover, we assume unlimited communication budget during the training time.

We design a method that would enable us to manage the tradeoff between model accuracy and communication effort that would be needed in the test time. Furthermore, we impose a requirement that our methodology needs to be simple and compatible with a wide range of machine learning methods. Based on these requirements, our optimization model has a loss function consisting of three parts:

1. minimizing the loss with respect to the ground truth labels on the training set

2. minimizing the loss with respect to the annotations

3. making the annotator weight vector sparse (selecting a small subset of annotators)

7.2.1 Simple Model

In a simpler instance of our model we consider annotator expertise in a form of weights. We introduce a vector of weights that determines the importance that we assign to particular annotators. Furthermore, we design an optimization approach that includes sparsification of this weight vector, i.e. we optimize a model that includes annotations from a reduced number of annotators.

The loss function for cross-entropy loss on ground truth, used in all our experiments, looks like the following:

$$L(w,v) = -\frac{1}{N}\sum_{n=1}^{N}(\hat{y}_n \log \sum_{i=1}^{M} v_i y_{ni} + (1-\hat{y}_n)\log(1-\sum_{i=1}^{M} v_i y_{ni}))+ \\ +\psi \sum_{n=1}^{N}\sum_{i=1}^{M}(y_{ni}-z_{ni})^2 + \lambda \mid v \mid \tag{7.1}$$

subject to:

$$\sum_{i=1}^{M} v_i = 1 \tag{7.2}$$

where $\hat{y}_n$ are ground truth labels for all training samples, y_{ni} are labels retrieved by querying current annotator models, z_{ni} are training labels from annotators, v_i is a normalized vector of weights for all the annotators. The final loss function consists of a crossentropy calculation for the empirical loss with respect to the ground truth, the quadratic loss with respect to the annotations weighted by a parameter ψ and the L_1-norm of the parameter vector v weighted by another parameter λ.

We want to use L_1 regularization to minimize the annotator weight vector, as this minimization enables fast setting of unimportant annotator weight vectors to zero. This is similar to the *Lasso* feature selection, except that we select annotators instead of features.

The gradient is defined with the following equations:

$$\frac{\partial L}{\partial w_j} = -\frac{1}{N}\sum_{n=1}^{N}(y_n \frac{1}{\sum_{i=1}^{M} v_i \hat{y}_{ni}} v_j \frac{\partial y_n}{\partial w_j} - (1-\hat{y_n})\frac{1}{1-\sum_{i=1}^{M}} v_j \frac{\partial y_n}{\partial w_j})+ \\ \psi \sum_{n=1}^{N}\sum_{i=1}^{M} 2(y_{ni}-z_{ni})\frac{\partial y_n}{\partial w_j}) \tag{7.3}$$

$$\frac{\partial L}{\partial v_i} = -\frac{1}{N}\sum_{n=1}^{N}(y_n \frac{1}{\sum_{i=1}^{M} v_i y_{ni}} + (1-\hat{y_n})\frac{1}{1-\sum_{i=1}^{M} v_i y_{ni}}(-y_n)) \tag{7.4}$$

The expression for the gradient $\frac{\partial y_n}{\partial w_j}$ depends on the concrete method used to model the annotators.

The gradient-based procedure enables us to gradually optimize the models of the annotators' decision process, as well as to exclude the annotators that do not have an important positive influence on the overall model accuracy. During this procedure the server needs to communicate with all the annotators, which causes the starting overhead. However, afterwards in test time there is only a need to communicate with the annotators selected by the model.

For logistic regression we directly define the cost for the errors with respect to ground truth as crossentropy. However, with minimal changes our framework can also be usable with other machine learning methods. For example, we tested our methodology with neural networks as well. These methods only need to be compatible to our gradient-based optimization procedure.

In order to use gradient descent as a convex optimization method with the L_1 norm, we use iterative soft-thresholding [21], a type of generalized gradient descent optimization when updating the value of v. The update that we derived is the following:

$$v = S_\lambda(v - t_s \frac{\partial L}{\partial v}, \lambda) \tag{7.5}$$

where

$$S_\lambda(k,\lambda) = \begin{cases} k-\lambda & \text{if } k > \lambda \\ k+\lambda & \text{if } k < -\lambda \\ 0 & \text{otherwise} \end{cases} \tag{7.6}$$

Furthermore, we normalize the values in the vector v in order to retrieve the proper values of annotator weights:

$$v = \frac{v}{\sum_{i=1}^{M} v_i} \tag{7.7}$$

In case of multiclass classification, we would define the loss function in a slightly different way, to accommodate with the fact that we need a label vector instead of scalar value:

$$L(w,v) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K}(\hat{y}_{nk}\log\sum_{i=1}^{M} v_i y_{nik}) + \psi\sum_{n=1}^{N}\sum_{i=1}^{M}(y_{ni}-z_{ni})^2 + \lambda \mid v \mid, \tag{7.8}$$

subject to

$$\sum_{i=1}^{M} v_i = 1 \tag{7.9}$$

This time, $\hat{y}$ is a matrix and y is a three-dimensional tensor. The optimization is done in a similar way, using projected gradient descent. We minimize the cross-entropy w.r.t the ground truth, as well as the squared difference of ground truth and annotator decision, while keeping the annotator vector sparse.

