Machine Learning for Classifying Malware in Closed-set and Open-set Scenarios

by

Mehadi Seid Hassen

A dissertation submitted to
Florida Institute of Technology
in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Melbourne, Florida
April 2018

We the undersigned committee hereby recommend that the attached dissertation

be accepted as fulfilling in part the requirements for the degree of

Doctor of Philosophy in Computer Science

Machine Learning for Classifying Malware in Closed-set and Open-set Scenarios

by

Mehadi Seid Hassen

Philip K. Chan, Ph.D.
Associate Professor, Computer Science
Advisor and Committee Chair

Marius Silaghi, Ph.D.
Associate Professor, Computer Science

Shengzhi Zhang, Ph.D.
Assistant Professor, Computer Science

Georgios Anagnostopoulos, Ph.D.
Associate Professor, Electrical and Computer Engineering

Phil Bernhard, Ph.D.
Associate Professor, Director of School of Computing

Abstract

Title: Machine Learning for Classifying Malware in Closed-set and Open-set Scenarios

Author: Mehadi Seid Hassen

Committee Chair: Philip K. Chan, Ph.D.

Anti-malware vendors regularly receive large amount of suspected malware files to be examined. However, the sheer number of files makes manual analysis time-consuming. Therefore, it is important to automate this process. Two of the main automation approaches are malware classification and clustering, where similar malware samples are grouped into malware families. Grouping malware into families allows malware analysts to examine fewer representative samples from each family, hence streamlining the malware defense process.

In this dissertation, we focus on two aspects of the automated malware defense. For the first part of our work, we focus on malware classification in a closed set scenario. The assumption in this scenario is that instances seen during testing are from the same set of classes that are seen in training. We explore ways to improve the scalability of feature extraction while retaining discriminative information about the malware sample. We propose a method for extracting features from function call graphs (FCGs). Our proposed approach achieves a linear time complexity compared to previous FCG-based approaches which have quadratic time complexity both in the size of the dataset and the size of the graph. Experimental results also indicate that our proposed feature also improves the classification accuracy compared to past research.

For the second part of our work, we propose supervised and unsupervised approaches for handling open set scenarios. Unlike closed set scenarios where the training data distribution is the same as the test data distribution, in open set, test data contains instances from data distributions that were not seen during training. First, we present an approach that builds on the output of an existing malware classifier and extracts features from its output to perform open set recognition. Then, we present a supervised neural-network-based representation in which instances from the same class are close to each other while instances from different classes are further apart. We evaluate this approach on two malware datasets; one Windows malware dataset and another Android malware dataset. We also evaluate the approach on an image dataset to show that it can be applicable to other domains. The evaluation shows that our representation results in a statistically significant open set recognition performance improvement when compared to a state of the art approach on the three datasets.

Finally, we extend our neural-network-based representation by combining it with Adversarial Autoencoders (AAE) to address unsupervised open set recognition problem. Our evaluations on three datasets (two malware, and one image dataset) show that our proposed approach gives improved open set recognition performance.

Contents

List of Figures

List of Tables

Acknowledgements

In the name of Allah, the Most Beneficent, the Most Merciful. All praise and thanks be to Allah.

There are many people I would like to thank who helped me in the course of this program. First and for most, I would like to express my deepest gratitude to my advisor, Dr. Philip K. Chan, for his constant guidance, for teaching me how to ask research questions, and for being an all-round great advisor. I am going miss our weekly discussions.

I would like to thank my parents for instilling in me the value of education and hard work from an early age. I would like to thank my sisters and brothers for their moral support and encouragement. They have always been there for me to advise in times of difficulty and to celebrate my achievements.

Last but not least, I would like to thank my friends and lab colleagues for making this journey an enjoyable one.

Dedication

To my parents.

Declaration

I declare that the work presented in this dissertation is my own work except where cited to another author. Parts of the material in this dissertation have been previously published by the author. For a complete list of publications, please refer to Appendix C at the end of this dissertation.

Chapter 1

Introduction

Anti-malware vendors regularly receive a large number of suspected malware files to be examined. For instance, Microsoft's real-time anti-malware detection products generate tens of millions of files that need to be analyzed on a daily basis [1]. The reason behind this huge influx of malware, as Microsoft explains, is that to avoid detection, malware authors modify and/or obfuscate malware binaries. This modification attempts to make what would have been otherwise similar malware samples appear different. We refer to these similar samples as belonging to a single malware class or "family".

A large number of malware released daily makes it very difficult for human experts to analyze all of the files. As a result, anti-malware vendors respond slowly to new malware. In some instances, as Giovanni Vigna [2] shows, even after two months of being released, new malware go undetected by one-third of anti-virus scanners. Therefore, there is a need for automated techniques to aid human analysts by grouping similar malware samples into one malware families. Doing so enable human analysts to focus their attention and analyze fewer representative samples per family. It also facilitates taking common defensive measures against an entire family rather than individual samples.

Categorizing similar instances into groups is a classic machine learning task. In the presence of labeled training data classification techniques can be used is for classifying malware samples into families. The challenge in using these classification techniques mostly arises from extracting meaningful features from malware samples. Since there are a large number of malware binaries to extract features from, the feature extraction also needs to be scalable.

Most research works in malware classification operate under a "closed set" assumption (i.e., assuming that the instances seen during testing are from the same set classes as the instances seen in training). However, it is not possible to collect training samples representing all malware families. Besides, because of the adversarial nature of this problem domain, new malware families are released each day by the malware authors. As a result, malware classifiers are forced to operate in an "open set scenario" where they face instances during testing from unknown classes (i.e., classes that are not in training set). Hence, it is essential for these classifiers to identify when instances come from unknown classes in addition to discriminating between the known classes.

In cases where the availability of labeled malware data is limited, unsupervised machine learning techniques such as clustering have been used to identify similar malware and group them into clusters representing families. Human analysts can then examine representative samples of each cluster. Similar to malware classification systems, malware clustering systems also operate in an open set scenario. Hence, the cluster models should be capable of identifying instances that are not similar to any of the clusters. In other words, these systems should be able to identify when the test data distribution is different from the training data distribution.

1.1 Problem Statement

Machine learning based malware defense systems that help automate and alleviate the burdens of human analysts need to fulfill certain requirements. First, the feature extraction process needs to retain more useful information about the malware. Second, the feature extraction process should be scalable to cope up with the influx of malware releases. Finally, the classification and clustering systems should be capable of operating in an open set scenario.

In this dissertation we are interested in addressing three main requirements the above mention system. These are:

- Improving the closed set classification performance and scalability of malware classification systems.

- Identifying instances belonging to unknown classes/families of malware in supervised manner.
- Identifying instances belonging to unknown classes/families of malware in unsupervised manner.

1.2 Approach

We start by first exploring effective and efficient malware feature extraction. We propose a new malware feature extraction approaches based on vector representation of a function call graph (FCG) that is scalable and retain structural information about the malware code. Although we start features extraction with a closed set assumption, we move on to further propose both supervised and unsupervised approaches for handling open set scenarios. We first propose an approach that extends a closed set classifier by extracting features from the classifier output to make it work in an open set scenarios. We also propose a neural network based open set recognition methods (both supervise and unsupervised) that are not limited to the malware domain. We showcase the applicability of our proposed approach to other domains by evaluating them on open set image recognition task.

1.3 Key Contributions

The following are some of our key contributions:

- Our FCG vector representation based feature extraction has a linear time complexity both in the number of malware samples and in the number of function call graph vertices. Our feature compares very favorably to previous FCG-based features which had quadratic time complexity both in the number of samples and the number of vertices.
- Our FCG vector representation results in improved classification accuracy compared to previous approaches.
- Our supervised open set recognition system that extends closed set classifier gives improved performance over earlier research works in the area.

- Our neural-network-based supervised open set recognition approach gives a statistically significant improvement over state of the art neural-network-based open set recognition system.
- We propose neural-network-based unsupervised open set recognition approach that has a statistically significant improvement in recognition performance compared to other methods.
- We show the applicability of our a neural network based open set recognition methods (both supervise and unsupervised) to other domains outside malware by evaluating on an image dataset.

1.4 Dissertation Structure

In Chapter 2 we provide a broad literature review in the application of machine learning for malware defense. We categorize existing works in this area based on the type of features they use and the type of machine learning approach they employ.

In Chapter 3, we explore malware features used for closed set malware classification in previous research works and give a baseline performance on the evaluation dataset. We also propose a new way of extracting opcode n-gram features based on control statement shingling that reduces the dimensionality of opcode n-gram features.

In Chapter 4, we present a scalable and effective function call graph (FCG) vector representation to be used as malware feature. Our proposed feature has a linear time complexity both in the number of malware samples and in the number of function call graph vertices. Additionally, our approach also gives improved classification performance when compared earlier FCG features.

In Chapter 5, we propose a supervised open set recognition system for malware. This approach builds on the output of existing malware classifier and extracts features from the output of the classifier to perform open set recognition. We evaluate our proposed approach on two malware datasets. Our proposed solution achieves improved open set recognition compared to earlier approaches.

In Chapter 6, we present a supervised neural-network-based representation for addressing the open set recognition problem. In this representation instances from the same class are close to each other while instances from different classes are further apart. We show that our representation results in open set recognition performance improvement when compared to state of the art approach on three datasets; Windows malware dataset, and Android malware dataset, and an image dataset to show how the proposed approach can be applied to other domains.

In Chapter 7, we present an unsupervised open set recognition approach by combining Adversarial Autoencoders (AAE) with our work presented in chapter 6 to learn a representation that facilitates open set recognition. We evaluate our approach on three datasets; two malware datasets and one image dataset to show how the proposed approach can be applied to other domains. According to the evaluation results, our approach gives improved unsupervised open set recognition performance compared to other methods.

Finally, we conclude this dissertation by presenting a summary of our approaches and contributions in Chapter 8.

Chapter 2

Literature Review

In this chapter we summarize the various research efforts that use machine learning for malware defense. We start, in section2.1, by giving an overview of the machine learning based malware defense process and give a brief descriptions of its components. Section 2.2 deals with the different types of machine learning features used in past research work. Finally, we survey the different machine leaning techniques that have been employed by research works in this area.

2.1 Malware Defense

A malware defense system can be logically viewed as consisting of two parts. The malware analysis part and the defense mechanism. This holds true for many systems that will look at through the length of this chapter.

In this section we will look at an overview of these two parts and the sub categories that exist within them. We will start by discussing malware analysis techniques and move on to the different defense mechanism that use these techniques.

2.1.1 Malware Analysis

In the context of machine learning based malware defense, malware analysis techniques are used for extracting information that will be directly used as features or to extract information that will be used to derive features. Broadly speaking, malware analysis can be categorized into static analysis and dynamic analysis techniques.

Static Analysis

In static analysis features are extracted from a program without running the binary. These features ranging from simple ones such as binary byte sequence [3, 4] to more complex features such as function-call graph [5, 6].

There are many aspects of static analysis that make it an attractive technique for malware analysis. Firstly, the amount of resource and time required for static analysis are much smaller compared to dynamic analysis. Secondly, they provide a more complete view of the malware binaries when compared to dynamic analysis. The later will be evident when we discuss dynamic analysis in the next section.

Static analysis, however, is not without its weaknesses. Since many of these techniques work by analyzing disassembled code they are susceptible to obfuscation by packing utilities as well as to malware obfuscation such as polymorphism and metamorphism. For instance, A Moser et al. present a code obfuscation technique, based on opaque constants, for obscuring program control flow, disguising access to local and global variables, and interrupt tracking of values held in processor registers [7].

Dynamic Analysis

Dynamic analysis, on the contrary, treats a malware sample as a black box and executes it in a sandbox, controlled environment, while tracing and monitoring its behavior. Behavior tracking can be performed at different levels. Some frameworks such as CWSandbox [8] provide tracking windows API function call level while others such as Anubis [9,10] can support much fine grained tracking at instruction level.

CWSandbox [8] uses a virtual machine environment as its sandbox and uses API hooking for intercepting calls to windows API functions of the monitored program. Using this technique it is able to monitor file system changes, mutex creation, process management operations, and registry modifications performed by the malware being analyzed. In addition to windows API functions, CWSandbox also monitors the malware's network communication activities.

Anubis analysis environment [9,10] works by instrumenting the virtual machine monitor used for running the malware analysis sandbox environment. This instrumentation allows it to track windows API calls and native API calls in addition to the call arguments. It can also perform dynamic taint analysis and monitor network communication activities.

Dynamic analysis can be further fine grained by using dynamic taint analysis. It is a technique for tracking the propagation of input data from untrusted sources during a program execution. Tracking input data allows the analyst to observe when tainted data is used in dangerous ways [11]. Additionally, it allows the identification of data flow dependencies between different components of a program.

Higher performance overhead is one of the drawbacks of dynamic analysis when compared to static analysis. In addition to these overheads, the observed behavior of a malware is not complete due to many factors. Firstly, malware samples can only be observed for a limited time hence only the behavior that is observed within this time frame is available for analysis. Secondly, malware authors can use various techniques to detect whether the execution is being monitored and can alter their behavior accordingly.

2.1.2 Malware Classification

We now shift our discussion to the second part of malware defense systems. We will start with malware classification systems and then discuss malware detection systems in the next section.

Nearly one millions malware samples are submitted for analysis to anti-virus vendors. As can be seen in [12], more than half of these are variants of an existing malware. Malware variants are obtained as a result making an incremental change to an existing malware or automated changes using techniques such as polymorphism, metamorphism, and using binary packing software.

Malware samples that exhibit similar functionality while having small differences are grouped together and are referred to as belonging to the sample malware family. Because of this similarity malware defense efforts can be facilitated by having an automated system capable identifying similarities and grouping(classifying) malware samples into their respective families. Such systems can be referred to as malware classification systems.

In different literatures, malware classification could mean classification into different malware families or classification between benign and malicious samples. In this dissertation, unless explicitly mentioned, malware classification is used to refer to classification into families. We will use malware detection to refer classification between benign and malicious files.

2.1.3 Malware Detection

Malware detection is the process of discriminating malicious programs from benign ones. Machine learning based malware detection systems can be categorized into:

Misuse Based Detection Malware detection is performed by modeling the behavior of malicious programs and benign programs separately and classifying a new sample into which of the two behaviors it belong to.

Anomaly Based Detection In contrast to misuse based detection, anomaly detection works by modeling a normal behavior of programs and flag any activity that falls outside this as malicious.

2.2 Features for Classification and Detection

Using static and dynamic analysis techniques, discussed in Section 2.1.1, different types of features can be extracted from malware binaries. We categorize these features into content based, reputation based and other types of features that do not belong in either of the two, shown in figure 2.1.

2.2.1 Content Based Features

The first set of features we will look are extracted from the content of the malware binaries, i.e. code, header and data.

Simple Features

In this section we will look at simple features that can be extract from a malware sample with minimum amount of preprocessing.

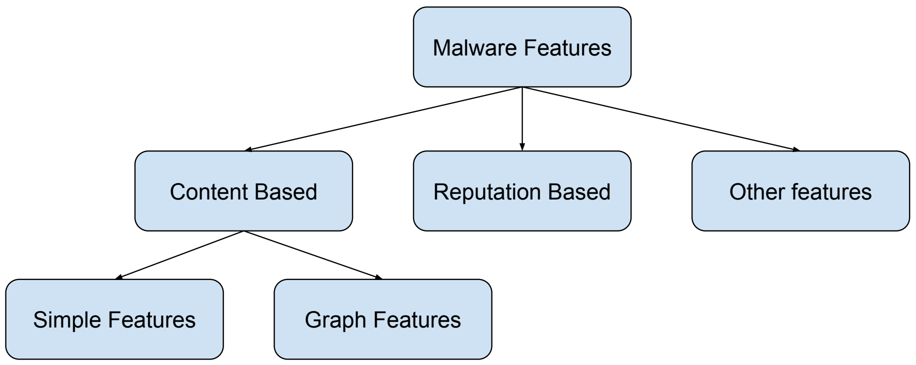


Figure 2.1: Taxonomy of Malware Features

One such feature is function length, measured in bytes, from disassembled malware sample as used in [13]. The function lengths are represented as a histogram with predefined number of bins. Then these function length frequency values are used used as features for a malware classification.

Another set of features are based on printable strings extracted from malware binaries through static or dynamic analysis. String features used in [3, 13] are extracted using static analysis. Where as in [14], null-terminated strings are extracted as they are observed in process memory, when the malware is executed under analysis environment.

A program import table, which is one of the section of a PE file, contains information about external libraries and the function in those libraries that are imported by a program. In [3, 4] they use this information to build a binary feature vector used for malware detection.

Instruction n-grams [4, 15, 16] are extracted from a disassembled program binary where n-grams are constructed from instruction sequences. The instructions in this case can be represented in terms of their mnemonics or in terms of instruction opcode, excluding the operands. In [15] they highlight one of the challenges of using these features, which is the high dimensionality of the features. Even for a bi-gram feature, the number possible bi-grams is in the tens of thousands. They use a hashing trick for dimensionality reduction, where the large feature space is hashed in a smaller dimension using a uniformly distributed hash function.

In [3, 4], they use binary file byte sequence n-gram as feature for malware detection. Unlike Instruction mnemonic or opcode n-gram sequence, these feature do not require disassembling the binary. The sequence n-gram extracted here includes contents of all sections of a binary, unlike the opcode sequence which only looks at the code segment of the binary.

In contrast to the features seen so far which are extracted mainly through static analysis, system-call sequence based features, such as [17], are extracted through dynamic analysis. Rieck et al. [18] go a step further and propose what they call Malware Instruction Set, a new layered representation of system-call and their arguments. In this representation, the system call names, which constitute the first layer, are encoded using hex values. The following layers contain the arguments to the system-calls reorganized based on their importance. This layered representation allows them to extract features at different levels of granularity and gives them control over how descriptiveness of the representation. Finally, they extract n-grams of these instructions and represent them as binary feature vectors and use is them for malware classification and clustering.

Dahl et al. [14] also use system-call trigram and system-call and distinct parameter combination features. To reduce the high dimensionality of the initial features, they use random projection using a sparse projection matrix R, whose elements are iid from a distribution over $\{0, 1, -1\}$.

A hight level abstraction is built on top of system-call dependencies in [19] to make the representation more resilient to code obfuscation. In this abstraction, a program is described in terms of OS objects(such as file, registry, etc.) and operations on these objects(such as read, write, open, etc.). In addition to operations, dependencies between system calls are abstracted as dependencies between OS objects. Finally, control flow dependencies are captured into comparison between OS objects. This abstract representation is then converted into a sequence of short texts which can then be given as features to the clustering algorithm.

Graph Features

There are tree main types graph information that can be extracted from malware samples; function-call graphs, control flow graphs and system-call dependency graphs. A function-call graph is directed graph representation of code where the graph vertices correspond to functions(procedures) and the edges represent the caller callee relation between the functions(vertice) [20]. A control flow graph is defined by Frances E Allen [21] as a "directed graph in which the nodes represent basic code blocks and the edges represent control flow paths". A basic control block is in turn defined as "a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed)". A system call dependency graph, Figure 2.2, is a directed graph where the vertices represent a system call and the edges represent a data dependency between system calls, usually determined by dynamic taint analysis discussed in section 2.1.1.

Function-call graphs(FCGs) are usually constructed from disassembled binary code using static analysis. Various works [5, 6, 22–24] have used call graphs to extract features for malware classification, indexing and clustering. After malware binaries are represented as FCGs, an approximate graph edit distance(GED) is used by some to measure the similarity between pair of FCGs. In [6, 23] they use Simulated annealing [25] to approximate GED. Hu et al. [5] on the other hand use the Hungarian Algorithm to approximate GED.

However, there are other works that do not use GED for measuring graph similarity. For example [22] uses the normalized number of common edges, between two graphs, as a measure of similarity. Dullien and Rolles [26] approximate graph similarity via fixed points and propagations. A fixed point between two graphs is defined as a two nodes(one each from graph) that can be easily determined to represent the same item in both executables. Their algorithm starts from an initial fixed point and propagates to mode fixed points by considering the neighboring nodes. Kong and Yan [24] use function-call graphs to extract new features and use these features to compute the similarity between two graphs.

Nikolopoulos and Polenakis [27] extract system call dependency graph from a malware sample using dynamic taint analysis and then convert it into a smaller graph where the vertice represent a group of system-calls that serve similar purpose and the edges represent the dependency between these groups based on the dependency between individual system calls. Once this representation is extracted they define different similarity metrics for detection and classification.

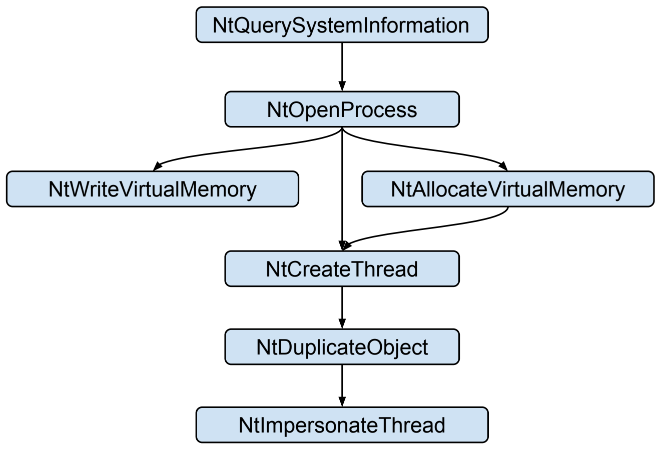


Figure 2.2: Sample system-call dependency graph

2.2.2 Reputation Based Features

Unlike content based features discusses so far, reputation based features focus on the association among files and association between files and machines. This is based on the intuition that good application are used by many users and come from known publishers. On the contrary bad application appear on fewer machines and come from unknown publishers.

Nachenberg et al. [28] infer a file's reputation, or it's goodness, based on its associated machine reputations. To achieve they transform the problem of calculating file reputation into a probabilistic graph model and they construct an undirected machine-file bipartite graph. The nodes in this graph correspond to individual machines and files. The edges, undirected and unweighted, connect the files with the machines that reported them. They also incorporate prior knowledge about the machine, the files and the machine-file relationship. Machine prior is based on machine reputation, which is calculated based on their proprietary formula. The file prior for labeled files is calculated based on the ground truth about the file's goodness or badness. Finally, the machine-file relationship is used to model the intuition that good files appear on good machines and good machines contain good files. They use belief propagation to iteratively infers the labels from the node and its neighbors.

Tamersoy et al. [29] work is based on the intuition that good files are expected to strongly co-occur with other good files and bad files are expected to strongly co-occur with other bad files. Each sample file is represented as set of machines it appears in. They use locality sensitive hashing , MinHash [30] a technique for approximating jaccard index, to cluster the document into buckets. Then an unweighted bipartite graph is built between the files and the buckets. The prior probability for labeled files "goodness" is set to 0.99 and 0.01 for benign and malicious labels, respectively. On the other hand the prior of unlabeled files and buckets is set to 0.5. Then belief propagation is used to compute the probability that an unlabeled file is malicious or benign through massage passing between the files and the buckets.

2.2.3 Other Features

In [31] meta-data is collected about programs running on a computer and these features are used for malware classification. The motivation for using these types of features, as pointed out in the paper, is that collection and in-depth analysis of all suspicious files is not scalable, there for they propose working with meta-data features. These include features such as file name, type of file, digital signer name, signer authority, signature type description in the PE header, organization, version and locality sensitive hashing fingerprint of the file content.

A very different approach is taken in [32–34] which works by visualizing malware binaries as images and using the texture information in this image to classify malware. To visualize, a malware sample is read as 8 bit unsigned integer, where each byte represents a pixel in the gray scale image. These pixels are organized in 2D array with a fixed width and variable height(which depends on the size of the malware binary). In the papers, they show that images of different malware in the same family appear similar. Once converted in to this representation, the problem of malware classification is reduced into one of image texture classification.

2.3 Machine Learning Algorithms

2.3.1 Malware Detection

In this section we will present the machine learning algorithms that have been use in past research works in the area of malware detection.

In reputation based features [28,29], as discussed section 2.2.2, a bipartite graph is constructed and then belief propagation is used to assign class labels to unlabeled files for malware detection. The purpose of belief propagation is to approximate the marginal class distribution of each file. The algorithm works in two phases. In the first phase(the message passing phase), messages are exchanged along each edge in the bipartite graph. These messages symbolize a node's belief about its neighbors based on the current nodes neighbor. Message passing is repeated until the messages converge or maximum iteration is reached. In the second phase, the beliefs of each unlabeled node are computed based on the prior probability of that file and the messages received from its neighboring nodes.

RIPPER algorithm is used in [3] with import table, string and byte sequence features for malware detection. RIPPER [35] a rule learning algorithm consisting of two training phases the rule learning phase and rule set optimization phase. In the rule learning phase RIPPER iteratively adds rules to the rule set until a stop condition is satisfied. Each rule in the rule set is grown by repeatedly adding a conjunction, consisting of an attribute value pair, that maximizes FOIL's information gain until the rule covers no negative example. After the rule adding stop condition is satisfied, the entire rule set will be optimized by considering either keeping the current rule, adding more conditions to the rule, or replacing the rule with a new rule all together to minimizing error on the entire data set. In addition to classifying RIPPER, Schultz et al. [3] also experimented with Naive Bayes algorithm. Naive Bayes uses Bayes' theorem for computing the likelihood that a given binary is malicious give features. Where the features are assumed to be independent given the class label.

2.3.2 Malware Family Classification

Another area where machine learning has been applied in malware defense is for automatically identifying malware families. Dahl et al. [14] use a logistic regression model for classifying malware samples into their respective families. Logistic regression models the posterior probability of K malware families via linear function in input features, while ensuring that they sum to one [36]. The model parameters(weights) can be trained by maximum likelihood using the conditional probability of the data given the model parameters.

In addition to using logistic regression, Dahl et al. also experiment with using neural networks. The neural network architecture in this case consists of 4000 input units, followed by hidden layer of sigmoid units and finally a softmax output layer.

In a slightly different approach, [24] uses an ensemble of classifiers. They first extract different types of features based on the malware function-call graphs. They then train different base classifiers, such as k-Nearest Neighbor , for each feature type. Finally, during classification take an ensemble of these classifier where the importance of each classifier is determined based on the confidence level associated with each. They use Adaboost algorithm to learn the confidence level associated with each classifier.

Techniques seen so far require hand crafted features to be extracted from the malware binaries. However in recently, deep learning has emerged as a very powerful machine learning technique in various fields such as image recognition [37], voice recognition [38]. One of the attractive aspects of deep leaning based approach is their ability to work with raw data and automatically learning high level features [39]. Researchers at Microsoft have demonstrated this possibility in [40] using Recurrent Neural Networks(RNN) [41] for malware classification. The RNN is used as a means of automatically learning malware features, or as the paper puts it "to learn the malware language", to be given as an input to another classifier such as logistic regression. The training input to the RNN are the API call sequence made by malicious program. RNN is trained to predict the next API call, and the weights of the trained hidden units are used as input to another classifier which is used to classify the malware samples.

2.3.3 Malware Clustering

Kinable et al. [6] use k-medoids and DBSCAN clustering algorithms for clustering malware samples into families. The k-medoids algorithm is a modified version of the famous k-means algorithm. In k-means the cluster centroids represent the average of cluster members. However, in this cases the cluster members are the malware function-call graphs and calculating the average is not trivial. So they instead use k-medoids which allows for selecting one of the cluster members to be specified as the cluster center, called cluster medoid. The medoid is going to be the call graph that has the highest similarity with all other members of a cluster. The algorithm starts from selecting the initial cluster medoids and then repeatedly updates cluster membership and recomputing the cluster medoid until a stop criterion is met.

Kinable et al. further evaluate grouping malware in to families using Density-Based Spatial Clustering(DBSCAN) clustering algorithm. The intuition behind their choice of DBSCAN is that instead of assuming that all malware samples in a family are mutually similar to a single parent sample, one can assume malware evolves. DBSCAN [42] is a density based clusting algorithm where instead of requiring all cluster members to be similar to the cluster centroid, it has what are called core points. A core point is a cluster member that has more $MinPts$ number of points, within a specific radius *Rad*. Core point can be thought of as multiple centroids for a single cluster, and this allows DBSCAN clusters to have arbitrary shapes and in this case accommodate the assumption of the evolutionary nature of malware. The cluster members that are with in *Rad* distance from any core point but are not them selves core points are called boarder points. Any remaining data points that are not a core points or boarder points are discarded as noise.

Because of the large amount of data that associated with this domain, Xin Hu et al. [15] point out performing clustering on such data is computationally very expensive. Therefore they resort to clustering prototype data points instead of all data points. Starting from a random data point as the first prototype, they iteratively select the next prototype data point. In each iteration, the data point with the largest distance to existent prototypes is selected as the next prototype. This is repeated until the distance from all the data points to their nearest prototype than a predefined threshold. After identifying the prototype points they carry out agglomerative hierarchical clustering on the prototypes. The non-prototype points will be assigned to the cluster of the closest prototype.

Rieck et al. [18] also perform prototype based clustering to make their algorithm scalable. Once the clusters are created they then use these prototypes to classification of new samples, where a new sample is classified as belonging to the family of the prototype it is most similar to, given the similarity is greater than a predetermined threshold. However, in cases where the similarity is less than this threshold they hold that sample as evidence. After enough such samples are collected the same clustering procedure is performed on them in an attempt to incrementally identify new malware families.

A different approach is taken Bayer et al. [19] to make clustering more scalable. After representing malware samples in terms of an abstract behavioral representation and extracting features from this representation, Minhash is used to approximate a set of all near similar pairs with similarity above a given threshold. Minhash is locality sensitive hashing(LSH) technique based on the idea that a hash of a set is the index of the first element under a random permutation. Then the probability that two sets will have the same index value of the first element is equal to the jaccard similarity between the two sets. However generating the permutation is computationally expensive. Hence, in practice the random permutations are approximated by computing the Minhash of a set s as $h(s) = \min(c_1 * x + c_2 \mod P)$ for $\forall x \in s$ where x is the index of an element in s, c_1 and c_2 are constants and chosen at random for each hash function and P is a prime much larger than the size of the universal set of all sets [43]. To improve the precision of the Minhash signature multiple hash functions are used together to generate a Minhash signature. It is still possible to have actually similar sets not to be present in all near similar pairs, to minimize this the previously discussed procedure is run multiple times.

The approximate set of all near similar pairs is obtained using Minhash, however, might contain pairs that are not similar. To handle this, Bayer et al. [19] calculate the exact jaccard similarity of all near similar pairs and those pairs with jaccard index less than a specific threshold are discarded. Afterwards single-linkage hierarchical clustering is applied using all near similar pairs, up to the threshold value. Single-linkage hierarchical clustering is a bottom up clustering technique which starts with each sample forming a cluster of one element, and then repeatedly combining two clusters that are most similar [44]. When choosing which clusters to merge, the similarity between two clusters is calculated as the similarity between pair of elements in the two clusters that are most similar, hence the name single-linkage. In this case, the near similar pairs list is sorted based on similarity and the clustering algorithm runs until there are no pairs left in the list.

Similarly Jang et al. [45] use hash functions to create a compact malware fingerprint which can be used to efficiently compare the similarity between malware samples. They proceed to using using hierarchical clustering on the fingerprint similarities. An interesting aspect of there work is that in addition to clustering the also do feature co-clustering to identify the correlated features among malware samples.

2.3.4 Anomaly Detection

Anomaly detection works by modeling a normal behavior of programs and flag any activity that falls outside this as malicious. In practice, anomaly detection is useful in detecting application vulnerability exploits. In [46] the authors describes their host based anomaly detection technique that models a normal behavior based on the analysis of system call arguments. For each system call used by an application, a distinct profile is created which models the notion of a 'normal' sys-call invocation in terms of what is considered to be 'normal' values for one or more of its arguments. This profile models the system calls string arguments by looking at characteristics such as the string length, string character distribution and regular grammar. During the learning step the mean and variance of the string argument value for a system call are computed. In addition to that, for each observed string argument its character distribution is stored and a Markov model is used to achieve a reasonable generalization of the grammar.

In addition to using the system call argument unary relation as in the previous case, Bhatkar et al. [47] model binary relations between system calls as well. They presents an algorithm to learn the data-flow properties of a program. In the context of their work data-flow property refers to the values of system call arguments and their flow from one system call to a subsequent one. They capture data flow dependencies through binary relations between system-calls in terms of their arguments, where the system calls are ordered in terms of their temporal relationships(time-stamp of a system call). The unary relations between a single system-call and its arguments is modeled in this work include relations such as $equal$, $subsetOf$, $elementOf$, etc. When learning unary relations the type of relation between a system-call and an argument value are identified, the value is stored and when the stored values become larger in number, they are approximated based on the type of relation and the type of value. During detection any unary or binary relations identified that are not observed during training are flagged as anomalous.

Li et al. [48] argue that data-flow relations can not be learned properly with out using information from control-flow to give context to the data-flow. So the main idea of the paper is to put system calls made by a program being monitored into different context and then learn rules that can capture data-flow relationships among system calls within a context and across different contexts. In their case a context of a system call is defined as a fixed sequence of system calls that are made before and after it. They then define three types of rule sets first type covering relations among system calls in the same context, the second type defining relations in repeated contexts and the third type defining relations between system calls in different context. They use minimum support and confidence to filter out weaker rules. The learned rules are finally used to perform on-line monitoring and flag anomalous behavior.

Maggi et al. [49] improves on the work of [46] based on the intuition that some system calls might have many normal behaviors in different portions of a program. To find these subsets of normal behavior for a system call, they perform hierarchical clustering on the invocations of each system call based on their arguments. Once the clustering is done and subsets of each system call invocations are identified they train a Markov model where the states are represent a single subset of a system call and the state transitions represent the system calls sequence.

2.4 Challenges and Conclusion

In this chapter we surveyed various research efforts that use machine learning for malware defense. We started off by explored the different types of features extracted from malware binaries and then showed how previous research works used these features to train machine learning models for tackling various problems in the malware defense domain.

We will conclude our discussion by highlighting the challenges that exist in this domain. Some of these are common in other domains where machine learning is applied while others are more specific to malware.

Malware features such as instruction n-grams have high dimensionality. This presents a challenge for many machine learning algorithms. The papers surveyed here used various techniques such as feature selection, random projection and hashing trick to reduced feature dimensionality.

Scalability is another major issue for malware defense systems, where close to a million malware samples are registered per day. Some researchers have used locality sensitive hashing techniques to improve performance, others have used techniques such as prototype based clustering to allow these systems to scale with requirements of the problem.

Finally, the adversarial nature of this domain presents another challenge. This is evident in the use of code obfuscation and binary packing techniques by malware authors. Malware analysis techniques such as dynamic analysis coupled with unpacking techniques are one way to address these problems. Malware authors are not limited to only using these techniques. Hence malware defense systems need to be able to quickly evolve while at the same time be scalable and effective.

Chapter 3

Malware Classification Using Static Analysis Based Features

3.1 Introduction

According to AV-Test [12] over 300,000,000 malware were registered in 2014 alone, more than half of which were variations of already existing malware. The sheer volume of the problem makes it impossible for a human expert to analyze each malware. Hence, automated classification of malware samples is needed.

In recent years various research projects [5, 15, 50] have focused on developing classification or clustering techniques to automatically categorize malware into malware families. However, there are many challenges, such as scalability and resilience to code obfuscation techniques, faced by these systems.

In our work we explore the use of different features, extracted using static analysis, for classifying malware into different families. We also propose a new feature based on control statement shingling which has comparable classification accuracy with ordinary opcode n-gram based features. These new features also result in shorter training time.

Before presenting our work, we review past researches in this area in Section 3.2. Background about the machine learning algorithm used in this work is presented in Section 3.3. In Sections 3.4 and 3.5 we discuss our solution, and we present our experimental results in Section 3.6.

3.2 Related Work

Malware Classification can refer either to classification of binaries as malicious or benign, or classification of malware samples into different known malware families. Our work deals with the later one. However, for the sake of completeness we will look at past research in both.

In [3], the authors evaluate the effectiveness of different data mining techniques in classifying new unseen binaries as malicious or benign. They extract features based on DLL imports, printable strings and program byte sequence. Then they use Naive Bayes and RIPPER, a rule learning algorithm, to learn models that can be used for classification of new samples. Other works, such as [13], also use program printable string feature in addition to function length features to training machine learning models.

The problem with using program byte sequence feature, as done in [3], is that it is easy to obfuscate program byte sequence using various binary packing techniques such as UPX [51]. One way to address this is to examine the opcode or instruction sequence after disassembling the binary program. X. Hu et.al. [15] use opcode sequence, n-gram, features for clustering malware samples.

To get a better understanding of the semantics of a program, other works such as [5,6,22,52] investigate function call graphs. A call graph provides an abstract representation of a program in the form a directed graph. In this graph, the vertices correspond to functions, and the edges correspond to one or more calls made from one function to another [20].

Hu et. al. [5] implement a database management system, SMIT, to support malware lookups. In SMIT, a malware sample is represented using a call graph. SMIT builds a two-level indexing scheme. The first level uses a B+ tree where the keys are the total number of instructions, total number of control instructions, total number of functions, and the median number of instructions per function. The second level index is an Optimistic Vantage Point Tree(VPT), which takes the similarity of the malware call graphs into account when placing malware into storage buckets.

The similarity measure used here is an approximate graph edit distance on the malware sample call graphs.

In SMIT, each function, vertex of the call graph, is represented in terms of the function name, assembly instruction mnemonics sequence, and a CRC of the instruction sequence to speed up exact matching between functions. Local, statically linked and dynamically linked functions are considered. When matching two vertices from two call graphs, a match based on function name is used if the functions are statically linked or dynamically linked. In case of local function, CRC is first used for exact matching; if that fails, then sequence edit distance is used as a measure of the similarity between the two functions' instruction sequences.

Similar to SMIT, [6] also uses graph edit distance as a measure of similarity between two malware call graphs. Vertex insertion/deletion, unpreserved edge and vertex relabelling cost are considered when calculating approximate edit distance. Unlike SMIT, where vertices corresponding to local functions are matched based on their instruction sequence when considering relabelling cost, they consider matching of external functions only when calculating vertex relabeling cost. They then apply Kmeans and DBSCAN clustering algorithms to cluster similar malware samples together.

An alternative call graph similarity metric, that is based on the number of matching edges, is proposed in [22] for discovering similar malware variants. Vertices of a call graph, which represent functions, are first matched. Similar to the previous two works, vertices representing external functions are matched based on name; however, local functions are matched based on the external functions they call, the cosine similarity of their instruction frequency, or neighboring matching functions. Once the vertices are matched then matching edges are identified and a similarity metric of two call graphs is computed.

3.3 Background

Random Forest is an ensemble learning technique which uses decision trees as the base classifiers. In general, ensemble learning techniques improve prediction accuracy by combining predictions from multiple classifiers. The individual classifiers can be built by [44]:

- Manipulating the training set. For example, bagging, boosting, random forest.

- Manipulating the input feature. For example, random forest.
- Manipulating class labels.
- Manipulating the learning algorithm. For example, varying the initial weights in neural networks.

Random Forest algorithm manipulates both the training set and input features. The general algorithm for random forest is shown in Algorithm 3.1. The algorithm builds T decision trees. For each tree, training samples are randomly selected, based on some distribution, from the original training dataset. The new training set D_t is equal in size to the original D. An unpruned decision tree is then trained on D_t. When building the individual trees, instead of considering all available features for splitting at each tree node, only F randomly selected features are considered. Finally, during classification, a majority vote is taken among all the trees.

There are various advantages to using Random Forests for the dataset and features explored here.

1. Random Forest tend to perform better when the set of feature to select from is large. This allows the construction of de-correlated individual trees, which in turn, improves the prediction performance of random forests. In our case for instance, opcode 4-gram feature after hash trick with 14-bit hash has a feature vector size of $2^{14} = 16384$.
2. Speed of training is fast because at any given tree node, random forest only considers a randomly selected subset of features that are much smaller in number than our entire feature space. Hence, reducing the training time.
3. Prediction accuracy was better than or comparable to other techniques we looked at for Microsoft Malware Classification Challenge Dataset, such as logistic regression, backpropagation artificial neural networks, and decision tree.

Algorithm 3.1: General Random Forest Algorithm

```
Input  : $D$: Training data,
         $T$: Number of decision trees in the random forest,
         $F$: Number of randomly selected features considered at each tree node.
Output: Random Forest with T decision trees base classifiers.
1 for $t = 1$ to $T$ do
2     - Create bootstrap sample $D_t$ data from original training data $D$ by randomly
        selecting elements from $D$ where $|D_t| = |D|$.
3     - Train an unpruned decision tree on $D_t$. At each tree node consider only $F$
        randomly selected features for splitting.
4 end
```

3.4 System Overview

Our malware classification pipeline consists of three components:

- Feature extraction
- Learning Classification Model
- Classification of Unseen Binaries

Feature extraction, discussed in detail in Section 3.5, is the first component in the pipeline. Here, static analysis of a disassembled malicious binary is performed to extract different features that will be used by the machine learning algorithm to learn models to be used for classification of malicious binaries.

In our experiments, Weka's implementation of Random Forest is used both to learn the classification models and classifying samples. Weka [53] is an open source implementation of a collection of machine learning algorithms. The algorithms can be accessed from the command line tool, GUI interface, or from a Java code.

When classifying new samples into the known malware families, we first extract features. Then, we use Weka by giving the trained model and feature vectors of the new samples as input.

3.5 Feature Extraction

The features discussed here are all extracted using static analysis of the disassembled malicious binaries. Static analysis is not the only way to perform analysis of malicious binaries. In dynamic analysis, a malicious binary is run in a sandbox environment while monitoring different aspects of its execution, such as system call, file, network, and registry activities. Despite all the advantages of dynamic analysis, its main shortcoming is its performance overhead. When considering large datasets with thousands of binaries, dynamic analysis does not scale well. Since scalability is one of the requirements of the current work, static analysis is used to extract features from disassembled files of malicious binaries.

To extract features discussed in the next sections, malware binaries need to be unpacked, if packing tools were used to obfuscate the binaries, and then disassembled using unpacking tools.

In the next subsections, we will discuss features that have been used in the past, such as Instruction frequency, opcode n-grams, DLL features, as well as a new control statement shingling based feature proposed in this work.

3.5.1 Assembly Instruction Mnemonics Frequency

The first feature considered is instruction mnemonics frequency. Each malware sample is represented as a vector, where the elements of the vector represent frequency of occurrence of an assembly instruction in that specific malware. The frequency values are then normalized by the total number of instructions in that malware.

3.5.2 Instruction Opcode N-Grams

The second set of features extracted from disassembled files of malicious binaries consider instruction opcode n-gram frequencies. We decided to use opcode n-grams instead of instruction mnemonic n-grams because opcodes are more specific, hence, providing more discriminating features. For example, there are several opcode values that represent an instruction mnemonic *mov* based the operand's location and type.

When extracting opcode n-grams, first a malware file is represented as a sequence of instruction opcodes. Then, n-grams of the instruction opcode sequence are created, and their frequencies are counted and normalized by the total number of opcode n-grams in the malware binary.

Ideally, we would want to represent the occurrence frequency of each possible opcode n-gram as a value in the feature vector. However, there are two challenges associated with that:

- The number of unique n-grams grows exponentially in the number of distinct instructions and the length of n-grams. This impacts both on the learning speed and prediction performance of the machine learning model.
- Sparseness of observed n-grams. In practice, not all of the possible opcode n-grams are observed in a given malware. This results in many opcode n-gram frequencies having zero values.

One way, to address the these problems is to use feature reduction techniques to reduce the feature space into lower dimensions. One such technique is using hashing. Previous works, both in the malware classification domain [15] and other domains [54], have used feature hashing for dimensionality reduction and have shown that it does not result in loss of classification performance, given a sufficiently long hashing bit. In fact, dimensionality reduction tends to reduce overfitting [55].