7.2.2 Full Expertise Model

The simple expertise model enables us to select the most reliable annotators for our data. However, assuming that the annotators do not have the same amount of expertise for various parts of the feature space, we could benefit from a more fine-grained annotator selection. Therefore we also design a full expertise model, where annotators are selected for each classification query. In this case, we have a different loss function:

$$L(w,v) = -\frac{1}{N}\sum_{n=1}^{N}(\hat{y}_n \log \sum_{i=1}^{M} v_{ni}y_{ni} + (1-\hat{y}_n)\log(1-\sum_{i=1}^{M} v_{ni}y_{ni}))+ \\ \psi \sum_{n=1}^{N}\sum_{i=1}^{M}(y_{ni}-z_{ni})^2 + \sum_{n=1}^{N}\lambda \mid v_n \mid \tag{7.10}$$

We can also optimize this loss function with a similar gradient descent procedure. Here we have a matrix instead of a vector v, where each row in the matrix needs to sum to one and we need to make them sparse. During the training, for each training point we maintain a multinomial logistic regression model, based on the annotator weights for different data points. This model will then be used to determine annotator weights in the validation and test phase. Furthermore, we can benefit from the simple model as an initialization procedure for our full expertise parameters. In the next section we compare the results from the simple and full expertise models.

Both the simple and the full expertise model can also be constructed using learning methods other than logistic regression, for example with neural networks or Gaussian processes. The only condition is that it needs to be fit to the described joint optimization scheme.

7.3 Evaluation

In this section we describe the evaluation results obtained with our methodology. We show results for a dataset from *Amazon Mechanical Turk*, as well as some first results for malware detection.

7.3.1 Amazon Mechanical Turk

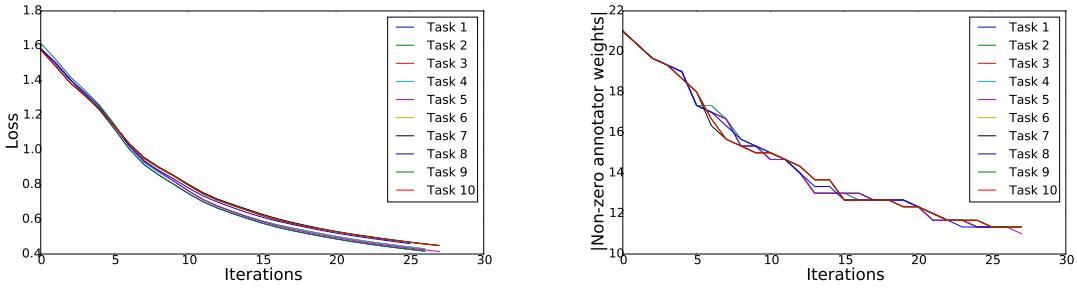
We obtained a real-life labeler set from the authors of a related paper about learning to predict from crowdsourced data by Bi et al. [16]. This dataset consists of results from an

experiment with 21 workers at the *Amazon Mechanical Turk*. These workers executed 10 tasks, all of the tasks being binary classification of images. Since each worker has different expertise, the challenge is to determine the subset of highly competent workers and combine their abilities for a superior classification system.

7.3.1.1 Optimization process

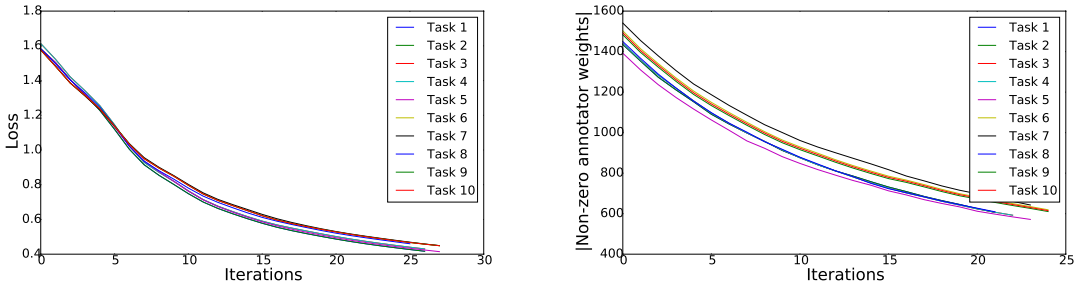
At first, we look at how the value of the loss function changes while the optimization is running. We plot the loss against the number of iterations, which shows the steep decline of the loss value. This shows that using our gradient-based procedure we can get fast to a very accurate classification system. The graph is shown for the simple and full expertise model on Figure 7.2 and Figure 7.3.

Next, we execute a similar test with the parameter v. We look at how the number of nonzero elements of v changes while optimization is running. In other words, we examine how the optimizer selects the competent annotators.