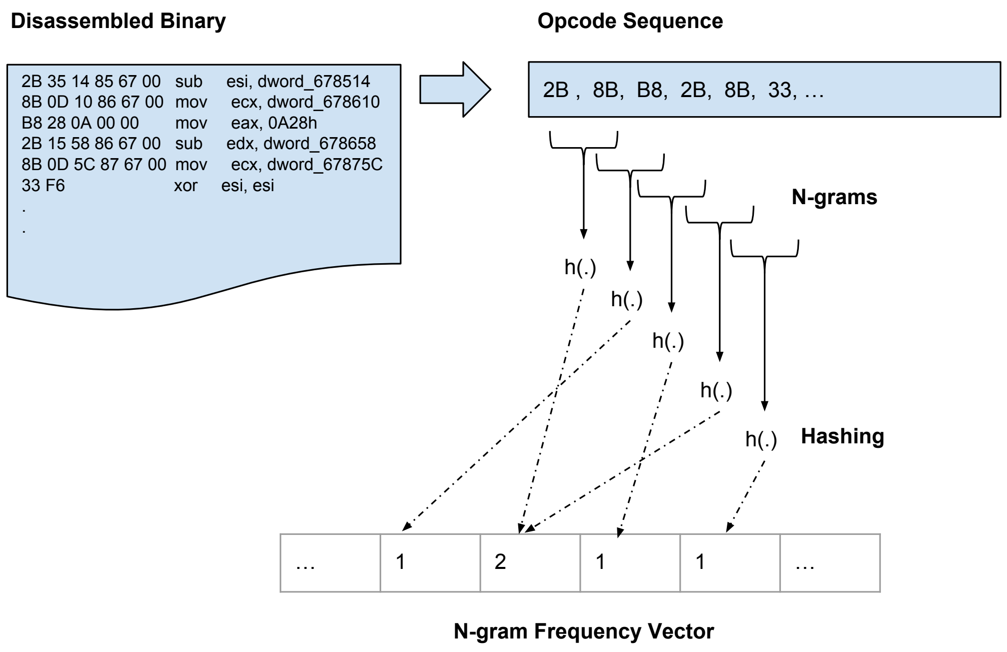

Disassembled Binary
2B 35 14 85 67 00 sub esi, dword_678514
8B 0D 10 86 67 00 mov ecx, dword_678610
B8 28 0A 00 00 mov eax, 0A28h
2B 15 58 86 67 00 sub edx, dword_678658
8B 0D 5C 87 67 00 mov ecx, dword_67875C
33 F6 xor esi, esi
.
.
Opcode Sequence
2B , 8B, B8, 2B, 8B, 33, …
N-grams
h(.)
h(.)
h(.)
h(.)
h(.)
Hashing
…
1
2
1
1
…
N-gram Frequency Vector

Figure 3.1: Extracting opcode n-grams and hashing to reduce dimensionality

The hashing used here works as follows: Opcode sequences are extracted from a disassembled malware binary, and a sliding window, of size n, is moved over the sequence to get the n-grams. In the example shown in Figure 3.1, a sliding window of size 2 to get 2-grams, such as 2B-8B, 8B-B8, B8-2B, etc., are extracted. Then a non-crypto hashing function, which uniformly distributes the keys, is used to hash the opcode n-grams into buckets. The feature vector has one feature for each bucket, where each feature value corresponds to the observed frequency of opcode n-grams hashed to that bucket. In our example, for instance, the 2-gram 2B-8B is observed twice so the frequency vector cell that corresponds to the bucket it hashes to has a count of two. This frequency vector is further normalized by scaling its values to have the same range, say $[-1, 1]$ or $[0, 1]$. In the experimental results section the effect of different number of buckets, which is the result of the bit length of hashing, is discussed in more detail.

3.5.3 Proposed Opcode N-Grams with Control Statement Shingling

Naturally, code is structured into functions, control blocks, and so on. However, the opcode n-gram feature discussed in the previous subsection does not take the structure of code into account. In an attempt to address this, we borrow the idea of word shingling from text document classification, where stop-words are considered as delimiters in the document ,and n-grams of characters, words, or phrases are constructed starting at the delimiter. In the case of disassembled malware code, we consider the opcode of control statements, such as JMP, LOOP, CALL, as stop-words. These control statements delimit the different control blocks in the opcode sequence.

Properly representing control blocks in the extracted feature presents a new challenge, such as how many n-grams to consider to represent each block and whether or not to include the stop-words(control statements) when constructing n-grams. Both of these impact the classification accuracy and feature dimensionality, which in turn affects model training speed. Considering stop-words as part of the n-gram, for instance, results in fewer numbers of unique n-grams, and hence smaller numbers of features. Similarly the smaller number of n-grams considered in each control block also results in fewer unique n-grams.

Giving dimensionality reduction more consideration, we decided to consider only one n-gram, including the stop-words, from each control block. We start by extracting opcode sequences from

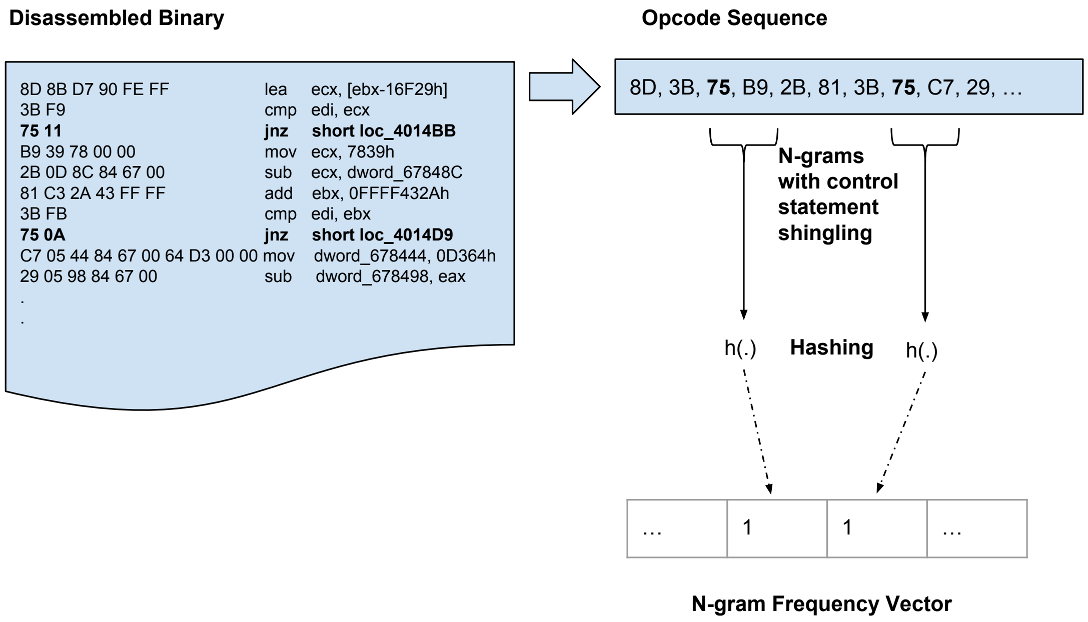

Disassembled Binary
8D 8B D7 90 FE FF lea ecx, [ebx-16F29h]
3B F9 cmp edi, ecx
75 11 jnz short loc_4014BB
B9 39 78 00 00 mov ecx, 7839h
2B 0D 8C 84 67 00 sub ecx, dword_67848C
81 C3 2A 43 FF FF add ebx, 0FFFF432Ah
3B FB cmp edi, ebx
75 0A jnz short loc_4014D9
C7 05 44 84 67 00 64 D3 00 00 mov dword_678444, 0D364h
29 05 98 84 67 00 sub dword_678498, eax
Opcode Sequence
8D, 3B, 75, B9, 2B, 81, 3B, 75, C7, 29, …
N-grams with control statement shingling
h(.)
Hashing
h(.)
… 1 1 …
N-gram Frequency Vector

Figure 3.2: Extracting opcode n-grams and hashing to reduce dimensionality

the disassembled binary. The shingles are then constructed as opcode n-grams starting with a stop-word. For example, in Figure 3.2 JNZ is a stop-word and n-gram is extracted including JNZ. We then use feature hashing to reduce dimensionality of our feature vector. The feature vector after hashing presents the frequency of n-gram shingles in the corresponding bucket.

Besides taking the structure of the code into account, these features have the added advantage of resulting in a fewer number of unique opcode n-grams. This translates to a smaller number of hashing collisions and hence, requires fewer hash bits, resulting in a lower dimensional feature vector, as shown in Section 3.6.2.

3.5.4 Import Address Table(DLL) Feature

Features discussed so far depend on a successful disassembling of the malware binaries. However, this is not always the case, as some advanced malware packers out there make it very difficult to unpack and disassemble the malware binaries. In such cases, we need to look for other features that can be obtained without the need for disassembling. On such feature is the import table information. The import address table is one of the sections of a PE file. It contains information about external libraries and the function in those libraries that are imported by a program [56].

As a machine learning feature, first all the imported libraries(DLLs) in all of the malware samples in the training set are listed. Once this is done, then a binary vector is constructed for them. A cell in this vector is 1 if that library is imported by the malware sample under consideration; it is 0 otherwise. Each malware sample will be represented by this binary vector.

3.6 Experiments and Results

3.6.1 Dataset

The dataset used in this chapter was obtained from the the Microsoft Malware Classification Challenge [1]. It has 10,867 labelled malware samples, which belong to 9 malware families. Table 3.1 shows the class distribution for the nine malware families. For each malware sample, the disassembled file, generated using IDA pro [57], and its binary file, without a PE header, are given.

3.6.2 Hash bit length when using feature hashing with n-gram features

The number of bits used to represent the hash values determines the total number of buckets to which the keys get hashed. The hash bits greatly affect the performance of the machine learning algorithm. If too small, there will be a lot of collisions and many n-gram features will be hashed into the same bucket, hindering descriptiveness of the reduced feature vector. Large values, on

Table 3.1: Dataset class distribution

Malware Family Name	Number of Samples
Ramnit	1541
Lollipop	2478
Kelihos ver3	2942
Vundo	474
Simda	42
Tracur	751
Kelihos˙ver1	398
Obfuscator.ACY	1228
Gatak	1013

the other hand, fail to reduce the dimension of our original opcode n-gram feature which results in the machine learning algorithm taking longer training time, and being susceptible to overfitting. Therefore, it is important to determine the appropriate hash bit length.

To determine the hash bit length that is appropriate for both the opcode n-gram and opcode n-gram with control statement shingling, we conducted experiments by varying both the length of the n-gram and the hash bit length. For these experiments we used Random Forest with 100 base classifier trees (i.e. the T in Algorithm 1). These experiments were conducted on a smaller subset of the original data, with 946 samples and 10 fold cross validation. The results of these experiments are shown in Figure 3.3.

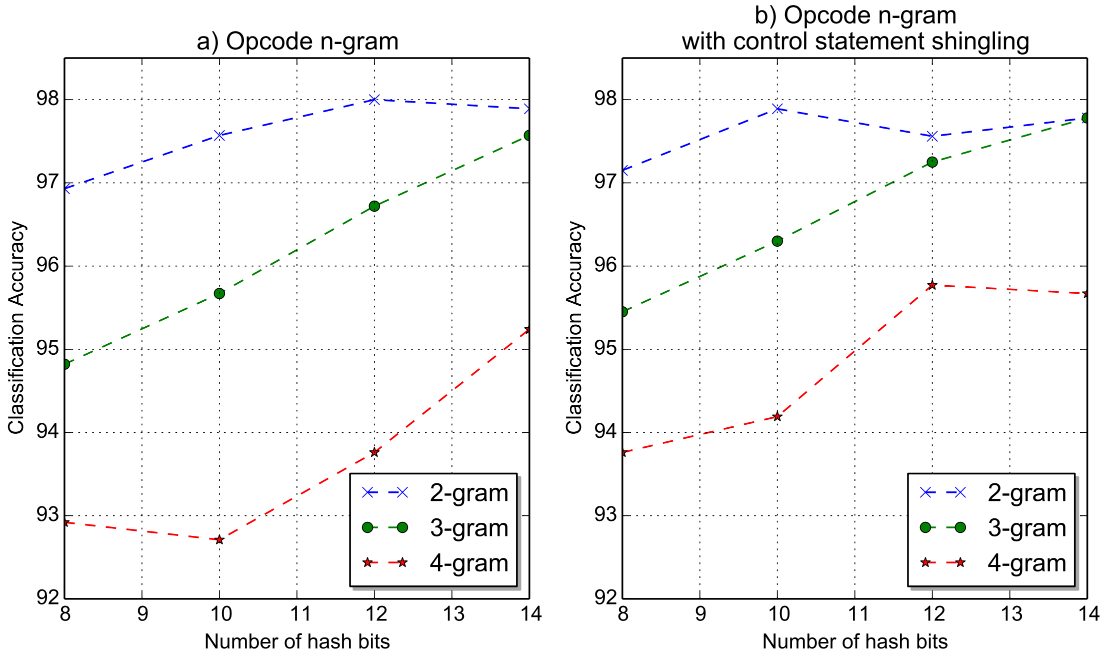


Figure 3.3: Effect of hash bit length on prediction accuracy using Random Forest

For a fixed value of n in n-gram, we expect the prediction accuracy to increase with an increase in the number of bits used to represent the hash value. This is because the larger number of hash bits results in smaller hash collisions. On the other hand, for a fixed hash bit length, we expect the prediction accuracy to decrease as the length of the n-gram increases. This is because the number of possible n-grams increase as n increases, and this will result in higher number of collisions.

The results in Figure 3.3 agree, for the most parts, with our expectation. For the opcode n-gram features, Figure 3.3a, 3-gram and 4-gram accuracy increases as the number of hash bits increase. In the case of 2-grams, however, accuracy for the 14-bit hash is less than accuracy at the 12-bit. For 2-gram opcode sequence there are I^2 distinct 2-gram opcode sequences, where I is the number of distinct x86 opcodes. And a 14-bit hash results in a feature vector of size 2^{14}=16384, resulting in $I^2/16384$ collision on average, if the hashing function uniformly distributes the keys across the 2^{14} buckets. However, the original 2-gram features are very sparse, so even though we expected $I^2/16384$ collisions per bucket, most of the 2-gram frequencies have zero values, thus resulting in the hash buckets also having zero values. This is exactly what we found out when we looked at the feature vector for 14-bit hashed 2-grams. When this sparse feature vector is given as input to our algorithm, Random Forest, it is possible that many of the F features that are considered by the algorithm at each tree node are zero and do not help in distinguishing the malware class. Hence, we see that in the case of 2-gram with 14-bit hashing, the prediction accuracy does not improve. One of the results that we haven't been able to explain has to do with the slight decrease in accuracy in case 4-gram when hash bits are increased from 8-bit to 10-bit in Figure 3.3a. In spite of this decrease, the over all trend of increasing in accuracy with increase in hash-bits is still evident in 4-grams for 12-bit and 14-bit hash.

Opcode n-gram with control statement shingling feature, Figure 3.3b, also exhibits similar behavior. In this case, the optimal hash length for 2-gram is at 10-bit. It is mainly due to the fact that our proposed features resulted in a smaller number of distinct n-grams. This, in turn, results in a fewer number of hash collisions, achieving good classification accuracy even at a smaller hash bit length.

We also conducted experiments to justify our decision to include stop-words (control statements), such as JMP,CALL, etc., when representing each control block in terms of n-grams. The classification accuracy including the stop words using 10-bit feature hashing is 97.89%, whereas, classification accuracy when excluding stop-words using 10-bit feature hashing is lower at 97.04%. The optimal hash bit length when excluding stop-words occurs at 12-bits due to the larger number of unique n-grams. Therefore, not only does including stop-words in the n-grams improve accuracy, it also reduces the number of unique n-grams which, in turn, reduces the feature dimensionality as well as improving model training speed.

3.6.3 Evaluating Features

We now compare the prediction accuracy of the features discussed in Section 3.5. In addition to the individual features, we also look at some of their combinations. Table 3.2 shows the prediction accuracy on the training set using 10 fold cross validation.

The last column in the Table shows the number of data instances considered for each feature. One of the problems we faced when applying n-gram based features to this data set was that not all of the malware samples were properly parsable by our feature extractor. Hence, we see that when using only instruction mnemonics or opcode based features, we are only able to classify 10,260 instances. To take advantage of the high prediction accuracy of n-gram based features and all inclusiveness of DLL boolean features, we combined the two features. Results are shown in the last two rows with prediction accuracy of 98.25% and 97.98% for opcode 2-gram features with 12-bit hashing and opcode 2-gram feature with 10-bit hashing and control statement shingling, respectively.

The new control statement shingling feature proposed in this chapter, opcode n-gram using control statement shingling, performs very well at 99.11% almost equivalent to normal opcode n-gram feature which has accuracy 99.21%. Since this new feature encounters a smaller number of unique n-grams, it has the added advantage that it needs a fewer number of hash bits to represent, hence having a smaller space requirement. In the results shown here, opcode n-gram using control statement shingling is represented using 10-bit hash, compared to the normal opcode n-gram feature which requires 12-bits, hence resulting in almost 4 times as many features.

The proposed control statement shingling feature also gives an insight into malware code. The good classification accuracy of 99.11% attained show that only considering a few instructions after a branching statement such as JMP, CALL and others can help discriminate malware families.

Table 3.2: Prediction accuracy using Random Forest

Features	Accuracy	Instances
Asm instruction frequency	97.71	10260
Opcode 2-gram	99.21	10260
Opcode 2-gram using control stmt shingling	99.11	10260
DLL boolean feature	83.41	10867
DLL with Opcode 2-gram	98.25	10867
DLL with Opcode 2-gram using control stmt shingling	97.98	10867

This implies that a considerable number discriminative operations are performed following a branching statement.

3.6.4 Training Time

The smaller number of features that result from our new feature also translates to a faster training time. As shown in Table 3.3, this is less evident with Random Forest because it works on a random subset of features when evaluating the the features for each tree node. The average training time for Random Forest using the control statement shingling features with 10-bit hash is 1.28 seconds, whereas using n-gram frequency features with 12-bit hash it was 1.52 seconds.

The training time difference becomes very significant when using machine learning algorithms that consider all features for model training. For example, using Logistic Regression, the average training time with our new feature which only needs 10-bit hash is 67.03 seconds, whereas when using normal n-gram features with 12-bit hash is 2169.72 seconds.

These experiments were carried out on a smaller subset of the original dataset consisting of 946 samples. We used Weka's implementation Random Forest and Logistic Regression ran on a machine with 2.6GHz 8-core cpu and 100GB main memory.

Table 3.3: Training time

Feature	Training Time (sec)	
	Random Forest	Logistic Regression
12-bit opcode n-gram	1.52	2169.72
10-bit control stmt shingling	1.28	67.03

3.7 Limitations

The machine learning features used in this chapter rely on the correct disassembly of the malware binary. The use of more advanced binary packers by malware authors can make disassembling difficult. One way to handle this is to use dynamic analysis to run the packed program and dump process memory image once the malware has unpacked itself. This is then given to the disassembler.

The proposed classification technique is for identifying the malware family of a given malware. This requires the malware to belong to, or to be a variation of, one of the already known malware families. Identifying new malware families is beyond the scope of of this work.

3.8 Conclusion

In this work we investigated the problem of classifying malware samples into known malware families. This is an important problem because more than half of the malware released each year are variations of existing malware. By classifying malware samples into families, representative samples of a given family can be analysed by human experts and defensive measures can be taken.

We evaluated various static analysis based features for malware classification. We also proposed a new feature which performs n-gram shingling with control statements as stop-words. We showed that the new feature performs comparably with opcode n-gram feature while requiring smaller feature vector and shorter training time.

Chapter 4

Scalable Function Call Graph-based Malware Classification

4.1 Introduction

In an attempt to preserve the malware codes structural information during machine learning feature extraction, function call graph-based features have been used in various research works [5, 6, 23] in malware classification. However, these approaches are usually based on computing graph similarity using computationally intensive techniques. Due to this, much of the previous work in this area incurred large performance overhead and does not scale well.

In this chapter, we propose a linear time approach for extracting function call graph (FCG) features. We overcome the performance overhead associated with FCG based features by using a novel technique to convert FCG representation into a vector representation based on function clustering. Our proposed approach has the following advantages:

- It is faster compared to previous works for FCG based malware classification.
- It has a higher classification accuracy compared to previous works.

- The graph feature vector, extracted from FCGs, can be combined with other non-graph features.

We will start by first reviewing related research works in Section 4.2. In Section 4.3, we will describe the details of our approach and implementation. And finally, we will evaluate our approach and compare its efficiency and effectiveness with previous research works in Section 4.4.

4.2 Related Works

Graph-based features have been used in many research works for malware clustering and classification. The main attraction of graph-based features is that they preserve information on how different parts of the malware code interact.

There are many types of graph information that can be extracted from malware samples: FCGs, control flow graphs and system-call dependency graphs. FCG is a directed graph representation of code where the vertices of the graph correspond to functions (procedures), and the edges represent the caller-callee relation between the functions (vertices) [20]. FCGs are usually constructed from disassembled binary code using static analysis. Various research works [5, 6, 22–24] have used FCGs to extract features for malware classification, indexing and clustering.

When representing malware as FCGs, be it for classification, indexing or clustering, the fundamental question that needs to be addressed is that of measuring graph similarity. Graph edit distance (GED) is one such metric that is used in various domains for measuring the dissimilarity between two graphs by providing a measure which quantifies the minimum amount of edit operations that need to be performed to transform one graph into the other. The appealing aspects of this metric is its customizability (flexibility), which it provides in the form of vertex and edge related edit distance costs that can be defined to incorporate domain knowledge [58]. However, exact computation of GED has exponential time complexity in the number of vertices. Hence, approximations of the metric are used.

As mentioned earlier, one needs to define cost functions to be used for calculating GED. In [6, 23], the authors define GED in terms of three cost function: vertex insertion/deletion cost, edge cost, and vertex relabeling cost. In their approach, before computing GED, a filtering step is applied to remove most similar function pairs from the two graphs, based on the Jaccard Index of the function instruction frequency vector. Then a random bipartite graph mapping of the remaining vertices between the two graphs is constructed. Simulated annealing [25] is then applied to find a bipartite mapping to approximate graph edit distance. Finally, the approximated GED is used in [6] to perform malware clustering.

In [5], a malware database management system is implemented that indexes malware samples based on their FCG similarity using an approximate GED. Similar to previously discussed works, they also approximate GED by finding minimum cost bipartite graph mapping between the vertices of the two graphs. In their case, however, they used the Hungarian Algorithm to find this mapping. The Hungarian algorithm finds the minimum cost of a bipartite mapping based on an input cost matrix. This matrix specifies the cost of mapping a vertex (function) in one graph to a vertex in a second graph. In their implementation this cost is composed of Relabeling Cost, which accounts for the cost of matching functions, and Neighborhood Cost, which takes into consideration matches between the neighboring vertices. Because of the time complexity of the Hungarian algorithm, they filter out highly similar functions based on names for external functions, and based on the similarity of instruction mnemonic sequences for local functions.

There are other research works that do not use GED for measuring graph similarity. For example, [22] uses the normalized number of common edges between two graphs as a measure of similarity. The authors begin by first matching external functions based on their function names. Local functions are first matched based on their external function calls, and matching functions (vertices) are removed. The remaining functions are next matched based on their instruction opcode. Then, they match the still remaining functions based on whether the neighboring vertices are matched. Finally, the similarity metric is calculated as the number of common edges between the two graphs and normalized by the sum of the number of edges in both graphs divided by two.

A work by Dullien et al. [26] is another example that uses a different metric than GED to measure graph similarity. They approximate graph similarity via fixed point propagation. A fixed point between two graphs is defined as a two nodes (one in each graph) that can be determined to represent the same item in both graphs. Their algorithm starts from an initial fixed point and propagates to more fixed points by considering the neighboring nodes.

In [24], the authors use FCGs to extract new features to compute the similarity between two graphs. They start by extracting FCGs, where each function is represented in terms of size types of initial features. They proceed to learning a distance metric for each attribute type and an optimal vertex matching matrix that maximizes between-class distance while minimizing within-class distance. Finally, they train classifiers for each feature type and combine the results in an ensemble classifier.

In [59], the authors propose a way to map function call graphs (FCGs) to vector form, inspired by linear-time graph kernel [60]. First, FGSs are extracted from android APK files. To label the graph vertices, instructions are grouped into 15 categories. A 15-bit vector is used to label the vertices, which indicates the presence or absence of these instructions. This label is further changed to the neighborhood hash [60], which is computed as bit-wise XOR of the vertices 15-bit vector and all of it's successors vertices. Finally, the neighborhood hash of the complete graph is obtained by calculating hashes for each node individually and replacing the original labels with the calculated hash values. The graphs are represented as multiset of these hash values. This work presents a way of representing these graphs as feature vectors such that the inner product between the two feature vectors is equal to the multi-set intersection of the two graphs.

The difference between [59] and our work arises primarily from the way we label the FCG. In our case, we label vertices by cluster-id of the function clusters. This allows us to represent functions in more detail because all instructions, as well as the sequence of the instructions, are encoded. It also allows us to control the granularity of this labeling by controlling the number of clusters. Secondly, our vector representation explicitly encodes graph edges, hence preserving more information.

System-call dependency is another type graph representation of a malware. A system-call graph is a graph representation where the vertices correspond to system calls made during the execution of the malware, and edges represent data-flow dependency between system calls, usually determined by dynamic taint analysis. In [27], the authors extract this graph from a malware sample using dynamic taint analysis and then convert it into a smaller graph where the vertices represent a group of system-calls that serve a similar purpose and the edges represent the dependency between these groups based on the dependency between individual system calls. Once this representation is extracted, they define different similarity metrics for detection and classification.

4.3 Approach

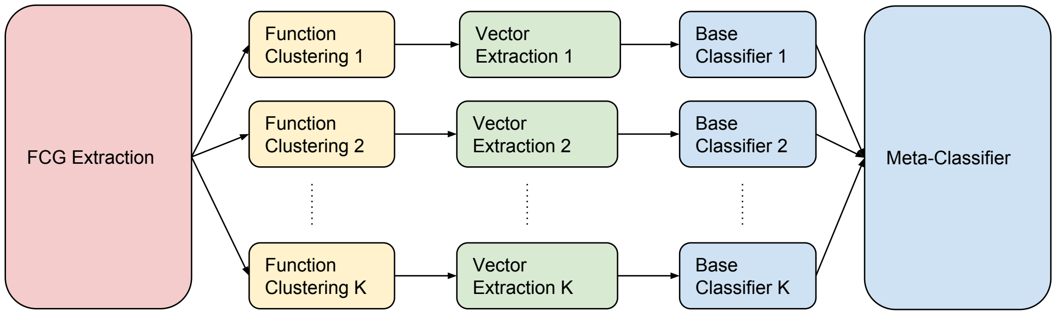


Figure 4.1: System overview with K pipelines

There are various ways to extract features through static analysis, for the purpose of classifying malware using machine learning. Previous research works have extracted features from printable strings in malware binaries [3,13], from the length of the functions in disassembled file [13], from instruction n-grams [4, 15, 16], FCGs [5, 6, 22–24], or a combination of different features [61].

In this chapter we chose to focus on features generated from function call graphs (FCGs). The main reason for this is that FCGs better preserve structural information in binaries, for instance, compared to n-gram features. In addition to containing information about the malware code in the form of functions and their code, they also contain information about the interaction between the functions. The details of FCG extraction are discussed in Subsection 4.3.1.

One of the challenges with using FCG is that the names of the local functions (the functions written by the program author) are lost during compilation; hence, the vertices of FCG corresponding to these functions are unlabeled. This makes it difficult to compare two functions in different FCGs. Our solution to this problem is to cluster functions based on their instruction sequence and use these cluster-ids as labels for the functions. One of the concern we had about clustering functions was that we would lose some information as multiple functions get hashed to same cluster. Even thought this is true, our results show that classification accuracy is still very high. Function clustering is discussed in Subsection 4.3.2.

There are multiple ways of comparing the labeled FCGs, such as graph edit distance [5, 6], fixed point propagation [26], etc. In our approach, in addition to being able to efficiently compare FCGs, we also wanted to be able to have the capability for integrating non-graph features when needed. So unlike many past research efforts that classify FCGs, our approach first converts a FCG into a feature vector and then applies machine learning algorithms on these vectors, as discussed Subsection 4.3.3.

A high level view of our approach is shown in Figure 4.1. Our system starts by first extracting FCG representations from disassembled malware binaries. Once a FCG is extracted, the functions (which are the graph vertices) are clustered using Locality Sensitive Hashing(LSH) based on the function's instruction opcode sequence. The FCG vertices are then labeled using the cluster-ids. After labeling the FCG, our system extracts a vector representation from the call graph, which will serve as the feature vector of the malware sample.

During function clustering, some functions that are similar might be grouped into different clusters due to the randomized nature of LSH functions. To address this shortcoming, during function clustering, graph labeling and extraction of vector representation is done K times. Each of these vector representations, extracted in parallel, are given as input to a separate classifier, which we refer to in Figure 4.1 as a base classifier. This is done both during training and test. The predictions of the base classifier are further used as input features for the meta-classifier. We will describe the functionality of each module in the following subsections.

4.3.1 FCG Extraction

There is much structural information that gets lost when features are extracted from a malware binary. For example, when extracting features, such as instruction n-gram, the organization of malware code into different functions, and the interactions between these functions is not captured in the extracted feature. So in an attempt to preserve the structural information, we use FCGs to represent malware binaries.

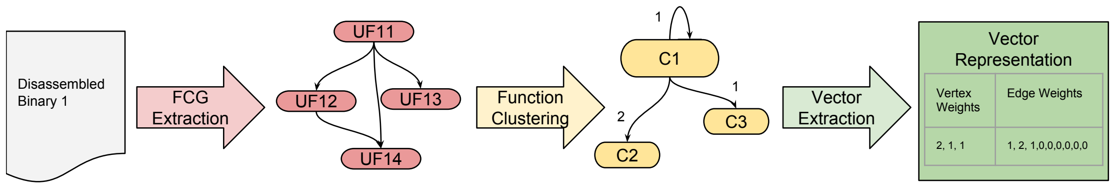
A FCG is a directed graph representation of code where the vertices of the graph correspond to functions and the directed edges represent the caller-callee relation between the functions (vertices) [20]. The vertices in this graph can represent local functions defined in the malware code or external functions imported from libraries.

As shown in Figure 4.1, the first module of our system takes disassembled malware binaries and extracts FCG representations. When extracting FCG, we label vertices of external functions with the function names. The original names of local functions are not preserved during compilation and eventually disassembly. Even if they were, these names may not represent well the instruction sequences that the function implements. Therefore, we leave the vertices corresponding to local functions unlabeled for now. Vertices representing local function also contain the instruction opcode sequence of that function. We represent the caller-callee relation between functions as directed, unweighted edges. After extracting FCG we pass this graph to the next module for clustering the local functions (vertices) and relabeling them with their cluster-id.

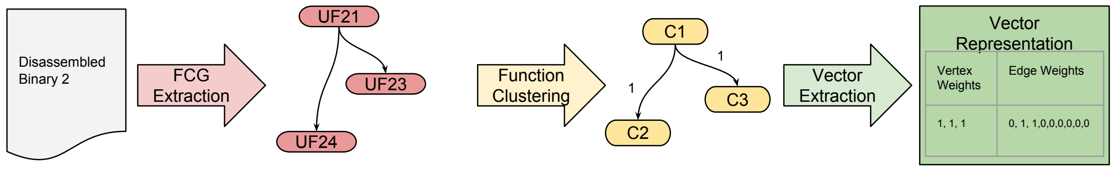
To help explain the different modules, we will use a toy example shown in Figure 4.2. In this example, we assume we have two disassembled binary files. These files are given as an input to the FCG Extraction module, which converts the two samples into a FCG representation. The first sample has FCG consisting of 4 functions: $UF11$, $UF12$, $UF13$ and $UF14$. The second sample has 3 functions $UF21$, $UF23$ and $UF24$. The function names, for the local function, at this point are arbitrary names. Hence, it is difficult to compare the two graphs. We will discuss our solution to make the comparison easier in the next section.

4.3.2 Function Clustering

As discussed in the previous section, local functions in FCG are unlabeled. This makes comparing two FCGs very difficult. Our solution to this problem is to cluster the functions (vertices) of FCG and label them with a cluster-id. To perform clustering, we need to have a way of identifying similar functions. Hu et al. [5], for instance, use the instruction mnemonic sequence edit distance to calculate the similarity between functions which do not have similar names or CRC values of their mnemonic sequence. Even though edit distance might be a good measure of similarity, it has $O(n^2)$ time complexity in the number of instructions. In our work, we try to alleviate this



(a) Sample 1



(b) Sample 2

Figure 4.2: Example

bottleneck using Locality Sensitive Hashing(LSH) for computing an approximate edit distance between the instruction sequences of two functions.

The challenge here is that there are no locally sensitive family of hashes, that we know of, for approximating the edit distance. To address this, we explored the possibility of approximating edit distance with Jaccard Index. We evaluated the effectiveness of Jaccard Index in approximating edit distance by carrying out experiments where we computed the similarity between opcode sequences of functions using edit distance as well as Jaccard Index and computed the Pearson's correlation coefficient between the two. For Jaccard Index, we represent the functions as a set of uni-gram, bi-gram, or trig-ram opcode. Out of the three, Jaccard Index of uni-gram opcodes had the highest Pearson's correlation coefficient with edit distance at 0.957.

Using Jaccard Index to approximate edit distance, however, is still not fast enough. Fortunately, there is a family of locality sensitive hash functions, commonly known as Minhash [30], that can be used to approximate Jaccard Index, allowing us to efficiently approximate the similarity between functions.

Minhash is a LSH technique based on the idea that a hash of a set is the index of the first element under a random permutation. Then the probability that two sets will have the same index value of the first element is equal to the Jaccard similarity between the two sets. However, generating these permutations is computationally expensive. Hence, in practice, the random permutations are approximated by computing the Minhash of a set S as:

$$h(S) = \min_{\forall x \in S}((c_1 * x + c_2) \mod P) \tag{4.1}$$

where

- x is the index of an element in set S w.r.t. the super set of S;
- c_1 and c_2 are constants and chosen at random for each hash function;
- P is a prime much larger than the size of the universal set of all sets [43].

To improve the precision of the Minhash signature, L hash functions in Equation 4.1 are used together to generate a single Minhash signature by concatenating the values from the different hash functions. However, it is still possible to have false negatives (i.e. sets that are highly similar but declared to be not similar by Minhash). Multiple runs of Minhash can be used to address this [62].

Algorithm 4.1: Function Clustering

```
Input  : G: Function call graph. Where functions are represented in terms of their
         instruction opcode sequence.
         [$h_1, \ldots, h_L$]: $L$ hash functions, in Equation 4.1, for generating Minhash
signature.
Output: Function call graph labeled with function cluster ids
1 foreach internal function $v$ in G.vertices do
2     signature ← minhashSignature($v$.ngramOpcodeSequence, [$h_1, \ldots, h_L$]);
3     clusterId ← hash(signature);
4     $v$.label ← clusterId;
5 end
6 return G;
```

In our systems case, instead of using Minhash to directly calculate the similarity between functions, we use it to cluster functions (vertices) of the given graph. Unlike normal clustering where we would have needed to calculate the similarity between each pair of functions, using LSH we can simply hash the functions into buckets which represent clusters. Logically, we can consider this process as if we are clustering all functions from all FCGs. In reality, however, our use of LSH allows us to cluster the vertices in one FCG without the need to look at other FCGs in an efficient way.

The pseudo-code for our implementation is shown in Algorithm 4.1 and the full source code is available on github [1]. For each vertex, we compute the cluster-id by first generating a Minhash signature for the n-gram opcode sequence. The function *minhashSignature*, in line 2, takes the n-gram opcode sequence and a list of hash functions, defined in Equation 4.1, as input and generates the Minhash signature. We represent this signature as an array of L hash values for the n-gram opcode sequence computed using the given L hash functions. Note that Minhash operates on set inputs rather than a sequence, as is the case in line 2, where the input to Minhash is an opcode n-gram sequence. One way to convert the n-gram sequence into a set representation is to have a universal set that contains all the n-grams observed in our training samples, and then the individual functions represented as subsets of this universal set. However, to speed up computation in our implementation we take the hash of an opcode n-gram and use this value as the index of the opcode n-gram when calculating the Minhash. The hash function used to compute the index value can be any hashing function with uniform value distribution and a large range of hash values to minimize collisions. In our case, we use murmurhash [63].

Next, these Minhash signatures are further hashed using an ordinary hash function to compute the cluster-id for that vertex (function) and this cluster-id value is used to label the vertex, as shown in lines 3-4. This secondary hashing also allows us to control the number of clusters.

During our implementations, we also experimented with the use of different hash functions for the secondary hash. As mentioned earlier, a hash function that uniformly distributes its keys across buckets is used. However, we thought it might make better sense to have hash collisions when the Minhash hash signatures closely match. So we experimented with using Simhash [64] as the secondary hash function. Our experiments, however, revealed that although using Simhash did improve the classification accuracy of a single base classifier, the hash function which uniformly distributes its keys achieves better accuracy on the overall meta-classifier. Therefore, we decided to use an ordinary hash function for the secondary hash.

At this point the FCGs, labeled by the cluster-ids, can be logically viewed as a graph where the vertices are clusters and the edges are calls made from functions in one cluster to a function in another cluster, or even in the same cluster. This logical representation is shown in Figure

[1]https://github.com/shrtCKT/FCG-to-Vector

4.2. However, in actual implementation we simply label the the vertices of input FCG with the cluster-ids.

In our running example in Figure 4.2, the FCGs extracted by the FCG Extraction module are given as input to the current module. In this module a Minhash signature is generated for each function in the input FCG, a cluster-id is determined using this signature and the function (vertex) is labeled by this id. In case of sample 1, we assume, functions $UF11$ and $UF12$ are hashed to cluster $C1$, $UF13$ is hashed to cluster $C3$, and $UF14$ is hashed to cluster $C2$. In the case of sample 2, we assume functions $UF21$, $UF23$ and $UF24$ are hashed to clusters $C1$, $C3$ and $C2$, respectively.

The resulting graphs shown in Figure 4.2 are logical view of FCG after labeling. This view shows a graph where the vertices are clusters and the edges are calls made from functions in one cluster to a function in another cluster, or even the same cluster. Now the labeled graphs of sample 1 and 2 are much easier to compare.

4.3.3 Vector Extraction

In past research works, techniques used for computing graph similarity have been a source of performance bottleneck on FCG based malware classification. Not only were these a performance bottlenecks, but they also made it difficult to integrate non-graph features with graph features. Motivated by these two aspects, we proposed a vector representation of FCGs. We would like our vector representation to have two properties: (Property 1) the representation should preserve information about the instructions of malware functions and (Property 2) the representation should preserve the structural information of the FCG.

We extract vector representation from a FCG labeled using the cluster-ids. This representation consists of two parts, vertex weight and edge weight. The vertex weight specifies the number of times a vertex with a given label (cluster-id) is found in a FCG, or in other words the number of vertices in each cluster for that FCG. the vertex weight indirectly preserves information about the function instructions. The edge weight specifies the number of times an edge is found from a vertex in one cluster to a vertex of another cluster or a vertex within the same cluster. The edge weights help preserve the structural information of the FCG.

Algorithm 4.2: Creating Vector Representation

```
Input  : $G$: Function call graph labeled with function cluster ids;
Output: Graph vector representation
1  Initialize vertexWeight with zero vector;
2  Initialize edgeWeight with zero vector;
3  foreach $v$ in G.vertices do
4      vertexWeight [$v$.label ] += 1;
5  end
6  foreach $e$ in G.edges do
7      index ← EdgeIndex($e$.source.label, $e$.target.label);
8      edgeWeight [index] += 1;
9  end
10 graphVector ← concatenate vertexWeight with edgeWeight;
11 return graphVector;
```

As shown in Algorithm 4.2, we start by initializing *vertexWeight* and *edgeWeight* vectors to zero vectors. In lines 3-4, we iterate over all the vertices in the input graph and count the number of vertices labeled with each cluster-id. Next, in lines 6-8 we compute edge weights for the edges between cluster-ids (i.e., an edge from a vertex labeled with one cluster-id to a vertex labeled with another cluster-id) or within a cluster (i.e. an edge between two vertices labeled with the same cluster-id). To do so we iterate over each edge in the input graph and update the frequency of occurrence edges. In our implementation, we use the *EdgeIndex* function to perform a simple lookup for the index representing the edge type. Finally, we concatenate *vertexweight* and *edgeWeight* vectors to form one vector representing the input FCG.

As mentioned in the Section 4.3.1, vertices of the FCG corresponding to external functions are already labeled by the name of the external functions. While computing the vertex weight we can use a lookup table to map the external function names to index values in the vertex weight vector that correspond to that external function. This requires first identifying all external functions observed in training dataset samples and adds an additional processing. To avoid this, we simply use an arbitrary hash which uniformly distributes its keys to relabel the external functions with this value and use this to as the labels refereed in lines 4 and 7 in Algorithm 4.2. The same applies for the edge weights. Through our experiment we didn't notice significant decrease in classification accuracy as a result of this hashing. Therefore, we decided to use this approach to avoid processing overhead.

We acknowledge that our vector representation does not preserve every detail of a graph structure. For instance, it is possible to have slightly different graphs with the same vector representation. However, we believe that this can have its own advantage in the field of malware classification by making this representation less susceptible to obfuscation techniques that might change function calling patterns. That is because small variations in the graph structure might not get expressed in the vector representation as long as the edge and vertex frequencies are not changed. Hence, this representation becomes more resilient to changes such as reordering of function calls.

In our running example, in the case of sample 1 for instance, the labeled FCG contains two functions that are labeled $C1$, one $C2$, and one $C3$. The vector representation of the vertex weight part will be $2, 1, 1$. The second part represents edge weights. In sample 1, we have one edge $C1 - C1$, two $C1 - C2$ and one $C1 - C3$. The vector representation of the edge weight part will be $1, 2, 1, 0, 0, 0, 0, 0, 0$. The zeros in the edge weight show that there are no edges, for example, $C2 - C2$. The final vector representation will be a concatenation of these two vectors.

Once the vector representation of all instances is created, the next step is to train models for classification of the malware samples.

4.3.4 Base Classifiers

The function clusters generated as discussed in Section 4.3.2 has high precision but lower recall. That is, each cluster has similar functions, but some similar functions may be hashed into different clusters. Recall can be improved by repeating this clustering step. In our system, K repeated clustering steps are run independently and in parallel. For each run, a separate vector representation is computed. Then separate base classifiers are trained using the output of each run.

Each of our base classifier is a Random Forest classifier [65]. The inputs to a base classifier are the vector representations for the malware FCGs. As shown in Algorithm 4.3, the input dataset, which is a list of vector representations for the malware FCGs, is segmented into T parts (line 2). Then for each data segment, a classifier is trained on the remaining data segments and used to predict, in the form of a probability distribution of over all classes, for each sample in

Algorithm 4.3: Training base classifier and creating training data for meta-classifier

```
Input  : D: Training set containing FCGs vector representation
Output: $D_{new}$: The training set represented in terms of class distributions.
1  Initialize $D_{new}$ to empty list;
2  Segment D into T parts;
3  for $t = 1$ to T do
4      Train classifier C on D $-D_t$;
5      foreach Sample d in $D_t$ do
6          $distribution_d \leftarrow$ C.predict(d);
7          Add $distribution_d$ to $D_{new}$;
8      end
9  end
10 return $D_{new}$;
```

the segment (lines 3-7). This is used to reconstruct the data set in terms of the predictions of the based classifier.

4.3.5 Meta-Classifier

The meta-classifier combines the predictions of the individual base classifiers to output a final prediction. It first receives the predictions for each data instance from the based classifiers. These predictions are in the form of probability distribution over the class labels. Then for each instance, these probability distributions are concatenated to form a single feature vector. In other words, the prediction from the base classifiers serve as an input feature to the meta-classifier. The meta-classifier is then trained on this vector. The output of the meta-classifier is a predicted class label for each data instance as illustrated in Figure 4.3.

4.3.6 Enhancements

For a meta-classifier to be effective, we need the base classifiers to be sufficiently different from each other [66]. In the case of our original design presented so far, the difference between the base classifiers comes as a result of the false negatives introduced by function clustering with LSH. Through our experiments, we were able to determine that the different runs of the Function clustering using a same number of Minhash functions(L) were not producing enough variation to take full advantage of ensembling. So to introduce more variation, we used different values for L in Function clustering for the K different runs.