(a) Loss minimization **(b)** Reduction of annotator subset

Figure 7.2: Training process for the simple expertise model

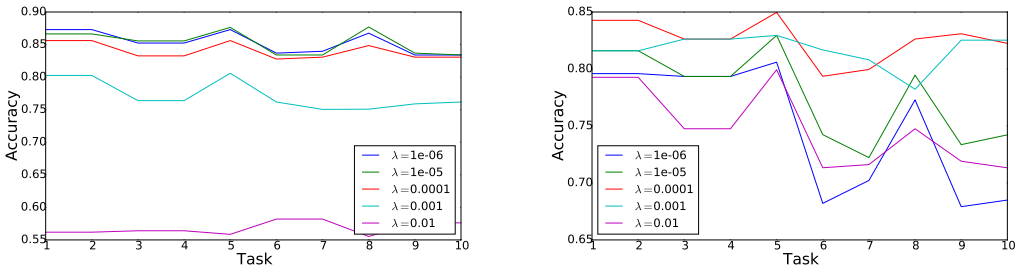


(a) Loss minimization **(b)** Reduction of annotator subset

Figure 7.3: Training process for the full expertise model

7.3.1.2 Performance breakdown

In the next figure we examine the breakdown of classification accuracy per labeling task, for different value of λ. We can notice on Figure 7.4 that the performance of our classifier does not significantly vary for different tasks in case of the simple model. The results are also similar for varying λ, except for $\lambda = 0.01$, where the accuracy falls fast. This might be due to a too aggressive sparsity enforcement. On the other hand, the full expertise model does not contain this fall in accuracy. However, in the full expertise model, there is a higher variation in accuracy per task, as the fine-grained optimization affects the accuracy results.



(a) Simple expertise model **(b)** Full expertise model

Figure 7.4: Performance per task for different value of λ

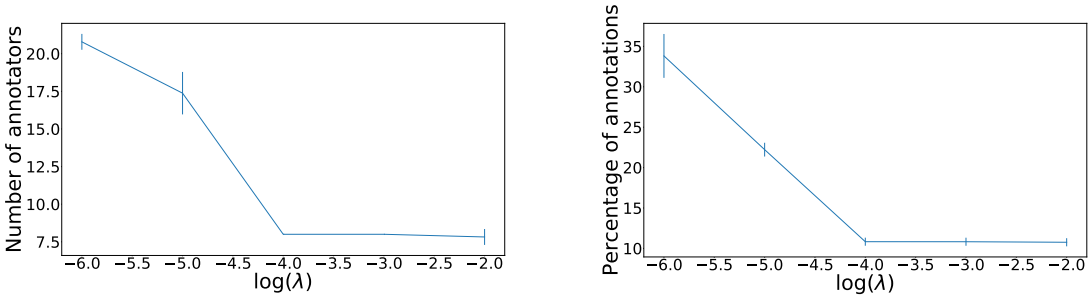
7.3.1.3 Comparison with baseline uniform weights

Furthermore, we compare the performance of our scheme with annotator selection to a baseline system where each annotator has an equal weight. This weight is kept constant, which means only model per annotator (weights w) is optimized.

Figure 7.5 shows the change in the final number of annotators for different value of sparsity constant λ. It shows that for the value of 0.0001 we already have a fast fall of the number of annotators. Nevertheless, the performance of our model does not significantly fall. This can be seen on Figure 7.6. On the other hand, we observe a similar fall of needed annotations for the increase of λ in case of the full expertise model as well. Here we also do not observe any further benefit for increasing λ after 0.0001 in terms of labeling budget saving. Furthermore, for this value of λ we get a slightly better value of AUC in comparison to the baseline value and even to the fully optimized model with using almost all the annotations.

7.3.2 VirusTotal Annotations

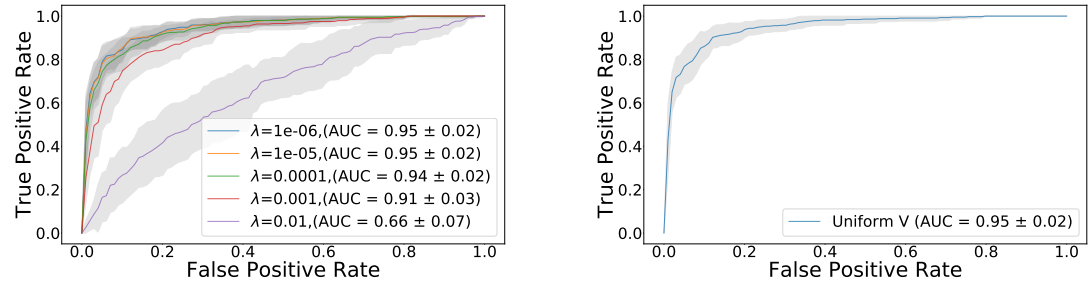
In addition to the results with Amazon Mechanical Turk, we execute an experiment with the malware detection dataset, containing annotations from VirusTotal [3]. We collect



(a) Number of annotators for various $\log(\lambda)$ when using the simple model

(b) Number of annotators for various $\log(\lambda)$ for the full expertise model

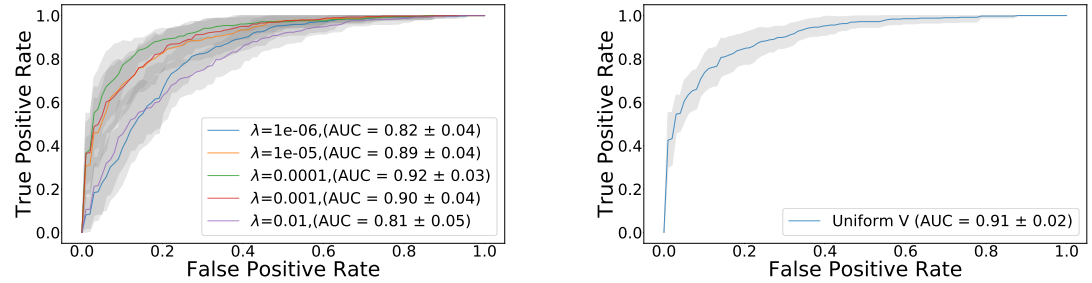
Figure 7.5: Change in the number of annotators for different value of λ



(a) ROC curves for different values of λ with the AUC results

(b) ROC curve for model with uniform annotator weights

Figure 7.6: Comparison among results with different λ and with the baseline for the simple expertise model



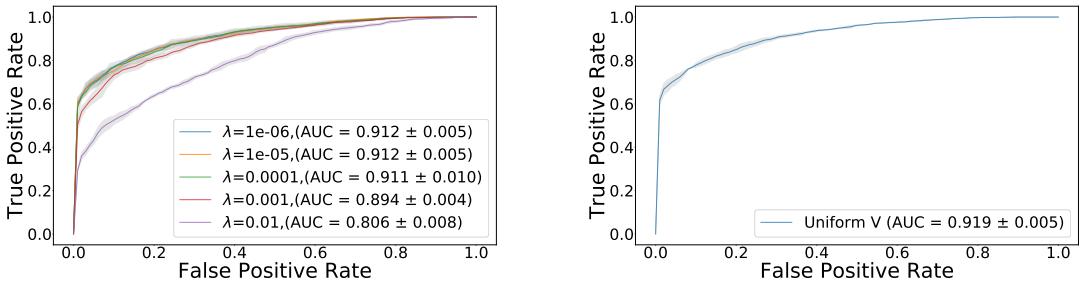
(a) ROC curves for different values of λ with the AUC results

(b) ROC curve for model with uniform annotator weights

Figure 7.7: Comparison among results with different λ and with the baseline for the full expertise model

annotations from 56 antiviruses of different quality. Each antivirus provides annotations for malicious and benign samples. We use 7444 malware samples available from our collection, and create a dataset based on raw byte sequences. For each sample we gather the information if the particular antivirus detected it or not, and create the appropriate label for binary classification. Using this data and the previously known ground truth we execute a preliminary test similar to the one with the Amazon Mechanical Turk. As a difference from the previous dataset, here we don't use logistic regression as classification model. Instead we take the MalConv neural network, a well-known architecture for malware detection using raw bytes [92].

Figure 7.8 shows an insight into the results that we retrieved using our communication-efficient approach.



(a) ROC curves for different values of λ with the AUC results

(b) ROC curves for model with uniform annotator weights

Figure 7.8: Comparison among results with different λ and with the baseline for the simple expertise model for our malware dataset

7.4 Limitations and Future Work

In the future it would be interesting to complete the study on multiple datasets and on more variants of underlying machine learning models in order to further prove the reliability of this approach. Furthermore, it would be beneficial to examine more closely the scalability of the solution and also focus on the direction of incremental learning.

7.5 Summary

In this work we consider a problem of communication-efficient ensemble learning. We envision a specific ensemble learning scenario where each member of the ensemble receives the same input feature values, but creates different annotations based on different expertise levels. These annotations are then used to model the labeling process per annotator. We design an architecture to use knowledge from a small number of annotators

to create a performant learning system. This architecture is tested on real-world data and shows that an optimized model for annotator selection can give similar performance to a model trained using all the available annotators. It can be particularly useful for supervised or semi-supervised anomaly detectors where the reliability of data annotators and the budget for annotations are limited.

Chapter 8

Conclusion and Future Work

In this chapter we summarize our thesis and provide an outlook on what topics we consider important for the future exploration.

8.1 Summary

Anomaly detection includes many scenarios where an event which does not conform to the regular functionality of a system needs to be detected in order to analyze its properties and overcome possible faults. As a consequence of this prevalence, methodology for detecting anomalies has a high importance, especially in computer security, where a potential attacker can obtain unauthorized access to the computer system and execute harmful functionality. Data-driven techniques that use machine learning for detecting outliers constitute an important set of techniques for detecting anomalies. These approaches range from supervised, semi-supervised to unsupervised anomaly detection. Furthermore, they include multiple machine learning paradigms, from topic modeling, kernel methods, to neural networks and deep learning. In this thesis we make use of this breadth of possible approaches by deriving particular methods for scenarios with special constraints. We examine constraints imposed by these use case scenarios and adapt baseline approaches accordingly. With this we explore the tradeoff between anomaly detection performance and these constraints.

8.2 Contribution

This thesis contains multiple contributions to constrained anomaly detection for specific scenarios:

- We design and implement a semantics-aware online learning approach to the analysis and classification of large amount of security logs consisting of system calls

executed by malware samples. Our implementation is based on the extension of the Hierarchical Dirichlet Process framework to online semi-supervised classification, whereby we make our approach work under constraint of constant influx of samples for analysis.

- We use convolutional networks for malware detection based on instruction-level and raw byte features. Furthermore, we design a heuristics for a gradient-based attack on these detection methods and indicate their vulnerability.

- We analyze the possibility of behavior-based user authentication on smartphone devices by collecting sensor values from touchscreen interaction, accelerometer and gyroscope readings. Using these values we train a resource-constrained model that is usable on a mobile device instead of sending data for external processing, proposed in previous publications.

- We analyze the scenario of resource-constrained sequential anomaly detection through the case of virtual machine monitoring and large-scale malware analysis. In order to enable efficient malware tracing, we design a method for detection of a small number of relevant events in system call sequences that gives a high classification performance for a small resource budget.

- We identify a scenario of communication-constrained learning from multiple data labelers. This scenario motivates a framework for communication-efficient learning where model performance is optimized simultaneously with the determination of a small number of essential labelers. The result is a joint optimization model that enables efficient classification and anomaly detection based on labels from a small number of highly confident and reliable annotators.