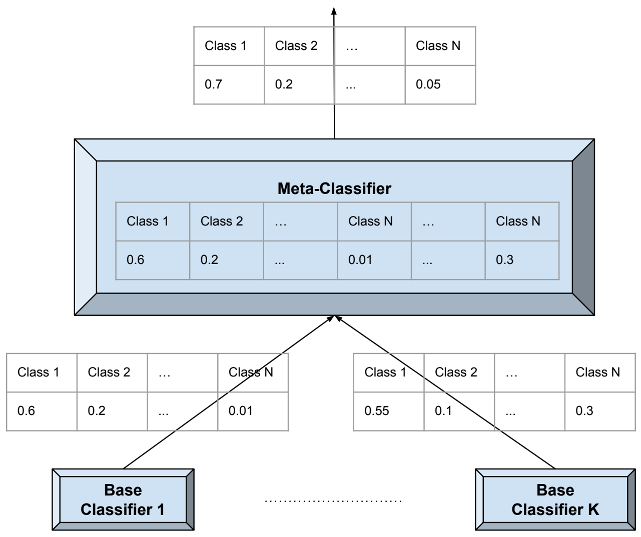


Figure 4.3: The meta-classifier

4.4 Experimental Evaluation

4.4.1 Dataset

For the purpose of evaluating our proposed approach for classifying malware into families, we will be using the Microsoft Malware Classification Challenge (BIG 2015) dataset [1]. The original dataset consists of 10,867 labeled malware samples. Our disassembled file parser were able to properly parse 10,260 of the samples. Hence, we will be using these in the following evaluations. The class distribution of these samples are shown in Table 4.1.

To compare our work with Adagoi [59], we will use a secondary dataset consisting of 1,113 benign android apps and 1,200 malicious android apps. The malicious samples are from the Android Malware Genome Project [67]. A colleague at our university provided us with the benign samples, downloaded from the Google Play Store.

4.4.2 Parameter Selection

The first two parameters of our algorithm that need to be configured experimentally are the length of instruction opcode n-grams, and the number of function (vertex) clusters. N-gram length determines how many instructions are used in each n-gram when representing local functions as

Table 4.1: Microsoft malware dataset class distribution

Malware Family Name	Number of Samples
Ramnit	1513
Lollipop	2470
Kelihos ver3	2936
Vundo	446
Simda	34
Tracur	294
Kelihos˙ver1	387
Obfuscator.ACY	1168
Gatak	1012

sets of n-grams; in other words it determines the n in n-gram. Cluster number determines the number of buckets functions are hashed into.

Figure 4.4 shows the classification accuracy results when using uni-gram, bi-gram, and tri-gram. In both the n-gram length and cluster number experiments, only a single base classifier was used. In the Figure 4.4, using uni-grams results in better accuracy than both bi-grams and trig-rams and that using bi-grams results in better accuracy than tri-gram. This result can be explained by going back to the distance function (we are trying approximate function similarity in our approach.) As discussed in Section 4.3.2, we are trying to approximate edit distance using Jaccard Index, which we in turn approximate using Minhash. When computing edit distance, the insertion and deletion operations work on uni-grams. So the longer the n-gram we use when approximating edit distance with Minhash, the less accurate our approximation.

When it comes to the number of function clusters, we expect the classification accuracy to increase as the number of function clusters increases. This expectation can be understood by considering the fact that by hashing function into smaller number of clusters, we increase the likelihood of dissimilar functions being hashed into the same cluster. Obviously, this in turn results in lower classification accuracy. The results in Figure 4.5 agree with our expectations. The experiments to determine the number of function clusters were carried out using uni-gram opcode sets to represent functions and clustering the functions (vertices) into the specified number of clusters before converting to vector representation.

The other parameters that need to be tuned experimentally are the number of base classifiers(K), and the number of Minhash functions used for computing the Minhash Signature

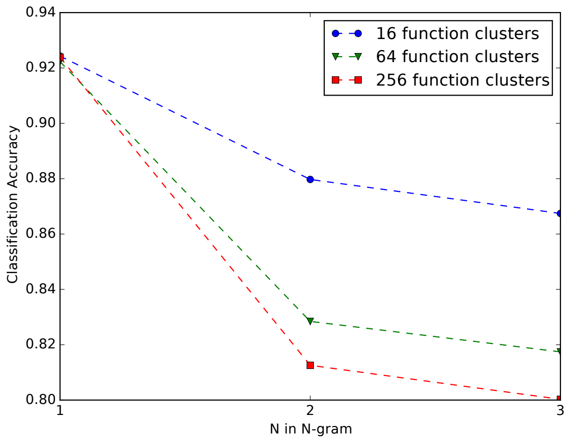


Figure 4.4: Effect of n-gram length on classification accuracy

(L). In our experiments, we evaluated various values of K. We observed that in the case the current evaluation dataset increasing the value of K above 6 didn't result in much improvement. As discussed in Section 4.3.6, using different values for L for the K different runs, results in more variation among the K base classifiers and results in better meta-classifier performance. In our experiments we evaluated a couple of L value combinations and picked the one that resulted in better performance.

4.4.3 Classification Accuracy Comparison

We evaluate our approaches accuracy by comparing it with our implementation of Kinable et.al. [6, 23] work on FCG classification. In [6], they use an approximate GED to measure the similarity between FCGs. GED distance is approximated using Simulated Annealing (SA) to find a bipartite graph mapping between the vertices of the two graphs that minimizes the GED. Since this is a computationally intensive task, functions in the two graphs that have instruction edit distance greater than some threshold τ are filtered out before creating the bipartite graph mapping. Then the FCGs are clustered based on GED.

Since using SA to approximate GED was computationally intensive, we evaluated both our proposed method and our implementation of Kinable et.al. work on a smaller sub set of the dataset consisting of 1000 malware samples. In our implementation of Kinable et.al. research,

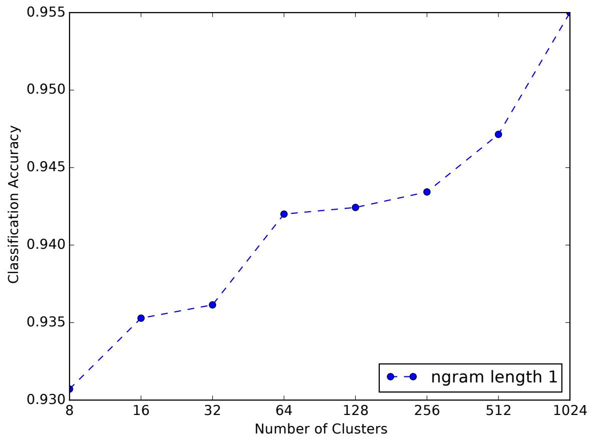


Figure 4.5: Effect of number of function clusters on classification accuracy

we filtered out most similar functions between pairs of graphs by setting the filter threshold τ to 0.9 and then created bipartite mapping on the remaining functions (vertices). We cluster the malware samples using k-medoids and then assigned a class label to each cluster based on the majority class. Hence, we determined every element of a cluster whose class label is different from the majority class as being misclassified. Figure 4.6 shows the confusion matrix of classifying the thousand samples in this way.

Next we evaluated our approach on the same 1000-malware sample with 10-fold cross validation to predict the labels on the thousand samples. Our system is set-up using six base classifiers(i.e. $K = 6$), which means that there are six parallel pipelines consisting of the Function clustering, Call graph vector extraction, and Base classifier modules. In all the of Function clustering modules, we represented functions (vertices) with uni-gram opcode sequence and clustered them into 64 clusters. To make the base classifiers more decorrelated, the Minhash signatures are generated using varying number of hash functions (i.e. value of L in Algorithm 4.1). In first two of the six pipelines $L = 1$, in the next two $L = 3$ and in the last two $L = 5$.

As can be seen from the results in Figures 4.6 and 4.7, our approach clearly out performs the SA based method with an overall accuracy of 0.979 versus the 0.840 of the previous works on the smaller dataset. Our system has high accuracy for almost all of the malware families, apart from the malware family Simda. The reason behind the poor accuracy in the case of Simda is

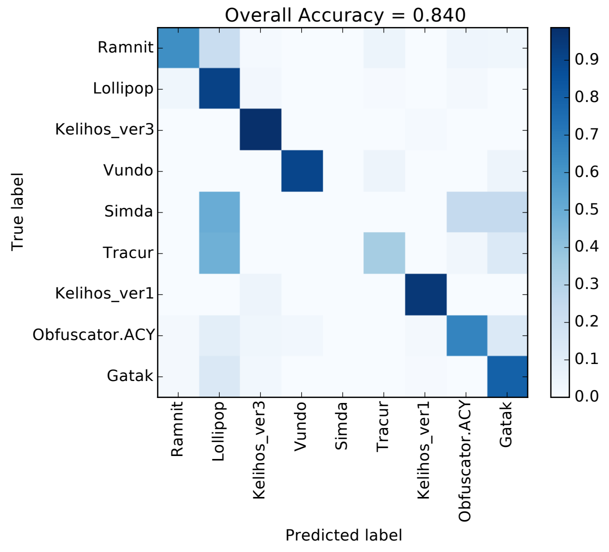


Figure 4.6: Confusion matrix for using Simulated Annealing on 1000-sample dataset

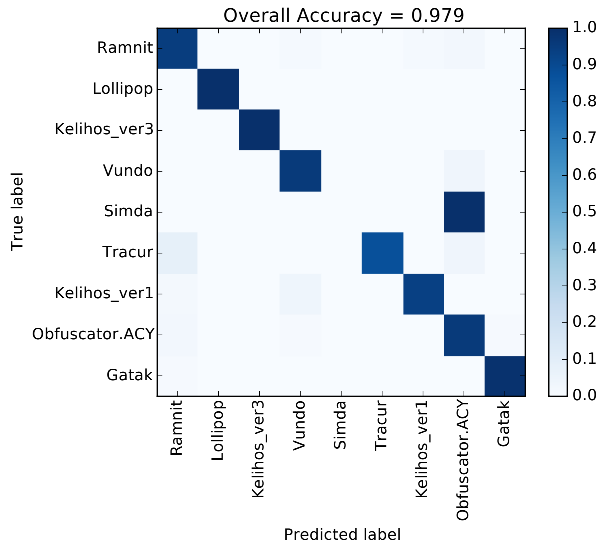


Figure 4.7: Confusion matrix using meta-classifier on 1000-sample dataset

the very small number of training samples. In the entire evaluation dataset there are only 34 instances, and in the case of the sampled 1000 instance dataset used in this section there were only 4 samples.

Figure 4.8 shows the classification accuracy of our approach when applied on the entire dataset using a 10-fold cross validation. Again we see that our system gets near perfect classification accuracy for all families except Simda. Which is again due to the smaller number of training samples.

4.4.4 Speed Comparison

We will now empirically compare the speed of our approach with that of SA based approach. For our approach, we timed the execution in the components starting with Call Graph Extraction to Call graph Vector representation (inclusive). We excluded the time taken by the learning algorithm. For the SA based approach we timed the process starting with Call Graph Extraction up to the GED computation between all pairs of malware samples, using SA. Similarly here, we excluded the time taken by the learning algorithm.

Table 4.2: Speed comparison

Approach	Average Time (min)
All pair similarity SA and GED	2006
Graph Vector Representation	7

The results in Table 4.2 show these time measurements, averaged over three runs, on a smaller subset of the evaluation data set consisting of 1000 samples. As we expected, our approach is significantly faster than the SA based technique making FCG classification scalable. These experiments were carried out on a machine with 2.3 GHz quad code CPU with 8 GB memory.

Time Complexity Comparison: To compare the training time complexity of these two approaches we will only consider the feature extraction phase in both methods and exclude the machine learning algorithms.

For the Simulated Annealing based approach, we consider the time complexity of graph similarity computation and exclude the clustering part when looking at the time complexity. As shown in [23], the time complexity of running Simulated Annealing to approximate Graph Edit Distance between two graphs is $O(|V_{max}|^2 \cdot d_{max})$, where $|V_{max}|$ is the number of vertices of the

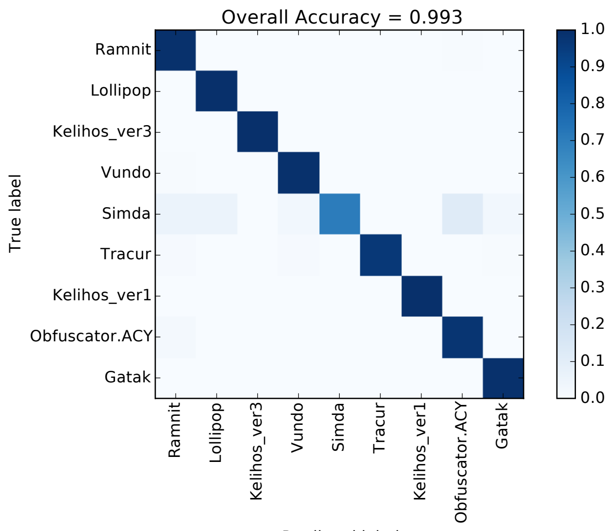


Figure 4.8: Confusion matrix using meta-classifier on entire dataset

largest graph, and d_{max} is the maximum value of degree for any node. For N graphs then $O(N^2)$ distance (similarity) computations are going to be required. Therefore, for N graphs the worst case time complexity becomes $O(N^2 \cdot |V_{max}|^2 \cdot d_{max})$.

In case of our system we consider time complexity of the Function clustering and Vector extraction modules. When we look at the Function clustering module, it is clear from Algorithm 4.1 that given an input graph $G = (V, E)$, it has time complexity of $O(|V|)$. For N graphs then the worst case time complexity becomes $O(N \cdot |V_{max}|)$, where $|V_{max}|$ is the number of vertices of the largest graph in the training dataset. For the Vector extraction module, we visit each vertex and each edge only once. Hence, the complexity of the module for a given input graph $G = (V, E)$ is $O(|V| + |E|)$. For N graphs then, the worst case time complexity becomes $O(N \cdot (|V_{max}| + |E_{max}|))$. Finally, the total worst case time complexity of the two modules together for N graphs is $O(N \cdot (|V_{max}| + |E_{max}|))$. So it can be clearly seen here that our approach compares favorably to the one based on Simulated Annealing.

4.4.5 Classifying benign and malware applications

In this section we will compare our approach with another closely related work, Adagoi [59]. We used the authors implementation of Adagoi [68], which performs classification between benign and malicious Android apps. To work with android apk files, we modified the implementation of the FCG Extraction module, in our system.

The evaluation was performed by randomly splinting the initial dataset to use 80% as training data and the remaining as test data. This was repeated 10 times, and the resulting average ROC curve is shown in Figure 4.9. As seen in the figure, our approach performs better than Adagoi. For instance, at 0.01 false positive rate, the area under the curve for our approach is 0.0099 where as for Adagoi it is 0.0092.

4.4.6 Combining with Non-graph Features

The former FCG based techniques, which relied solely on graph representation and on measuring graph similarity, were not easy to use together with other non-graph based features. Our proposed approach, on the other hand, extracts feature vectors from FCGs that can be easily used with other features by simply concatenating the feature vectors.

To demonstrate this, we use binary byte bi-gram frequency features. These features are extracted by computing the frequency of byte bi-grams in malware sample raw binary files. We take these feature vectors and concatenate a random subset of them with the graph vectors before passing them to the base classifiers. The reason for taking a random subset of these features is to help make the base classifiers more decorrelated.

Combining these features with the graph features, our overall classification accuracy slightly increased to 0.9933 from 0.993. This can be explained by the fact that both the graph and the n-gram frequency features are have overlaps in that they both are extracted based on the content of malware binary. As a future work we would like to examine combining our FCG features with the various features used in [61].

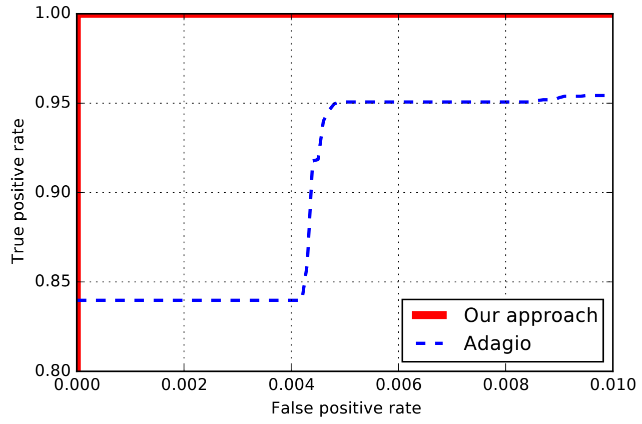


Figure 4.9: ROC curve

4.5 Conclusion

In this chapter we presented a fast and effective malware classification system based on extracting feature vector from FCG representation. Our approach was able to address two bottlenecks in previous malware classification systems that were based on FCG features. First, we were able to speed up the process of measuring similarity between functions using Minhash, an Locality Sensitive Hashing technique. Second, we avoided the major bottleneck of computing graph similarity by converting the graph representation into vector representation using function clustering based on the Minhash signatures of the functions.

Chapter 5

Learning to Identify Known and Unknown Classes: A Case Study in Open Set Malware Classification

5.1 Introduction

In a classification problem, we are given a set of classes between which the model has to learn to discriminate. There are many cases in which we know the possible classes in the problem domain beforehand and simply learning to distinguish between these classes is sufficient. This can be thought of as the "closed set" scenario. In other problem domains, however, we are aware of only some number of classes during training and instances that belong to classes not present in the training set can be present in a test set. In this chapter we will refer to this as the "open set" scenario.

This is especially true in the domain of malware family classification, in which we want to identify the family of a malware sample. There are a variety of reasons why this is the case. One reason is that it's not possible to collect all samples from every existing malware class/family for training. Even if we were able to do so, because of the adversarial nature of this problem domain, malware authors constantly release new malware families. Therefore, it is important to have a classifier that is not only capable of discriminating between instances from known malware families but one that is also able to determine if samples do not belong to any of the known families and label them as *unknown*. In this chapter we present a classifier system that achieves this goal. Our proposed approaches combine a classifier and an outlier detector to build a system where the outlier detector is trained on new features extracted from the output of the classifier. The following points summarize our main contributions:

- We present a different look from the majority of the research in the malware classification domain by addressing classification in an open set.
- We propose the extraction of new features from classifier output which transforms the problem of identifying unknown class instances into a new feature space.
- We provide an approach that is more accurate and scales better than previous open set classification approaches on the evaluation datasets.

5.2 Related Works

To the best of our knowledge there are very few works in the malware classification domain that focus on having the capability to operate in an open set scenario. One such work in the malware domain comes from K. Rieck et al. [18] in which malware samples are clustered into families and then the distance of a test instance to the closest cluster centroids is used as an outlier score. The limitation of this approach comes from the use of unsupervised methods for identifying between the known families which does not take advantage of label information that might be present for the known classes.

Open set classification has aspects similar to anomaly detection problems. Anomaly detection is a well researched area and has found application in the security realm such as intrusion detection. For example to mention a few, Ramaswamy et al. [69] propose a distance based approach which looks at the distance of an instance from its k^{th} nearest neighbor to identify outliers. Breunig et al. [70] propose a density based approach that looks at the local density of a point to determine its anomaly score. Many more approaches have been proposed for this problem; we direct the reader to [44, 71, 72] for a more detailed survey of the different anomaly detection techniques. The similarity of anomaly detection to open set problem is that in both cases there is an interest in identifying outlier samples. However, there are two main differences. First, the learning task in anomaly detection is in only detecting anomalous samples and not learning to discriminate the normal samples into different classes. Second, the assumption which is taken in anomaly detection is that outlier instances are rare [44]. This assumption, however, does not hold in the case of the open set problem.

Scheirer et al. [73] propose a modification to one-class SVM, called 1-vs-set SVM, to handle a one class open set problem for image recognition. Multi-class anomaly detection without classification called Kernel Null Foley-Sammon Transform (KNFST) proposed in [74] shares one of the objectives of the open set problem, which is to identify instances that do not belong to any of the known classes. Bodesheim et al. [75] build on this idea by taking a local approach where they only considering k closest elements to an instance when deciding the novelty on a test instances. In [76], Jain et al. present a multi-class open set classifier framework PI_SVM for fitting a single-class probability model over the known class scores from a discriminative binary classifiers.

5.3 Approach

The prediction of a classifier can be obtained in the form of predicted class probabilities $Pr(y_i = c \mid \vec{x_i})$, where $c \in C^k$ is a class label among the known classes in a training set, $\vec{x_i}$ is the feature vector of instance i, and y_i is the class label of instance i. Each of these probabilities can be interpreted as the confidence of the classifier labeling an instance as belonging to that class.

Our approaches are based on the intuition that class probabilities predicted by a classifier can help distinguish unknown class instances from known class instances. We expect the classifier to be less confident when making a prediction about instances from an unknown class as compared to instances from the known class.

5.3.1 New Derived Features

We propose extracting features from the output of a multi-class classifier to help identify known class instances from unknown class instances. The first feature we extract is $P_{max} = \max_{c \in C^k} Pr(y_i = c \mid \vec{x_i})$. We aim to capture the confidence of the classifier by extracting the P_{max}.

In addition to P_{max} we also want to capture another perspective on the prediction confidence by observing how spread out the predicted class probabilities are. For example assume a classifier trained to classify between 3 classes makes prediction for instance A with probabilities of 0.8, 0.1 and 0.1 for classes c1, c2 and c3 respectively. Also assume that for instance B it makes prediction with probabilities 0.8, 0.2 and 0.0. Although their maximum prediction probabilities for both example instances are the same at 0.8, the fact that the classifier predicts A as possibly belonging to all three classes, albeit with low probabilities for class c2 and c3, gives more insight about the classifiers confidence on its prediction. We try to capture this by measuring the entropy of the predicted class probabilities. Entropy quantifies the diversity in the predicted class probabilities(in other words it tells us how unevenly distributed the predicted class probabilities are.) The entropy for probability distribution p over $|C^k|$ classes is defined as $entropy(p) = -\sum_j^{|C^k|} p_j \log p_j$.

Algorithm 5.1 outlines the procedure used to extract new features from predicted class probabilities. The algorithm starts by splits the training set into T segments (line 1). For each segment, first a classifier is trained on data D excluding the instances in D_t (line 4). Then, predictions are made on the instances in D_t. Afterwards, two features are extracted from the predicted class probabilities: the maximum predicted class probability P_{max} and the *entropy* of the predicted class probabilities. Finally, the extracted features X_{tmp} for each instance in D_t are added to X_{inlier}, which holds the representation of the training set in terms of the new features.

5.3.2 Classification in an Open Set (COW)

By considering different aspects of a classifiers output, such as P_{max} and *entropy*, we should be able to distinguish between instance belonging to known classes C^k from those belonging to unknown classes C^u. One of the challenges of trying to recognize instances from classes in C^u

Algorithm 5.1: Use T-fold cross validation to generate an outlier detector training set with the new features.

```
Input  :
        $D$: Training set consisting of feature matrix $\mathsf{X}$ and class labels $\mathsf{Y}$. The class
    labels are from the known class $C^k$.
Output:
        $X_{inlier}$: Training data represented in terms of new prediction probability based
    features.
1 Split $D$ into $T$ segments;
2 Initialize $X_{inlier}$ as empty;
3 for $t = 1$ to $T$ do
4     Train multi-class classifier model $M_{tmp}$ on $D$ without $D_t$;
5     $P_{inlier} \leftarrow$ Predict class probabilities for $D_t$ using $M_{tmp}$;
6     $X_{tmp} \leftarrow$ extract features from $P_{inlier}$;
7     $X_{inlier} \leftarrow X_{inlier} \cup X_{tmp}$;
8 end
9 return $X_{inlier}$;
```

comes from the fact that we can only get training data representing classes in C^k. In our case this means that during training we can get only the predicted class probabilities for instances from classes in C^k. To address this, we use an outlier detection method trained on predictions of the instances from C^k.