8.3 Future Outlook

This thesis is motivated by the lack of existing solutions for scenarios with constraints in anomaly detection. While there are many publications and tools for anomaly detection, most of them do not consider this issue. We propose solutions for some important particular scenarios, however there are multiple other ideas for extensions of this work:

- Large-scale graph data

  Graph analytics can be difficult to implement in case that we need to extract summary information out of large-scale connected structures. One example is a social network, where many different problems are difficult to solve at scale, such as community detection, assessment of node importance, detection of various types of frauds and general anomaly detection. Especially in case of hypergraphs or time-evolving graphs, the data becomes very high-dimensional [7]. Another example

is the detection of malicious activity from data flows in programs. For example, by following the intended behavior of Android applications through the data flow graph, one could detect if an application tries to extract user's private data and send it over the network without authorization. Although this connection can sometimes be detected by analyzing single Android programs [12], we can also benefit from analyzing a large set of those programs. However, here we can run into a problem of scalability [54].

- Fairness, accountability, transparency, interpretability

  In many contemporary publications there is a new problem introduced in machine learning systems, which is how to make the mechanism of decision support using these systems more transparent [72]. Especially in more complex methods, such as deep neural networks, it is often difficult to interpret the decisions of the model and fully trace it back to the changes in the input data. Furthermore, recent research shows the existence of a tradeoff between performance and interpretability of machine learning systems. Interpretability is of special importance for anomaly detection, as the detection of anomalies is often a precursor to system diagnostics and maintenance.

- Distributed training

  Furthermore, a very important topic in the recent years is the approximate learning by distributing the model updates to multiple machines [35]. Especially in case of large-scale data, it is very beneficial to divide the data points into multiple chunks and perform the update in small batches, which are in turn accumulated. Since there is a tradeoff between the accurate and slow update on one central machine and fully distributed updates, there is a need to optimize this tradeoff for concrete applications in anomaly detection.

Furthermore, future work should be focused on bringing the solutions for particular scenarios with various constraints to a common theoretical framework, which could be used to rapidly instantiate new solutions for constrained anomaly detection problems.

Bibliography

[1] ObjDump UNIX Tool. https://sourceware.org/binutils/docs/binutils/objdump.html.

[2] PEInfo Service. https://github.com/crits/crits_services/tree/master/peinfo_service.

[3] VirusTotal. http://www.virustotal.com.

[4] Virustotal statistics. https://www.virustotal.com/en/statistics/. Accessed: 2019-04-09.

[5] ZDNet Most Popular Downloads. http://downloads.zdnet.com/popular/.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[7] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, 29(3):626–688, 2015.

[8] Victor M. Alvarez. Yara 3.3.0. VirusTotal (Google, Inc). http://plusvic.github.io/yara/, 2015.

[9] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. Graph-based malware detection using dynamic analysis. *Journal in computer Virology*, 7(4):247–258, 2011.

[10] Brian Anderson, Curtis Storlie, and Terran Lane. Multiple Kernel Learning Clustering With an Application to Malware. In *IEEE International Conference on Data Mining (ICDM)*, 2012.

[11] Hyrum S Anderson, Anant Kharkar, Bobby Filar, and Phil Roth. Evading machine learning malware detection. *Black Hat*, 2017.

[12] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[13] Adam J. Aviv, Katherine Gibson, Evan Mossop, Matt Blaze, and Jonathan M. Smith. Smudge attacks on smartphone touch screens. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[14] Michael Bailey, Jon Oberheide, Jon Andersen, Z Morley Mao, Farnam Jahanian, and Jose Nazario. Automated Classification and Analysis of Internet Malware. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2007.

[15] Yoshua Bengio. Learning Deep Architectures for AI. *Foundations and trends in Machine Learning*, 2(1):1–127, 2009.

[16] Wei Bi, Liwei Wang, James T Kwok, and Zhuowen Tu. Learning to predict from crowdsourced data. In *UAI*, pages 82–91, 2014.

[17] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402. Springer, 2013.

[18] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

[19] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[20] Cheng Bo, Lan Zhang, Xiang-Yang Li, Qiuyuan Huang, and Yu Wang. Silentsense: Silent user identification via touch and movement behavioral biometrics. In *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, MobiCom '13, pages 187–190, New York, NY, USA, 2013. ACM.

[21] Kristian Bredies and Dirk Lorenz. Iterative soft-thresholding converges linearly. Technical report, Zentrum für Technomathematik, 2007.

[22] Daniel Buschek, Alexander De Luca, and Florian Alt. Improving accuracy, applicability and usability of keystroke biometrics on mobile touchscreen devices. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 1393–1402, New York, NY, USA, 2015. ACM.

[23] Giovanni Cavallanti, Nicolo Cesa-Bianchi, and Claudio Gentile. Tracking the best hyperplane with a simple budget perceptron. *Machine Learning*, 69(2-3):143–167, 2007.

[24] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[25] Olivier Chapelle, Bernhard Schölkopf, Alexander Zien, et al. Semi-Supervised Learning. 2006.

[26] Duen Horng Chau, Carey Nachenberg, Jeffrey Wilhelm, Adam Wright, and Christos Faloutsos. Polonium: Tera-Scale Graph Mining and Inference for Malware Detection. In *SIAM International Conference on Data Mining (SDM)*, 2011.