Algorithm 5.2: Training COW

```
Input  :
        $D$: Training set consisting of feature matrix $\mathsf{X}$ and class labels $\mathsf{Y}$. The class
    labels are from the known classes $C^k$.
Output:
        $M_{multi}$: Multi-class classifier model
        $M_{outlier}$: Outlier detection model
1 $X_{inlier} \leftarrow$ Algorithm1 ($D$) ;
2 Train outlier detector model $M_{outlier}$ on $X_{inlier}$;
3 Train multi-class classifier model $M_{multi}$ on $D$;
4 return $M_{multi}$ and $M_{outlier}$;
```

Our proposed approach uses a classifier and an outlier detector to build an open set classification system (COW). Algorithm 5.2 outlines the procedure for training COW. The algorithm starts by calling Algorithm 5.1 to extract the new features which will be used to train the outlier detector in COW. An outlier detector $M_{outlier}$ is then trained using X_{inlier} (line 2). Finally, a multi-class classifier is trained on the original dataset D, and the two models are returned (lines 3-4). During testing, the multi-class classifier M_{multi} is first used to make predictions about a test instance. Then the outlier detector $M_{outlier}$ is used to determine if the instance is an inlier in which case the predicted class label is used. Otherwise it is labeled *unknown*.

Per Class Classification in an Open Set (COW_PC)

COW trains one global outlier detection model for all the known classes. This, however, might face challenges in scenarios in which a classifier is not equally good in identifying all the known classes. In such a case the classifier might make predictions about certain classes with ease while struggling for other classes. So predictions made about the difficult classes might be confused for predictions of instances from an unknown class. In trying to address such a scenario we present a modified version of COW which we call COW_PC. COW_PC trains separate outlier detection models per each class in C^k. By doing so we aim to address the aforementioned challenges.

The training procedure for COW_PC, Algorithm 5.3, is a modified version of Algorithm 5.2. Similar to COW, new features are extracted (line 1) and a classifier M_{multi} is trained using the original dataset (line 8). The difference between the two approaches lies in training of the outlier detector (lines 2-6). In case of COW_PC separate outlier detection models, one for each class in C^k, are trained and then added to the list of outlier detectors.

During testing, similar to COW, the multi-class classifier M_{multi} is first used to make a prediction. In the case of COW_PC, if the instance is predicted as class $c \in C^k$ then the outlier detector for class c ($M_{outlier_c}$) is then used to determine if the instance is an inlier in which case the predicted class label is used. Otherwise it is labeled *unknown*.

```
Algorithm 5.3: Training COW_PC
Input :
        $D$: Training set consisting of feature matrix X and class labels Y. The class
    labels are from the known class $C^k$.
Output:
        $M_{multi}$: Multi-class classifier model
        $ListM_{outlier}$: List of outlier detection model
1 $X_{inlier} \leftarrow$ Algorithm1 ($D$) ;
2 Initialize $ListM_{outlier}$ as empty;
3 foreach class c in $C^k$ do
4     $X_{inlier\_c} \leftarrow X_{inlier}$ rows corresponding to instances correctly predicted as c;
5     Train outlier detector model $M_{outlier\_c}$ using $X_{inlier\_c}$;
6     Add $M_{outlier\_c}$ to $ListM_{outlier}$;
7 end
8 Train multi-class classifier model $M_{multi}$ on $D$;
9 return $M_{multi}$ and $ListM_{outlier}$;
```

5.4 Experimental Evaluation

5.4.1 Evaluation Datasets

We used two datasets for evaluating the proposed approaches. The first is the Microsoft Malware Classification Challenge (MS-Challenge) dataset [1]. This dataset contains 10,867 disassembled and labeled windows malware binaries from 9 malware families/classes. Our disassembled file parser was able to properly parse 10,260 of the samples used in the following experiments.

We also evaluate on the Android Malware Genome Project [67] dataset which contains different families of malicious Android apps. The original dataset contained some classes with very few samples. This presented a challenge when during the evaluation because of the way we set up the open set experiments results in too few training samples. For this reason, we use only those classes which have at least 40 samples. After removing the infrequent classes the Android dataset that consisted of 986 samples from 9 classes.

5.4.2 Simulating Open Set Scenario

To simulate an open set scenario we first take a dataset and randomly designate $|C^u|$ number of classes to constitute C^u and the remaining classes constitute C^k. We add all the instances belonging to classes in C^u to the test set. We randomly added 75% of instances belonging to classes in C^k to the training set and the remaining 25% to the test set.

5.4.3 Malware Features and Learning Algorithms

The malware features used to train our classifier models in these experiments are based the proposed features in Chapter 4. These are features extracted from the function call graph (FCG) of Windows and Android samples. During the extraction of these features, the functions in the FCG are first clustered using Locality Sensitive Hashing (LSH) and the resulting cluster ids are used to label the functions. Then a vector representation of the FCG is created. Even though in Chapter 4 we use a two-level classifier system with these features, we use a single classifier in our experiments here.

All the experiments involving COW and COW_PC were preformed with Random Forest (RF) [65] as the classifier model and KD-Tree based Kernel Density Estimation as the outlier detector.

5.4.4 Discriminating Known Class Instances from Unknown Class Instances

To evaluate the ability of our approach to discriminate known class instances from unknown class instances, we use Area Under Curve of the ROC as a measurement metric. We use the outlier score and compute the True Positive Rate and False Positive Rate. When generating these ROC figures, we treat identifying the known classes as positive class and identifying the unknown class as the negative class. We chose to present the results in this manner because the main objective of the system is in classifying malware. Therefore, doing so in an open set scenario involves first identifying if an instance is indeed from a known class.

We compare our approaches with two other previous approaches on open set recognition: PI_SVM [76,77] and KNSFT [74,78]. For the sake of completeness we also report the performance of using an outlier detector, trained on the original data input space rather than on the prediction probability features we recommend in this chapter, to identify unknown class instances. The results for this outlier detector are reported under the name "Original Input" in figures 5.1a and 5.1b. For PI_SVM and KNFST we use the original authors implementation.

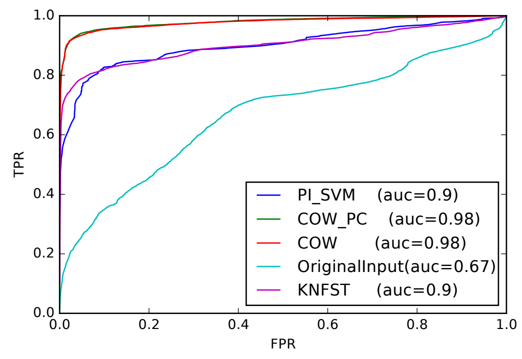
We performed 10 experiments for each approach. In each experiment the training and test sets are created to simulate an open set scenario by setting the number of unknown classes $|C^u|$ to be 3. We then use the learned model to generate an outlier score for each test instance to indicate the degree to which the model believes the instance does not belong to any of the classes in C^k. Figure 5.1a shows the result of these experiments carried out on the MS-Challenge dataset in the form of the average ROC. Figure 5.1b, on the other hand, shows the result of similar experiments on the Android Malware Genome Project dataset.

We would like to highlight two observations from the results in Figures 5.1a and 5.1b. First, our proposed approaches perform better compared to PI_SVM and KNFST. For example on the MS-Challenge dataset, at a 10% false positive rate (i.e. where 10% of instances from unknown classes get predicted as belonging to one of the known classes) our two approaches COW and COW_PC achieve a true positive rate (i.e. percentage of instances from known classes that are detected as known) of 95.36% and 95.61% respectively. This results in an area under the curve(AUC) up to 10% FPR of 0.0917 and 0.0923. At the same point the PI_SVM and KNFST achieve TPR of 82.76% 81.89%, respectively.

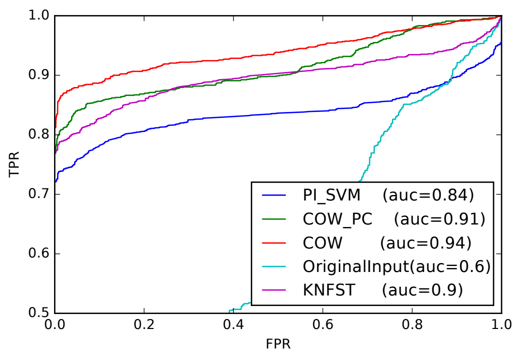
Second, our approach achieves a relatively high TPR even at 0% FPR. Table 5.1 shows, for instance, in case of MS-Challenge dataset our approach COW_PC achieves 69.81% TPR compared to PI_SVM's 23.97%. Similarly, for the Android dataset COW achieves 81.21% TPR compared to 77.25% TPR of KNFST.

Table 5.1: TPR at a very low FPR of 0% for all four methods on both datasets.

	TPR at 0% FPR	
	MS-Challenge Dataset	Android dataset
COW	42.41%	81.21%
COW_PC	69.81%	76.90%
PI_SVM	23.97%	72.11%
KNFST	9.94%	77.25%



(a) Average ROC up to 100% FPR on MS Dataset



(b) Average ROC up to 100% FPR on Andriod Dataset

Figure 5.1: Average ROC from 10 runs for distinguishing between instances from known and unknown classes.

5.4.5 Discriminating Among Known Classes

The results presented so far show how well the different approaches perform on the task of distinguishing between known class instances and unknown class instances. In addition to this, an open set classifier also needs to be able to discriminate between the known classes.

To evaluate this we run experiments on training and test sets created to simulate an open set scenario. In each experiment we set the number of unknown classes $|C^u|$ to be 3, which means $|C^k|$ will be 6 for both datasets. To make sure that all possible $\binom{|C^k|+|C^u|}{|C^u|} = \binom{9}{3}$ combinations of classes are used in C^u, 84 such experiments are performed. Hence, each class in the two datasets gets to be in the known class set exactly 56 times. In each experiment we recorded the weighted average precision, recall and f-score values of each class in the known class. This is obtained by first calculating the average of these three metrics for each known class. Then further calculating the average across all the known classes while weighting the values of these metrics by the fraction of the size of each known class in the test sets.

Since KNFST does not have the capability to discriminate between the known classes, we report results for COW, COW_PC, and PI_SVM Table 5.2. All three algorithms have one hyperparameter that specifies the threshold for discriminating between known and unknown class instances. We propose using a validation set to perform binary search over the hyperparameter space using f-score as the search metrics.

Table 5.2: Weighted average precision, recall and f-score.

	MS-Challenge Dataset		
	Precision	Recall	F-score
COW	0.94	0.90	0.91
COW_PC	0.92	0.90	0.90
PI_SVM	0.94	0.80	0.85
	Android Dataset		
	Precision	Recall	F-score
COW	0.95	0.86	0.89
COW_PC	0.93	0.84	0.87
PI_SVM	1	0.66	0.78

5.4.6 Efficiency Comparison with other Approaches

To compare the time efficiency of our approaches with PI_SVM and KNFST, we record the training and test times when running the experiments. Table 5.3 presents the average training and test time of 10 runs on MS and android datasets. These experiments were carried out on a machine with an Intel-i7 2.60GHz processor and 20GB RAM. As the results in Table 5.3 show, our two approaches, COW and COW_PC, seem to scale better compared to the other two algorithms.

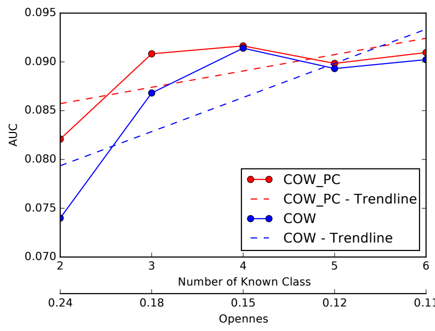
5.4.7 Effect of Number of Known Classes

Another interesting phenomenon to study is how our approaches perform as the number of known classes varies while keeping the number of unknown classes constant. The purpose of these experiments is to try to understand whether gathering more malware families for training can help improve performance of the open set classifiers.

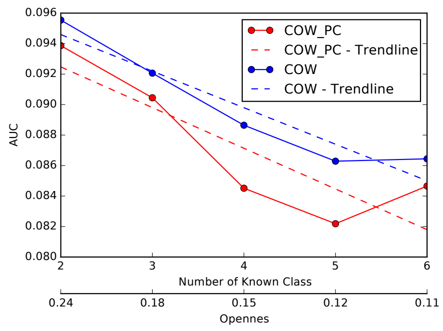
We set up the experiment in the following manner for both of the evaluation datasets. First, we select the original test and training data to simulate an open set scenario. Afterwards, we chose t number of classes at random from the known classes C^k in the training set, and create a new training set from the previously created training set that contains instances from the t selected classes. As for the test dataset we select all the instances from the unknown class that are in the previously created test set together with the instances from the t selected known classes. We then train a model on the new training set and evaluate it on the new test set. We record the performance of the model in terms of the AUC up to a FPR of 10%.

Table 5.3: Comparing the training and test times. The MS dataset has average training and test size of 5151 and 5110, respectively. The android dataset has training size of 480 and test size of 506 on average.

Algorithm	MS Dataset Average Time (sec)		Android Dataset Average Time (sec)	
	Training	Test	Training	Test
COW	9.15	2.11	4.11	0.24
COW_PC	7.42	1.42	4.06	0.41
PI_SVM	79.06	34.87	3.41	1.90
KNFST	577.50	202.10	3.96	2.21



(a) Average area under ROC up to 10% FPR on MS-Challenge Dataset



(b) Average area under ROC up to 10% FPR on Android dataset

Figure 5.2: The relationship between the performance of openset classifier and number of known classes.

Another way to understand these experiments is in terms of the percentage of opennes defined by [73], Equation 5.1. For a closed set scenario where the same classes are seen both during training and testing, we get an openness value of 0. On the other hand a higher openness value indicates a more open problem. In the context of these experiments, increasing the number of known classes during training while keeping the number of unknown classes seen during testing constant will result in decreasing openness. We expect that this decrease in openness should in turn result in improved performance. This expectation aligns with what Jain et al. [76] show in their experiments.

$$openness = 1 - \sqrt{\frac{2 \times |\, training\ classes \,|}{|\, test\ classes \,| + |\, target\ classes \,|}} \tag{5.1}$$

The result for MS-Challenge dataset indeed agrees with the expectation that as the openness decreases (i.e. number of known classes increases) the performance of our approach improves,

Figure 5.2a. The more surprising result comes in the case of the Android dataset, Figure 5.2b. In this case we see that performance actually degrades as openness decreases (i.e. as number of known classes increases).

We hypothesize that the unexpected results have to do with the number of instances in each class. When we compare the class distributions of the MS-challenge dataset with the Android dataset, we see that the number of instances per class in the android dataset is considerably smaller. Generally, the classification task becomes more difficult as the number of classes to classify increases and also as the number of training instances gets smaller. Since our approach depends on the predicted class probabilities generated by the classifier, we believe the smaller size of the Android dataset has resulted in decreased performance as the number of classes increases (openness decreases). This hypothesis also helps explain why our approach records lower AUC in case of the android dataset as presented in previous Sections.

We evaluated this hypothesis by down-sampling the MS-Challenge dataset to have a comparable number of instances for each class with the Android dataset. The results in Figure 5.3 show the average AUC up to a 10% FPR of 10 runs. We observe that the AUC generally decreases as the number of known class increase (i.e. as openness decreases).This result is consistent with our hypothesis that smaller number of instances in each class can degrade performance. This phenomenon needs to be studied further with more datasets. The implications of this result is that knowing more classes is not enough but sufficient amount of data samples for each class is also needed.

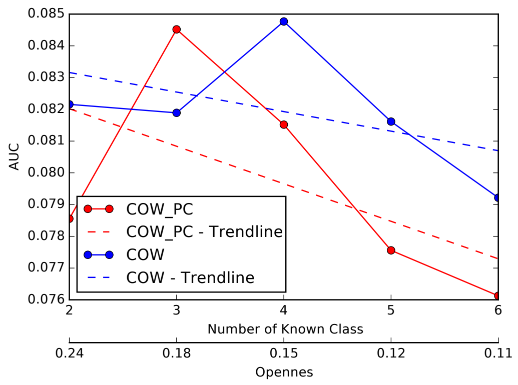


Figure 5.3: Experiment carried out on an down-sampled MS-Challenge dataset so as to have a comparable number of per class instances with the android dataset.

5.5 Conclusion

In this chapter we presented an open set classification approach used for malware family classification. The approach uses features extracted from the predicted class probabilities received from a classifier to train an outlier detector. The classifier and the outlier detector together form a system that is not only capable of distinguishing between known classes but is also capable of identifying instances arising from unknown (never before seen) classes. The evaluation results show that our approach compares favorably in accuracy with previous works on open set classification and multi-class outlier detection techniques. The evaluation also shows that our approach takes less time for both training and test.

Chapter 6

Learning a Neural-network-based Representation for Open Set Recognition

6.1 Introduction

To build robust AI systems, Dietterich [79] reasons that one of the main challenges is handling the "unknown unknowns." One idea for handling the unknown unknowns is to detect model failures; that is, the system understands that its model about the world/domain has limitations and may fail. For example, assume you trained a binary classifier model to discriminate between pictures of cats and dogs. You deploy this model and observe that it does a very good job at recognizing images of cats and dogs. What would this model do if it is faced with a picture of a fox or a caracal (mid sized African wild cat)? The model being a binary classifier will predict these pictures to be either a dog or a cat, which is not desirable and can be considered as a failure of the model. In machine learning, one direction of research for detecting model failure is "open category learning", where not all categories are known during training and the system needs to appropriately handle instances from novel/unknown categories that may appear during testing. Besides "open category learning", terms such as "open world recognition" [80] and "open set recognition" [73, 81] have been used in past literatures. In this chapter we will use the term "open set recognition".

Where does open set recognition appear in real world problems? There are various real world applications that operate in an open set scenario. For example, Ortiz and Becker [82] point to the problem of face recognition. One such use case is automatic labeling of friends in social media posts, "where the system must determine if the query face exists in the known gallery, and, if so, the most probable identity." Another domain is in malware classification, where training data usually is incomplete because of novel malware families/classes that emerge regularly. As a result, malware classification systems operate in an open set scenario.

In this chapter we propose a neural network based representation and a mechanism that utilizes this representation for performing open set recognition. Since our main motivation when developing this approach was the malware classification domain, we evaluate our work on two malware datasets. To show the applicability of our approach to domains outside malware, we also evaluate our approach on images.

Our contributions include: (1) we propose an approach for learning a representation that facilitates open set recognition, (2) we propose a loss function that enables us to use the same distance function both when training and when computing an outlier score, (3) our proposed approaches achieve statistically significant improvement compare to previous research work on three datasets.

The remainder of this chapter is organized in the following manner. In Section 6.2, we give an overview of related works and present our approach in Section 6.3. We present our evaluation methodology, results, and provide further discussions in Section 6.4.

6.2 Related Work

We can broadly categorize existing open set recognition systems into two types. The first type provides mechanisms to discriminate known class instances from unknown class instances. These systems, however, cannot discriminate between the known classes, where there is more than one. Research works such as [73–75] fall in this category. Scheirer et al. [73] formalized the concept of open set recognition and proposed a 1-vs-set binary SVM based approach. Bodesheim et al. [74] propose KNFST for performing open set recognition for multiple known classes at the same time. The idea of KNFST is further extended in [75] by considering the locality of a sample when calculating its outlier score.

The second type of open set recognition system, provides the ability to discriminate between known classes in addition to identifying unknown class instances. Research works such as [80, 81, 83–85] fall in this category. PI-SVM [83], for instance, uses a collection of binary SVM classifiers, one for each class, and fits a Weibull distribution over the score each classifier. This approach allows PI-SVM to be able to both perform recognition of unknown class instances and classification between the known class instances. Bendale and Boult [80] propose an approach to extend Nearest Class Mean (NCM) to perform openset recognition with the added benefit of being able to do incremental learning.

Neural Net based methods for open set recognition have been proposed in [81, 85, 86]. Openmax [81] (a state-of-art algorithm) modifies the normal Softmax layer of a neural network by redistributing the activation vector (i.e. the values of the final layer of a neural network that are given as input to the Softmax function) to account for unknown classes. A Neural Net is first trained with the normal Softmax layer to minimize cross entropy loss. The activation vector of each training instance is then computed; and using these activation vectors the per-class mean of the activation vector (MAV) is calculated. Then each training instance's distance from its class MAV is computed and a separate Weibull distribution for each class is fit on certain number of the largest such distances. Finally, the activation vector's values are redistributed based on the probabilities from the Weibull distribution and the redistributed value is summed to represent the unknown class activation value. The class probabilities (now including the unknown class) are then calculated using Softmax on the new redistributed activation vector.

The challenge of using the distance from MAV is that the normal loss functions, such as cross entropy, do not directly incentivize projecting class instances around the MAV. In addition to that, because the distance function used during testing is not used during training it might not necessarily be the right distance function for that space. We address this limitation in our proposed approach.

Ge et al. [85] combine Openmax and GANs [87] for open set recognition. The network used for their approach is trained on the known class instances plus synthetic instances generated using the DCGAN [88]. In the malware classification domain, K. Rieck et al. [18] proposed a malware clustering approach and an associated outlier score. Although the authors did not propose their work for open set recognition, their outlier score can be used for unsupervised open set recognition. Rudd et al. [89] outline ways to extend existing closed set intrusion detection approaches for open set scenarios.

6.3 Approach

For open set recognition, given a set of instances belonging to known classes, we would like to learn a function that can accurately classify an unseen instance to one of the known classes or an unknown class. Let D be a set of instances X and their respective class labels Y (ie, $D = (X, Y)$), and K be the number of unique known class labels. Given D for training, the problem of open set recognition is to learn a function f that can accurately classify an unseen instance (not in X) to one of the K classes or an unknown class (or the "none of the above" class).