[27] Michael Cogswell, Faruk Ahmed, Ross Girshick, Larry Zitnick, and Dhruv Batra. Reducing overfitting in deep networks by decorrelating representations. *arXiv preprint arXiv:1511.06068*, 2015.

[28] Gabriella Contardo, Ludovic Denoyer, and Thierry Artières. Recurrent neural networks for adaptive feature acquisition. In *International Conference on Neural Information Processing*, pages 591–599. Springer, 2016.

[29] The MITRE Corporation. Collaborative Research Into Threats. `https://crits.github.io/`, Apr 2015.

[30] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[31] Koby Crammer, Jaz Kandola, and Yoram Singer. Online classification on a budget. In *Advances in neural information processing systems*, pages 225–232, 2004.

[32] George Cybenko. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[33] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-Scale Malware Classification Using Random Projections and Neural Networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[34] Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. *arXiv preprint arXiv:1612.08083*, 2016.

[35] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[36] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv: Machine Learning*, 2017.

[37] Susan T Dumais. Latent Semantic Analysis. *Annual review of information science and technology*, 38(1):188–230, 2004.

[38] Tudor Dumitras and Darren Shou. Toward a Standard Benchmark for Computer Security Research: The Worldwide Intelligence Network Environment (WINE). In *Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2011.

[39] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases With Noise. In *Kdd*, 1996.

[40] Mario Frank, Ralf Biedert, Eugene Ma, Ivan Martinovic, and Dawn Song. Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication. *CoRR*, abs/1207.6231, 2012.

[41] Pedro Garcia-Teodoro, J Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-Based Network Intrusion Detection: Techniques, Systems and Challenges. *Computers & Security*, 28(1):18–28, 2009.

[42] Hugo Gascon, Sebastian Uellenbeck, Christopher Wolf, and Konrad Rieck. Continuous authentication on mobile devices by analysis of typing motion behavior. In *Sicherheit*, pages 1–12. Citeseer, 2014.

[43] Xavier Glorot and Yoshua Bengio. Understanding the Difficulty of Training Deep Feedforward Neural Networks. In *International Conference on Artificial Intelligence and Statistics*, 2010.

[44] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[45] Dan Goodin. Fingerprint lock in Samsung Galaxy 5 easily defeated by whitehat hackers. `https://arstechnica.com/security/2014/04/fingerprint-lock-in-samsung-galaxy-5-easily-defeated-by-whitehat-hackers/`, Apr 2014.

[46] Claudio Guarnieri, A Tanasi, J Bremer, and M Schloesser. The Cuckoo Sandbox, 2012.

[47] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[48] Zachary Hanif, Telvis Calhoun, and Jason Trost. Binarypig: Scalable Static Binary Analysis Over Hadoop. *Black Hat USA*, 2013.

[49] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep Into Rectifiers: Surpassing Human-Level Performance on Imagenet Classification. In *IEEE International Conference on Computer Vision*, 2015.

[50] Katherine Heller, Krysta Svore, Angelos D Keromytis, and Salvatore Stolfo. One Class Support Vector Machines for Detecting Anomalous Windows Registry Accesses. In *Workshop on Data Mining for Computer Security (DMSEC)*, 2003.

[51] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[52] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[53] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.

[54] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 106–117. ACM, 2015.

[55] Wenyi Huang and Jack W Stokes. Mtnet: a multi-task neural network for dynamic malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 399–418. Springer, 2016.

[56] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Conference on Computer and Communications Security (CCS)*, 2011.

[57] Yoon Kim. Convolutional Neural Networks for Sentence Classification. *arXiv preprint arXiv:1408.5882*, 2014.

[58] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[59] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 533–537. IEEE, 2018.

[60] Bojan Kolosnjaji, Ghadir Eraisha, George Webster, Apostolis Zarras, and Claudia Eckert. Empowering convolutional networks for malware classification and analysis. In *Neural Networks (IJCNN), 2017 International Joint Conference on*, pages 3838–3845. IEEE, 2017.

[61] Bojan Kolosnjaji, Antonia Huefner, Claudia Eckert, and Apostolis Zarras. Learning on a Budget for User Authentication on Mobile Devices. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018.

[62] Bojan Kolosnjaji, Apostolis Zarras, Tamas Lengyel, George Webster, and Claudia Eckert. Adaptive semantics-aware malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 419–439. Springer, 2016.

[63] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, pages 137–149. Springer, 2016.

[64] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification With Deep Convolutional Neural Networks. In *Advances in neural information processing systems*, 2012.

[65] Ludmila I Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. John Wiley & Sons, 2004.

[66] Yann LeCun and Yoshua Bengio. Convolutional Networks for Images, Speech, and Time Series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[67] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[68] Tamas K Lengyel, Steve Maresca, Bryan D Payne, George D Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, Fidelity and Stealth in the Drakvuf Dynamic Malware Analysis System. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[69] Kingsly Leung and Christopher Leckie. Unsupervised Anomaly Detection in Network Intrusion Detection Using Clusters. In *Australasian Conference on Computer Science*, 2005.

[70] H. Li and Q. Liu. Cheaper and Better: Selecting Good Workers for Crowdsourcing. *ArXiv e-prints*, February 2015.

[71] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, volume 6, page 2, 2013.

[72] Zachary C Lipton. The mythos of model interpretability. *ICML Workshop on Human Interpretability in Machine Learning*, 2016.