The problem of open set recognition differs from the problem of closed set ("regular") classification because the learned function f needs to handle unseen instances that might belong to classes that are not known during training. That is, the learner is robust in handling instances of classes that are not known. This difference is the main challenge for open set recognition. Another challenge is how to learn a more effective instance representation that facilitates open set recognition than the original instance representation used in X.

6.3.1 Overview

Consider $\vec{x}$ is an instance and $y = f(\vec{x})$ is the class label predicted using $f(\vec{x})$. In case of a closed set, y is one of the known class labels. In the case of open set, y could be one of the known classes or an unknown class. The hidden layers in a neural network, $\vec{z} = g(\vec{x})$, can be considered as different representations of $\vec{x}$. Note, we can rewrite y in terms of the hidden layer as $y = f(\vec{z}) = f(g(\vec{x}))$.

The objective of our approach is to learn a representation that facilitates open set recognition. We would like this new representation to have two properties: (P1) instances of the same class are closer together and (P2) instances of different classes are further apart. The two properties can lead to larger spaces among known classes for instances of unknown classes to occupy. Consequently, instances of unknown classes could be more effectively detected.

This representation is similar in spirit to a Fisher Discriminant. A Fisher discriminant aims to find a linear projection that maximizes between class (inter class) separation while minimizing within class (intra class) spread. Such a projection is obtained by maximizing the Fisher criteria. However, in the case of this work, we use a neural network with a *non-linear* projection to learn this representation.

For closed set classification, the hidden layer representations are learned to minimize classification loss of the output y using loss functions such as cross entropy. However, the representations might not have the two desirable properties we presented earlier, useful in open set recognition. In this chapter we present an approach in Section 6.3.2 for learning the hidden layer $\vec{z} = g(\vec{x})$ using a loss function, that directly tries to achieve the two properties.

Once such representation is learned, we can use the distance of $\vec{z}$ from a class center as an outlier score (i.e. further away $\vec{z}$ is from the closest class center the more likely it is to be from unknown class/outlier.) This is discussed in more detail in Section 6.3.3. By observing how close training instances are to the class center, we estimate a threshold value for the outlier score for identifying unknown class instances (Section 6.3.3). We can further use the distance of $\vec{z}$ from all the known class centers to predict a probability distribution over all the known classes (Section 6.3.4).

To make the final open set prediction, we use the threshold on the outlier score to first identify known class instances from unknown class instances. Then, for the instances that are predicted as known, we use the predicted probability distribution over the known classes to identify the most likely known class label (Section 6.3.5).

6.3.2 Learning representations

Recall that to learn a new representation $\vec{z}$ for an original instance $\vec{x}$, we learn a function g that projects $\vec{x}$ to $\vec{z}$ (ie, $\vec{z} = g(\vec{x})$). The learning process is guided by a loss function that satisfies properties P1 and P2 in Section 6.3.1.

II-Loss Function

In a typical neural network classifier, the activation vector that comes from the final linear layer are given as input to a Softmax function. Then the network is trained to minimize a loss function such as cross entropy on the outputs of the Softmax layer. In our case the output vector $\vec{z_i}$ of the final linear layer (i.e activation vector that serves as input to a softmax in a typical neural net) are considered as the projection of the input vector $\vec{x_i}$, of instance i, to a different space in which we aim to maximize the distance between different classes (inter class separation) and minimize distance of an instance from its class mean (intra class spread). We measure intra class spread as the average distance of instances from their class means:

$$intra\_spread = \frac{1}{N}\sum_{j=1}^{K}\sum_{i=1}^{|C_j|}\|\vec{\mu_j} - \vec{z_i}\|_2^2 \tag{6.1}$$

where $|C_j|$ is the number of training instances in class C_j, N is the number of training instances, and μ_j is the mean of class C_j :

$$\vec{\mu_j} = \frac{1}{|C_j|}\sum_{i=1}^{|C_j|}\vec{z_i} \tag{6.2}$$

We measure the inter class separation in terms of the distance between the closest two class means among all the K known classes:

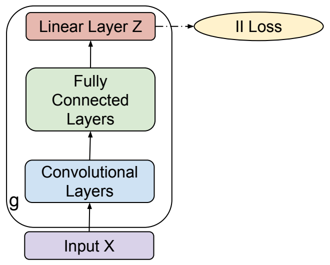
$$inter\_sparation = \min_{\substack{1 \le m \le K \\ m+1 \le n \le K}} \|\vec{\mu_m} - \vec{\mu_n}\|_2^2 \tag{6.3}$$

An alternative for measuring inter class separation would have been to take the average distances between all class means. Doing so, however, allows the largest distance between two classes to dominate inter class separation, hence does not result in a good separation.

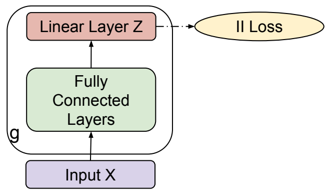
The network is then trained using mini-batch stochastic gradient descent with backpropagation as outlined in Algorithm 6.1 to minimize the loss function in Equation 6.4, which we will refer to ii-loss for the remainder of this chapter. This loss function minimizes the intra class spread and maximizes inter class separation.

$$ii\text{-}loss = intra\_spread - inter\_sparation \tag{6.4}$$

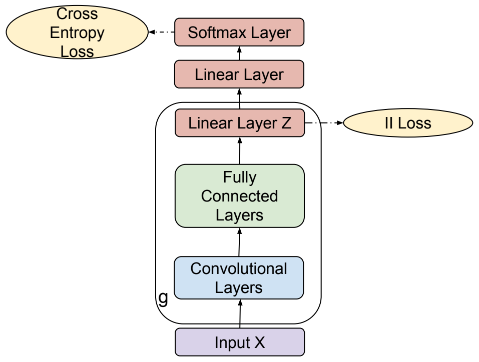
After the network finishes training, the class means are calculated for each class using all the training instances of that class and stored as part of the model.



(a) Convolutional network with ii-loss



(b) Fully connected network with ii-loss



(c) Combining ii-loss with Cross Entropy Loss

Figure 6.1: Network architecture with ii-loss.

The neural network g used to learn the representation can be either a combination of convolution and fully connected layers, as shown in Figure 6.1a, or it can be all fully connected layers, Figure 6.1b. Both types are used in our experimental evaluation.

Combining ii-loss with Cross Entropy Loss

While the two desirable properties P1 and P2 discussed in Section 6.3.1 aim to have a representation that separates instances from different classes, lower classification error is not explicitly stated. Hence, a third desirable property (P3) is a low classification error in the training data. To achieve this, alternatively a network can be trained on both cross entropy loss and ii-loss (Eq 6.4) simultaneously. The network architecture in Figure 6.1c can be used. In this configuration, an additional linear layer is added after the z-layer. The output of this linear layer is passed through a Softmax function to produce a distribution over the known classes. Although

Algorithm 6.1: Training to minimize ii-loss.

```
   Input  :
           $(X, Y)$: Training data and labels
1  for number of training iterations do
2      Sample a mini-batch $(X_{batch}, Y_{batch})$ from $(X, Y)$
3      $Z_{batch} \leftarrow g(X_{batch})$
4      $\{\vec{\mu}_1 \cdots \vec{\mu}_K\} \leftarrow class\_means(Z_{batch}, Y_{batch})$
5      $intra\_spread \leftarrow intra\_spread(Z_{batch}, \{\vec{\mu}_1 \cdots \vec{\mu}_K\})$
6      $inter\_separation \leftarrow inter\_separation(\{\vec{\mu}_1 \cdots \vec{\mu}_K\})$
7      ii-loss $\leftarrow intra\_spread$- $inter\_separation$
8      update parameters of $g$ using stochastic gradient descent to minimize ii-loss
9  end
10 $\{\vec{\mu}_1 \cdots \vec{\mu}_K\} \leftarrow class\_means(g(X), Y)$
11 return $\{\vec{\mu}_1 \cdots \vec{\mu}_K\}$ and parameters of $g$ as the model.
```

Figure 6.1c shows a network with a convolutional and fully connected layers, combining ii-loss with cross entropy can also work with a network of fully connected layers only.

The network is trained using mini-batch stochastic gradient descent with backpropagation. During each training iterations the network weights are first updated to minimize ii-loss and then updated to minimize cross entropy loss. Other researchers have trained neural networks using more than one loss function. For example, the encoder network of an Adversarial autoencoders [90] is updated both to minimize the reconstruction loss and the generators loss.

6.3.3 Outlier Score for Open Set Recognition

During testing we use an outlier score to indicate the degree to which the network predicts an instance $\vec{x}$ to be an outlier. This outlier score is calculated as the distance of an instance to the closest class mean from among K known classes.

$$outlier\_score(\vec{x}) = \min_{1 \leq j \leq K} \|\vec{\mu}_j - \vec{z}\|_2^2 \tag{6.5}$$

where $\vec{z} = g(\vec{x})$.

Because the network is trained to project the members of a class as close to the mean of the class as possible the further away the projection $\vec{z}$ of instance $\vec{x}$ is from the closest class mean, the more likely the instance is an outlier for that class.

Threshold Estimation

Once an outlier score is identified, the next step is to determine a threshold value for the outlier score. The threshold will indicate how far the projection of an instance needs to be from the closest class mean for it to be deemed an outlier. For this work, we propose a simple threshold estimation. To pick an outlier threshold, we assume that a certain percent of the training set to be noise/outliers. We refer to this percentage as the contamination ratio. For example, if we set the contamination ratio to be 0.01 it will be like assuming 1% of the training data to be noise/outliers. Then, we calculate the outlier score on the training set instances, sort the scores in ascending order, and pick the 99 percentile outlier score value as the outlier threshold value.

The reader might notice that the threshold proposed in this section is a global threshold. This means that the same outlier threshold value is used for all classes. An alternative to this approach is to estimate the outlier threshold per-class. However, in our evaluation we observe that global threshold consistently gives more accurate results than per-class threshold.

6.3.4 Prediction on Known Classes

Once a test instance is predicted as known, the next step is to identify to which of the known class it belongs. An instance is classified as belonging to the class whose class mean its projection is nearest to. If we want the predicted class probability over the known classes, we can take the softmax of the negative distance of a projection $\vec{z}$, of the test instance $\vec{x}$ (i.e. $\vec{z} = g(\vec{x})$), from all the known class means. Hence the predicted class probability for class j is given by:

$$P(y = j \mid \vec{x}) = \frac{e^{-\|\vec{\mu}_j - \vec{z}\|_2^2}}{\sum_{m=1}^{K} e^{-\|\vec{\mu}_m - \vec{z}\|_2^2}} \tag{6.6}$$

6.3.5 Performing Open Set Recognition

Open set recognition is a classification over $K+1$ class labels, where the first K labels are from the known classes the classifier is trained on. The $K+1$st label represents the *unknown* class that signifies that an instance does not belong to any of the known classes. This is performed using the outlier score in Equation 6.5 and the *threshold* estimated in Section 6.3.3. The *outlier_score* of a test instance is first calculated. If the score is greater than *threshold*, the test instance is labeled as $K+1$, which in our case corresponds to the *unknown class*, otherwise the appropriate class label is assigned to the instance from among the known classes:

$$y = \begin{cases} K+1, & \text{if } \mathit{outlier\_score} > \mathit{threshold} \\ \underset{1 \le j \le K}{\operatorname{argmax}} P(y=j \mid \vec{x}), & \text{otherwise} \end{cases} \tag{6.7}$$

When a network is trained on ii-loss alone, $P(y=k \mid \vec{x})$ in the above equation comes from Equation 6.6; whereas in case of a network trained on both ii-loss and cross entropy loss, discussed in Section 11, $P(y=k \mid \vec{x})$ is from the Softmax layer in Figure 6.1c.

6.4 Evaluation

6.4.1 Datasets

We evaluate our approach using three datasets. Two malware datasets Microsoft Malware Challenge Dataset [1] and Android Genome Project Dataset [67]. The Microsoft Dataset consists of disassembled windows malware samples from 9 malware families/classes. For our evaluations, we use 10260 samples which our disassembled file parser was able to correctly process. The Android dataset consists of malicious android apps from many families/classes. In our evaluation, however, we use only classes that have at least 40 samples so as to be able to split the dataset in to training, validation and test and have enough samples. After removing the smaller classes the dataset has 986 samples. To show that our approach can be applied in other domains we evaluate our work on the MNIST Dataset [91], which is a dataset consisting of images of hand written digits from 0 to 9.

We extract function call graph (FCG) from the malware samples as proposed in Chapter 4. In case of the Android dataset we first use [68] to extract functions and the function instructions from the APK files. We then extract the FCG features. For MS Challenge dataset, we reformat the FCG features as a graph adjacency matrix by taking the edge frequency features in Chapter 4 and rearranging them to form an adjacency matrix. Formating the features this way allowed us to use convolutional layers on the MS Challenge dataset.

6.4.2 Simulating Open Set Dataset

To simulate an open world dataset for our evaluation datasets, we randomly choose K number of classes from the dataset, which we will refer to as known classes in the remainder of this evaluation section, and keep only training instances from these classes in the training set. We will refer to the other classes as unknown classes. We use the open datasets created here in Sections 6.4.4 and 6.4.5.

In case of the MS Dataset and Android Dataset, first we randomly chose 6 known classes and treat set the remaining 3 as unknown classes. We then randomly select 75% of the instances from the known classes for the training set and the remaining for the test set. We further withhold one third of the test set to serve as a validation set for hyper parameter tuning. We use only the known class instances for tuning. In these two datasets all the unknown class instances are placed into the test set. In case of the MNIST dataset, first we randomly chose 6 known classes and the remaining 4 as unknown classes. We then remove the unknown class instances from the training set. We leave the test set, which has both known and unknown class instances, as it is.

For each of our evaluation datasets we create 3 open set datasets. We will refer to these open set datasets as OpenMNIST1, OpenMNIST2 and OpenMNIST3 for the three open set evaluation datasets created from MNIST. Similarly, we also create OpenMS1, OpenMS2, and OpenMS3 for MS Challenge dataset and OPenAndroid1, OpenAndroid2, and OpenAndroid3 for Android Genom Project dataset.

6.4.3 Evaluated Approaches

We evaluate four approaches, all implemented using Tensorflow. The first (*ii*) is a network setup to be trained using ii-loss. The second (*ii+ce*) is a network setup to be simultaneously trained using ii-loss and cross entropy (Section 11). The third (*ce*) is a network which we use to represent the baseline, is trained using cross entropy only (network setup in Figure 6.1c without the ii-loss.) The final approach is Openmax [81] (a state-of-art algorithm), which was reimplemented based the original paper and the authors' source code to fit our evaluation framework. The authors of Openmax state that the choice of distance function Euclidean or combined Euclidean and cosine distance give similar performance in the case of their evaluation datasets [81]. In our experiments, however, we observed that the combined Euclidean and cosine distance gives a much better performance. So we report the better result from combined Euclidean and cosine distance.

The networks used for MS and MNIST datasets have convolution layers at the beginning followed by fully connected layers, whereas for the android dataset we use only fully connected layers. The architecture is detailed in Appendix A. Our source code is available on github [1]. The evaluation datasets are available online on their respective websites.

6.4.4 Detecting Unknown Class Instances

We start our evaluation by showing how well *outlier_score* (Section 6.3.3) is able to identify unknown class instances. We evaluate it using 3 random open set datasets created from MS, Android and MNIST datasets as discussed in Section 6.4.2. For example, in the case of MNIST dataset, we run 10 experiments on OpenMNIST1, 10 experiments on OpenMNIST2, and 10 experiments on OpenMNIST3. We then report the average of the 30 runs. We do the same for the other two datasets.

Table 6.1 shows the results of this evaluation. To report the results in such a way that is independent of outlier threshold, we report the area under ROC curve. This area is calculated using the outlier score and computing the true positive rate (TPR) and the false positive rate (FPR) at different thresholds. We use t-test to measure statistical significance of the difference in AUC values. Looking at the AUC up to 100% FPR in all tree datasets, our approach *ii* and *ii+ce* perform significantly better(with p-value of 0.04 or less) in identifying unknown class instances than the baseline approach *ce* (using only cross entropy loss.) Although AUC up to 100% FPR gives as a full picture, in practice it is desirable to have good performance at lower false positive rates. That is why we report AUC up to 10% FPR. Our two approaches report a significantly better AUC than the baseline network trained to only minimize cross entropy loss. We didn't include Openmax in this section's evaluation because it doesn't have a specific outlier score.

Table 6.1: Average AUC of 30 runs up to 100% FPR and 10% FPR (the positive label represented instances from unknown classes and the negative label represented instances from the known classes when calculating the AUC). The underlined average AUC values are higher with statistical significance (p-value ¡ 0.05 with a t-test) compared to the values that are not underlined on the same row. The average AUC values in **bold** are the largest average AUC values in each row.

	FPR	ce	ii	ii+ce
MNIST	100%	0.9282 (±0.0179)	**0.9588** (±0.0140)	0.9475 (±0.0151)
	10%	0.0775 (±0.0044)	**0.0830** (±0.0045)	0.0801 (±0.0044)
MS Challenge	100%	0.9143 (±0.0433)	0.9387 (±0.0083)	**0.9407** (±0.0135)
	10%	0.0526 (±0.0091)	**0.0623** (±0.0030)	0.0596 (±0.0035)
Android Genom	100%	0.7755 (±0.1114)	0.8563 (±0.0941)	**0.9007** (±0.0426)
	10%	0.0066 (±0.0052)	0.0300 (±0.0193)	**0.0326** (±0.0182)

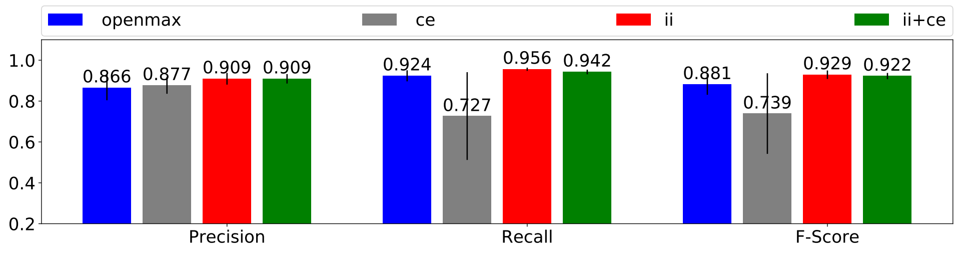
6.4.5 Open Set Recognition

When the proposed approach is used for open set recognition, the final prediction is a class label, which can be one of the K known class labels if the a test instances has an outlier score less than a threshold value (Section 6.3.3) or it can be an "*unknown*" label if the instance has an outlier score greater than the threshold. In addition to the three approaches evaluated in the previous section, we also include Openmax [81] in these evaluations because it gives final class label predictions.

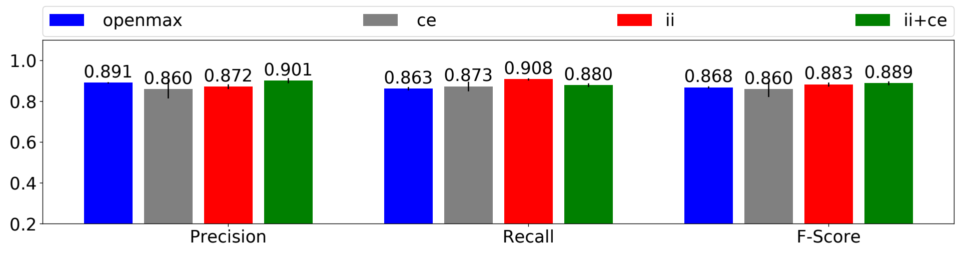
We use average precision, recall and f-score metrics to evaluate open set recognition performance and t-test for statistical significance. Precision, recall and f-score are first calculated for each of the K known classes labels and the one "*unknown*" label. Then the average overall the $K + 1$ classes is calculated. Using the same experimental setup as Section 6.4.4 (i.e. using 3 random open set datasets created from MS, Android and MNIST datasets as discussed in Section 6.4.2.), we report the result of the average precision, recall and f-score, averaged across all class labels and 30 experiment runs in Figure 6.2.

On all three datasets the *ii* and *ii+ce* networks gives significantly better f-score compared to the other two configurations (with p-value of 0.0002 or less). In case of the Android dataset, all networks perform lower compared to the other two datasets. We attribute this to the small number of samples in the Android datasets. The dataset is also imbalanced with a number of classes only having less than 60 samples.

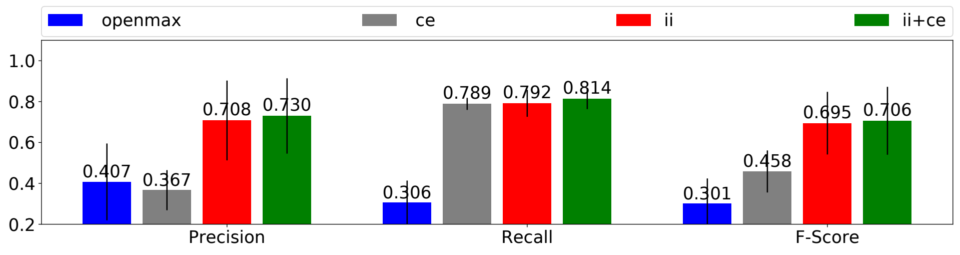
[1]https://github.com/shrtCKT/opennet



(a) MNIST Dataset with 6 known and 4 unknown classes.



(b) MS Challenge Dataset with 6 known and 3 unknown classes.



(c) Android Genom Dataset with 6 known and 3 unknown classes.

Figure 6.2: Average Precision, Recall and F-Score of 30 Runs. The metric calculations included the average of K known class labels and the one *unknown* class label further averaged over 30 experiment runs. The results are from Openmax, a baseline network trained on cross entropy loss only (*ce*), a network trained on ii-loss only (*ii*), and a network trained on combination of ii-loss and cross entropy (*ii+ce*).

6.4.6 Closed Set Classification

In this section we would like to show that on a closed dataset, a network trained using ii-loss performs comparably to the same network trained using cross entropy loss. For closed set classification, all the classes in the dataset are used for both training and test. For MS and Android datasets we randomly divide the datasets into training, validation, and test and report the results on the test set. The MNIST dataset is already divided into training, validation and test.