[73] M. Frank, R. Biedert, E. Ma, I. Martinovic, and D. Song. Touchalytics: On the Applicability of Touchscreen Input as a Behavioral Biometric for Continuous Authentication. *IEEE Transactions on Information Forensics and Security*, 8(1):136–148, 2013.

[74] Kyle Maxwell. Maltrieve. `https://github.com/krmaxwell/maltrieve`, Apr 2015.

[75] H Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, et al. Communication-efficient learning of deep networks from decentralized data. *arXiv preprint arXiv:1602.05629*, 2016.

[76] Al Mead. Review of the development of multidimensional scaling methods. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 41(1):27–39, 1992.

[77] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[78] Dejan Milojičić, Ignacio M Llorente, and Ruben S Montero. Opennebula: A cloud management tool. *IEEE Internet Computing*, 15(2):11–14, 2011.

[79] Rahul Murmuria, Angelos Stavrou, Daniel Barbará, and Dan Fleck. Continuous authentication on mobile devices using power consumption, touch gestures and physical movement of users. In *International Workshop on Recent Advances in Intrusion Detection*, pages 405–424. Springer, 2015.

[80] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[81] Lakshmanan Nataraj, S Karthikeyan, Gregoire Jacob, and BS Manjunath. Malware Images: Visualization and Automatic Classification. In *International Symposium on Visualization for Cyber Security*, 2011.

[82] Yurii Nesterov. *Introductory Lectures on Convex Optimization: A Basic Course*. Springer Science & Business Media, 2013.

[83] David Newman, Chaitanya Chemudugunta, Padhraic Smyth, and Mark Steyvers. Analyzing Entities and Topics in News Articles Using Statistical Topic Models. In *Intelligence and Security Informatics*, 2006.

[84] Francesco Orabona, Joseph Keshet, and Barbara Caputo. The projectron: a bounded kernel-based perceptron. In *Proceedings of the 25th international conference on Machine learning*, pages 720–727, 2008.

[85] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[86] Nicolas Papernot, Patrick D. McDaniel, Ananthram Swami, and Richard E. Harang. Crafting adversarial input sequences for recurrent neural networks. *CoRR*, abs/1604.08275, 2016.

[87] Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware Classification With Recurrent Networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015.

[88] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[89] Roberto Perdisci and Man Chon U. VAMO: Towards a Fully Automated Malware Clustering Validity Analysis. In *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[90] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Leveraging string kernels for malware detection. In *International Conference on Network and System Security*, pages 206–219. Springer, 2013.

[91] Matt Pietrek. An In-Depth Look Into the Win32 Portable Executable File Format. *MSDN magazine*, 17(2):80–90, 2002.

[92] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K. Nicholas. Malware detection by eating a whole exe. *arXiv preprint arXiv:1710.09435*, 2017.

[93] Daniel Ramage, David Hall, Ramesh Nallapati, and Christopher D Manning. Labeled LDA: A Supervised Topic Model for Credit Attribution in Multi-Labeled Corpora. In *Conference on Empirical Methods in Natural Language Processing*, 2009.

[94] Vikas C Raykar, Shipeng Yu, Linda H Zhao, Gerardo Hermosillo Valadez, Charles Florin, Luca Bogoni, and Linda Moy. Learning from crowds. *Journal of Machine Learning Research*, 11(Apr):1297–1322, 2010.

[95] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[96] Matilda Rhode, Pete Burnap, and Kevin Jones. Early-stage malware prediction using recurrent neural networks. *computers & security*, 77:578–594, 2018.

[97] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and Classification of Malware Behavior. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2008.

[98] J-Michael Roberts. Virus Share. `https://virusshare.com/`, Nov 2015.

[99] Filipe Rodrigues, Francisco Pereira, and Bernardete Ribeiro. Learning from multiple annotators: distinguishing good from random labelers. *Pattern Recognition Letters*, 34(12):1428–1436, 2013.

[100] Filipe Rodrigues, Francisco C Pereira, and Bernardete Ribeiro. Gaussian process classification and active learning with multiple annotators. In *ICML*, pages 433–441, 2014.

[101] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 65–79. IEEE, 2012.

[102] Thomas Rückstieß, Christian Osendorfer, and P Patrick van der Smagt. Sequential feature selection for classification. In *Australasian conference on artificial intelligence*, pages 132–141. Springer, 2011.

[103] Tim Salimans, Diederik Kingma, and Max Welling. Markov chain monte carlo and variational inference: Bridging the gap. In *International Conference on Machine Learning*, pages 1218–1226, 2015.

[104] Joshua Saxe and Konstantin Berlin. Deep Neural Network Based Malware Detection Using Two Dimensional Binary Program Features. *arXiv preprint arXiv:1508.03096*, 2015.

[105] Florian Schaub, Ruben Deyhle, and Michael Weber. Password entry usability and shoulder surfing susceptibility on different smartphone platforms. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, MUM '12, pages 13:1–13:10, New York, NY, USA, 2012. ACM.

[106] Bernhard Schölkopf, Robert C Williamson, Alex J Smola, John Shawe-Taylor, and John C Platt. Support vector method for novelty detection. In *Advances in neural information processing systems*, pages 582–588, 2000.

[107] Matthew G Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J Stolfo. Data Mining Methods for Detection of New Malicious Executables. In *IEEE Symposium on Security and Privacy*, 2001.

[108] Shai Shalev-Shwartz et al. Online learning and online convex optimization. *Foundations and Trends® in Machine Learning*, 4(2):107–194, 2012.

[109] Sheng-syun Shen and Hung-yi Lee. Neural attention models for sequence classification: Analysis and application to key term extraction and dialogue act detection. *arXiv preprint arXiv:1604.00077*, 2016.

[110] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *arXiv preprint arXiv:1312.6034*, 2013.

[111] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[112] Gianluca Stringhini, Manuel Egele, Apostolis Zarras, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. B@bel: Leveraging Email Delivery for Spam Mitigation. In *USENIX Security Symposium*, 2012.

[113] James Surowiecki, Mark P Silverman, et al. The wisdom of crowds. *American Journal of Physics*, 75(2):190–192, 2007.

[114] Benjamin Taubmann and Bojan Kolosnjaji. Architecture for resource-aware vmi-based cloud malware analysis. In *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*, pages 43–48. ACM, 2017.

[115] Benjamin Taubmann, Noelle Rakotondravony, and Hans P Reiser. Libvmtrace: Tracing virtual machines. 2016.

[116] Florian Tegeler, Xiaoming Fu, Giovanni Vigna, and Christopher Kruegel. Botfinder: Finding Bots in Network Traffic Without Deep Packet Inspection. In *International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2012.

[117] Yee Whye Teh, Michael I Jordan, Matthew J Beal, and David M Blei. Hierarchical Dirichlet Processes. *Journal of the american statistical association*, 101(476), 2006.

[118] Yuandong Tian and Jun Zhu. Learning from crowds in the presence of schools of thought. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 226–234. ACM, 2012.

[119] Laurens Van der Maaten and Geoffrey Hinton. Visualizing Data Using T-Sne. *Journal of Machine Learning Research*, 9(2579-2605):85, 2008.

[120] Michael Velten, Peter Schneider, Sascha Wessel, and Claudia Eckert. User identity verification based on touchscreen interaction analysis in web contexts. In *Information Security practice and experience*, pages 268–282. Springer, 2015.

[121] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence-video to text. In *Proceedings of the IEEE international conference on computer vision*, pages 4534–4542, 2015.

[122] Martin J Wainwright and Michael I Jordan. Graphical Models, Exponential Families, and Variational Inference. *Foundations and Trends in Machine Learning*, 2008.

[123] Chong Wang, John W Paisley, and David M Blei. Online Variational Inference for the Hierarchical Dirichlet Process. In *International Conference on Artificial Intelligence and Statistics*, 2011.

[124] Joseph Wang, Kirill Trapeznikov, and Venkatesh Saligrama. Efficient learning by directed acyclic graph for resource constrained prediction. In *Advances in Neural Information Processing Systems*, pages 2152–2160, 2015.

[125] Zhuang Wang, Koby Crammer, and Slobodan Vucetic. Breaking the curse of kernelization: Budgeted stochastic gradient descent for large-scale svm training. *Journal of Machine Learning Research*, 13(Oct):3103–3131, 2012.

[126] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting Intrusions Using System Calls: Alternative Data Models. In *IEEE Symposium on Security and Privacy*, 1999.

[127] Peter Welinder, Steve Branson, Serge J Belongie, and Pietro Perona. The multidimensional wisdom of crowds. In *NIPS*, volume 23, pages 2424–2432, 2010.

[128] Georg Wicherski. Pehash: A Novel Approach to Fast Malware Clustering. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[129] Han Xiao and Claudia Eckert. Efficient Online Sequence Prediction With Side Information. In *IEEE International Conference on Data Mining (ICDM)*, 2013.

[130] Han Xiao and Thomas Stibor. A Supervised Topic Transition Model for Detecting Malicious System Call Sequences. In *Workshop on Knowledge Discovery, Modeling and Simulation*, 2011.

[131] Hui Xu, Yangfan Zhou, and Michael R Lyu. Towards continuous and passive authentication via touch biometrics: An experimental study on smartphones. In *Symposium On Usable Privacy and Security, SOUPS*, volume 14, pages 187–198, 2014.

[132] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057, 2015.

[133] Yan Yan, Rómer Rosales, Glenn Fung, Ramanathan Subramanian, and Jennifer Dy. Learning from multiple annotators with varying expertise. *Machine learning*, 95(3):291–327, 2014.

[134] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 371–372. ACM, 2014.

[135] Apostolis Zarras, Antonis Papadogiannakis, Robert Gawlik, and Thorsten Holz. Automated Generation of Models for Fast and Precise Detection of HTTP-Based Malware. In *Annual Conference on Privacy, Security and Trust (PST)*, 2014.

[136] Cheng Zhang, Judith Butepage, Hedvig Kjellstrom, and Stephan Mandt. Advances in variational inference. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[137] Cheng Zhang, Carl Henrik Ek, Xavi Gratal, Florian T. Pokorny, and Hedvig Kjellström. Supervised hierarchical dirichlet processes with variational inference. In *2013 IEEE International Conference on Computer Vision Workshops*, pages 254–261, 2013.

[138] Ping Zhang, Weidan Cao, and Zoran Obradovic. Learning by aggregating experts and filtering novices: a solution to crowdsourcing problems in bioinformatics. *BMC bioinformatics*, 14(12):S5, 2013.

[139] Dengyong Zhou, Olivier Bousquet, Thomas Navin Lal, Jason Weston, and Bernhard Schölkopf. Learning With Local and Global Consistency. *Advances in Neural Information Processing Systems*, 16(16):321–328, 2004.