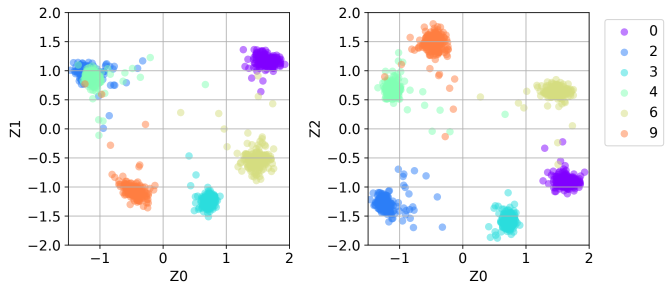
On closed MNIST dataset, a network trained with cross entropy achieved, a 10-run average classification accuracy of 99.42%. The same network trained using ii-loss achieved an average accuracy of 99.31%. The network trained only on cross entropy gives better performance than the network trained on ii-loss. We acknowledge that both results are not state-of-art as we are using simple network architectures. The main goal of these experiments is to show that the ii-loss trained network can give comparable results to a cross entropy trained network. On the Android dataset the network trained on a cross entropy gets an average classification accuracy of 93.10% while ii-loss records 92.68%, but the difference is not significant (with p-value at 0.43).

In Section 11, we proposed training a network on both ii-loss cross entropy loss in an effort to get lower classification error. The results from our experiments using such a network for closed MNIST dataset give an average classification accuracy of 99.40%. This result makes it comparable to the performance of the same network trained using cross entropy only (with p-value of 0.22).

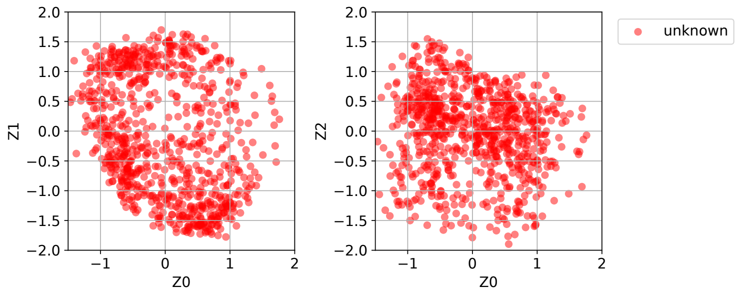
6.4.7 Discussions

As discussed in Section 6.2 the limitations of openmax are 1) it does not use a loss function that directly incentivizes projecting class instances around the mean class activation vector and 2) the distance function used by openmax is not necessarily the right distance function for final activation vector space. We addressed these limitations by training a neural network with a loss function that explicitly encourages the two properties in Section 6.3.1. In addition, we use the same distance function during training and test. As a result, we observe in Section 6.4.5 that our two proposed approaches perform better in open set recognition.

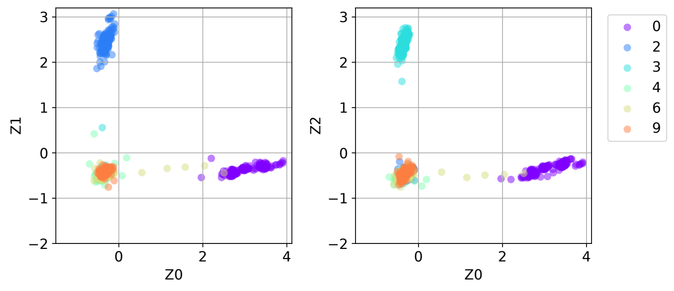
Figure 6.3 provides evidence on how our network projects unknown class instances in the space between the known classes. In the figure, the z-layer projection of 2000 random test instances of an open set dataset created from MNIST with 6 known and 4 unknown classes. The class labels 0, 2, 3, 4, 6, and 9 in the figure represent the 6 known classes while the “unknown” label represents all the unknown classes. The network with ii-loss is setup to have a z-layer dimension of 6, and the figure shows a 2D plot of dimension (z0,z1), (z0,z2). The Openmax network also has a similar network architecture and last layer dimension of 6. In case of ii-loss based projection, the instances from the known classes (Figures 6.3a) are projected close to their respective class



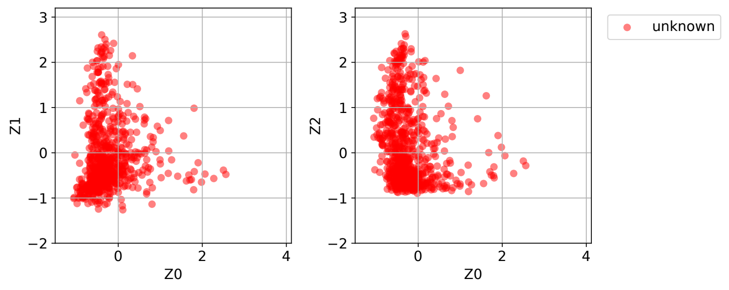
(a) ii-loss



(b) ii-loss



(c) openmax



(d) openmax

Figure 6.3: The z-layer projection of (a, c) known and (b, d) unknown class instances from test set of MNIST dataset. The labels 0,2,3,4,6,9 represent the known classes while the label "unknown" represents the unknown classes.

while the unknown class instances (Figures 6.3b) are projected, for the most part, in the region between the classes. It is this property that allows us to use the distance from the class mean as an outlier score to perform open set recognition. In case of openmax, Figures 6.3c and 6.3d, the unknown class instances do not fully occupy the open space between the known classes. In openmax, most instances are projected along the axis, this is because of the one-hot encoding induced by cross entropy loss. So compared to openmax, ii-loss appears to better utilize space "among" the classes.

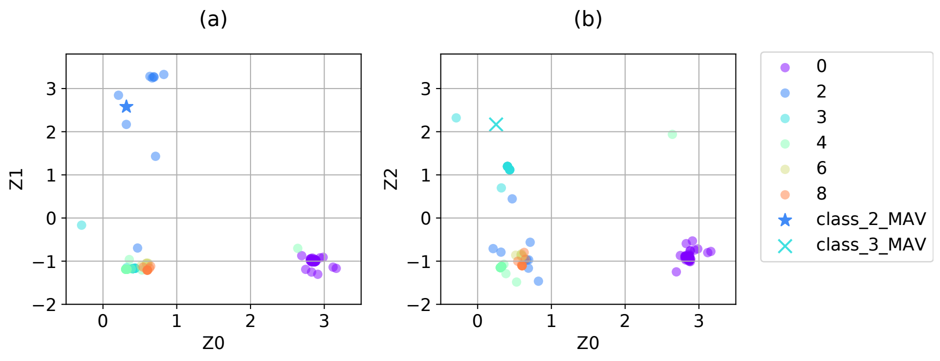
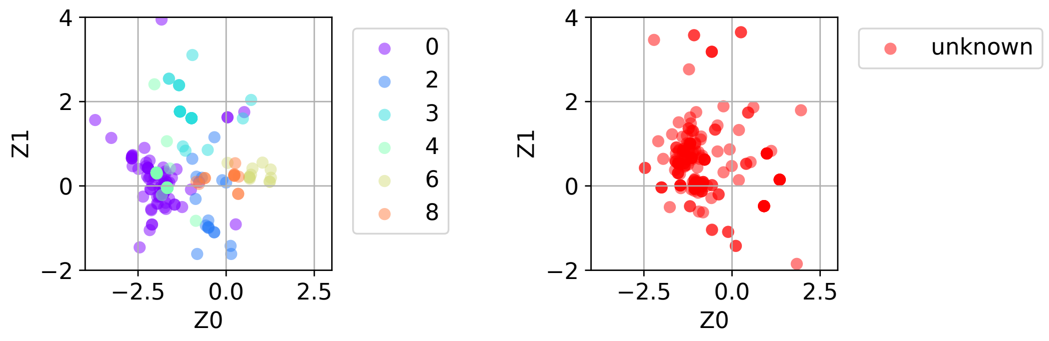


Figure 6.4: Projections of Android dataset known class test instances from final activation layer of Openmax.



(a) (b)

Figure 6.5: Projections of Android dataset (a) known class and (b) unknown class test instances from z-layer of a network trained with only cross entropy.

Performance of Openmax is especially low in case of the Android dataset because of low recall on known classes with small number training instances. The low recall was caused by test instances from the smaller classes being projected further away from the class's mean activation vector (MAV). For example, in Figure 6.4a we see that test instances of class 2 are further away from the MAV of class 2 (marked by '⋆'). As a result, these test instances are predicted as unknown. Similarly, in Figure 6.4b instances of class 3 are far away from the MAV of class 3(marked by 'X'). Performance of ce is also low for Android dataset. This was because unknown class instances were projected close to the known classes (Figure 6.5) resulting them being labeled as known classes. In turn resulting in lower precision score for the known classes.

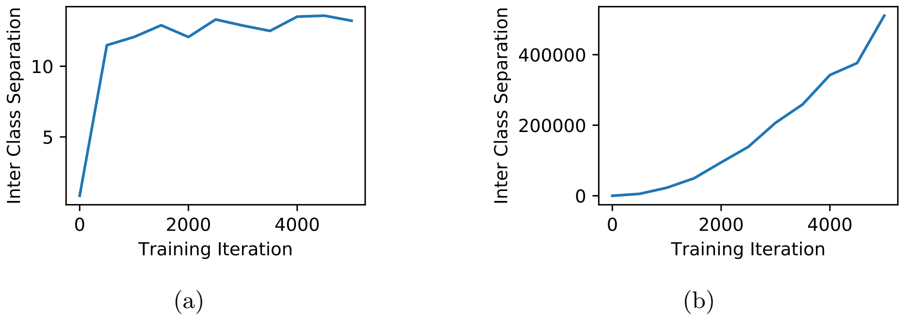


Figure 6.6: Inter class separation for networks trained (a) with batch normalization used in all layers and (b) without batch normalization at the z-layer.

On the topic of the z-layer, one question the reader might have is how to decide on the dimension of this layer. Our suggestion is to chose this experimentally as is the case for other neural network hyper-parameters. In our case, we tuned the z-layer dimension together with other hyper-parameters for experiments in Sections 6.4.4 and 6.4.5 based on the closed set performance on the known class instances in validation set. Based on the results we decided to keep z-layer dimension equal to the number of classes as we didn't see significant improvement from increasing it.

In our experiments, we have observed batch normalization [92] to be extremely important when using ii-loss. Because batch normalization fixes the mean and variance of a layer, it bounds the output of our z-layer in a certain hypercube, in turn preventing the *inter_separation* term in ii-loss from increasing indefinitely. This is evident in Figures 6.6a and 6.6b. Figure 6.6a shows the *inter_separation* of the network where batch normalization used in all layers including the z-layer where *inter_separation* increases in the beginning but levels off. Whereas when batch normalization is not used in the z-layer the *inter_separation* term keeps on increasing as seen in Figure 6.6b.

Autoencoders can also be considered as another way to learn a representation. However, autoencoders do not try to achieve properties P1 and P2 in Section 6.3.1. One of the reasons is auto encoder training is unsupervised. Another reason is because non-regularized autoencoders fracture the manifold into different domains resulting in the representation of instances from the same class being further apart [90]. Therefore, the representation learned does not help in discriminating known class instances from unknown class instances. Figure 6.7 shows the output of an encoder in an autoencoder trained on known class instances and then used to project both known and unknown class instances. For the projection in autoencoder, the known classes are not well separated and outliers get projected to roughly the same area as the known classes.

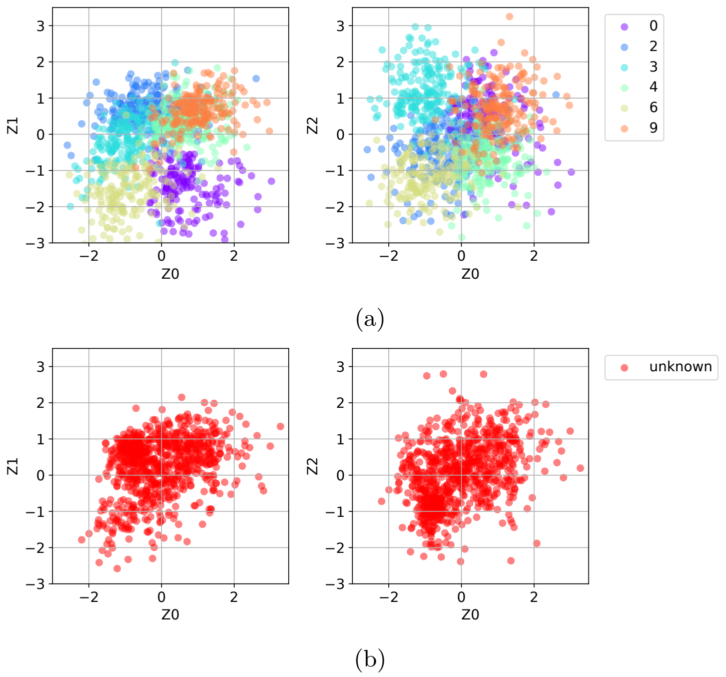


Figure 6.7: Projections of (a) known and (b) unknown class instances using the hidden layer of an Autoencoder. The labels 0,2,3,4,6,9 represent the known classes while the label "unknown" represents the unknown classes.

In Section 6.4.1 we mentioned that we used function call graph (FCG) feature for the malware dataset. We also mentioned that in case of the MS Challenge dataset we reformatted the FCG features proposed in [52] to form a $(63, 63)$ adjacency matrix representation of the graph. We feed this matrix as an input to the convolutional network with a (4,4) kernel. Such kernel shape makes sense when it comes to image input because in images proximity of pixels hold important information. However, it is not obvious to us how nearby cells in a graph adjacency matrix hold meaning full information. We tried different kernel shapes, for example taking an entire row of the matrix at once (because a row of the matrix represent a single nodes outgoing edge weights). However the simple (4,4) giving better close set performance.

6.5 Conclusion

We presented an approach for learning a neural network based representation that projects instances of the same class closer together while projecting instances of the different classes further apart. Our empirical evaluation shows that the two properties lead to larger spaces among classes for instances of unknown classes to occupy, hence facilitating open set recognition. We compared our proposed approach with a baseline network trained to minimize a cross entropy loss and with Openmax (a state-of-art neural network based open set recognition approach). We evaluated the approaches on datasets of malware samples and images and observed that our proposed approach achieves statistically significant improvement.

We proposed a simple threshold estimation technique, Section 6.3.3. However, there is room to explore a more robust way to estimate the threshold. We leave this for future work.

Chapter 7

Unsupervised Open Set Recognition using Adversarial Autoencoders

7.1 Introduction

Scheirer et al. [73] define an open set recognition scenario as one where an “incomplete knowledge of the world is present at training time, and unknown classes can be submitted to an algorithm during testing”. in other words, the classes seen during testing may not have been seen in training; unlike closed set recognition where the assumption is that the same set of classes are seen both during training and test.

Many research works on open set recognition, such as [81, 93, 94], all propose solutions for a supervised open set recognition task where the training data is labeled. However, there are scenarios where open set recognition is needed in an unsupervised setting. That is, in the unsupervised setting, the training data are not labeled with classes and the test data could belong to classes beyond those “latent” classes in the training data.

Consider a scenario where the training set is comprised of instances from multiple data generating distributions (we will refer to these as known or training data distributions).

Furthermore, test data can contain instances from unknown data generating distribution never seen in the training set. Unsupervised open set recognition is the task of identifying whether an instance is from unknown data distributions or the known data distributions in the absence of labeled training data. Wang et al. [95] highlight one such scenario for the task of person re-identification from surveillance camera footages in public space.

Another case where unsupervised open set arises is in malware defense. Malware clustering has been proposed in [6, 15, 18, 19] for automatically identifying similar malware. Since a large number of malware instances are regularly released, malware clustering is used as part of malware analysis workflow where analysts look at representative samples from each cluster instead of analyzing each sample. As new samples come in, it is important to identify if they belong to one of the already created clusters or if they are different from them. Hence, it is important to have the capability to perform unsupervised open set recognition. K. Rieck et al. [18] identified the necessity of having such capability and proposed a malware clustering approach and an associated outlier score for recognizing instances that did not belong to the already identified clusters. Although the authors did not specify that their work was for open set recognition, their outlier score can be used for unsupervised open set recognition.

Unsupervised open set recognition is very similar to anomaly detection. However, there are certain aspects of typical anomaly detection approaches that make them a poor fit for open set recognition tasks. For example, statistical anomaly detection explicitly models the data probability distribution; however, these approaches can have a poor performance on high dimensional data [44]. Proximity-based and density-based approaches determine if an instance is anomalous based on its distance from normal data or based on the number of instances around it; however, both approaches rely on the use of the appropriate distance function which can be difficult to find for the original feature space. As a result, typical anomaly detection methods do not perform well on open set recognition as we show in our evaluations.

We make the following contributions in this chapter: (1) we propose an unsupervised open set recognition approach that uses an Adversarial Autoencoder (AAE) [90] for clustering and combine it with ii-loss, proposed in Chapter 6, to learn a representation that facilitates open set recognition. (2) Our proposed approach gives statistically significant improvement over other approaches on three evaluation datasets; two malware datasets (as the primary focus of this work is the security domain) and one image dataset. (3) We show anomaly or outlier detection algorithms, such as Isolation Forest, do not perform well on unsupervised open set scenarios. (4) We show the applicability of our approach to image recognition domain by evaluating on handwritten digit dataset (MNIST).

7.2 Related Works

Open set recognition is starting to get considerable attention in recent literature. Dietterich [79] even presents it as one of the challenges that need to be addressed to build a robust AI system. Most of the research in this area [73, 81, 83, 93, 94] so far has focused on supervised open set recognition; a few of these are discussed below. Scheirer et al. [73] formalize open space risk and propose a one-class open set recognition (they call a 1-vs-set machine) as an SVM where the positive region is bounded by two parallel planes. Scheirer et al. [94] further develop this idea to address a multi-class open set recognition and provide a probabilistic decision score for rejecting instances as unknown. Bendale and Boult [81] present a deep neural network based multi-class open set recognition approach which redistributes the activation vector of the last neural network layer to compute the pseudo-activation for the unknown class and provide the probability of the unknown class. On the unsupervised open set recognition side, Want et al. [95] present an approach for learning to identify if individuals in surveillance footages that are of the same individuals in a target list without the need for a labeled data.

Another related area is anomaly or outlier detection. Proximity-based anomaly detection approaches such as [69], for example, use the distance of an instance to its k-nearest neighbor as its anomaly/outlier score. Sugiyama and Borgwardt [97] sample a random subset of the dataset, for improved scalability, and use the distance to the nearest sample in the subset as an outlier score. Others such as the widely used LOF algorithm [70] use the density of the neighborhood around an instance relative to the density of other instances in its neighborhood as a measure of the instances outlier score. Isolation Forest [98], a random forest like anomaly detection algorithm, can also be thought of as a density-based approach. Although isolation forest does not directly calculate the density of an instance, it measures how anomalous an instance is by how difficult it is to isolate that instance from other instances around it through a random recursive partitioning. The denser the area around an instance the more difficult it is to separate it hence the less likely for it to be an anomaly. Clustering based anomaly detection has also been proposed [99].

7.3 Approach

7.3.1 Background

Adversarial Autoencoder (AAE)

Makhzani et al. [90] propose an approach for turning the decoder network of an autoencoder into a generative model. In their proposed approach an adversarial autoencoder, shown in Figure 7.1, is trained on two objectives (1) the traditional reconstruction error criteria and (2) adversarial training criteria. The reconstruction objective aims to train the encoder network to encode the original input into latent variables and the decoder network to reconstruct the original input from the latent variables.

$$reconstruction\_loss = \frac{1}{2N} \sum_{i=1}^{N} \left\| \hat{\vec{x_i}} - \vec{x_i} \right\|_2^2 \tag{7.1}$$

where N is number of instances and $\hat{\vec{x_i}}$ is the reconstruction of the input $\vec{x_i}$.

The task of the adversarial criteria is to match the latent variable posterior distribution to an arbitrary prior, usually a Gaussian distribution $\mathcal{N}(0, I)$. An adversarial criteria can be viewed as min-max game between two networks; a generator network (G) and a discriminator network (D). Goodfellow et al. [87] express this min-max game as having value function $V(G, D)$, Equation 7.2. In this game, the discriminator network maximizes the probability of assigning the correct labels to the real samples from the data distribution p_{data} and the fake samples from the generator G. The generator on the other hand, minimizes $\log(1 - D(G(v))$ (i.e., tries to fool the discriminator into labeling the fake samples as real).

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}}[\log D(x)] + \mathbb{E}_{v \sim p(v)}[\log(1 - D(G(v))] \tag{7.2}$$

As Goodfellow points out in [100], although the min-max game in the above equation is useful for theoretical analysis, it does not perform well in practice. In practice, we use two separate loss functions for the discriminator (D) and the generator (G). The discriminator is trained to minimize the standard cross-entropy function used in conventional binary classifier with a sigmoid output, Equation 7.3. The caveat here is that the discriminator is trained on mini-batch data from p_{data} and $p(v)$. The generator, on the other hand, is trained to minimize the loss function in Equation 7.4.

$$J^{(D)} = -\frac{1}{2}\mathbb{E}_{x\sim p_{data}}[\log D(x)] - \frac{1}{2}\mathbb{E}_{v\sim p(v)}[\log(1 - D(G(v))] \tag{7.3}$$

$$J^{(G)} = -\frac{1}{2}\mathbb{E}_{v\sim p(v)}[\log D(G(v)] \tag{7.4}$$

In case of the AAE shown in Figure 7.1, the encoder network and the z-layer together form the generator G. The discriminator D, on the other hand, is a separate network which we refer to as "gaussian discriminator" in the Figure 7.1. We use the term "Gaussian Discriminator" to signify that it is used to discriminate instances sampled from a Gaussian Distributions from those coming from the z-layer. The real samples are sampled from an arbitrary prior distribution; for example, p_{data} in Equation 7.2 and 7.3 is a Gaussian distribution $\mathcal{N}(0, I)$. The fake samples are from the output of the z-layer (i.e., $p(v)$ in Equation 7.2 and 7.4 is the encoder and the z-layer).

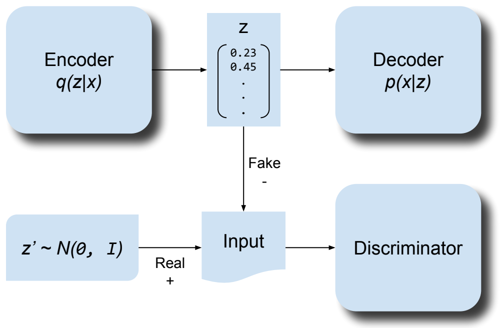


Figure 7.1: Adversarial autoencoder architecture.

During each training iteration the training alternates between two phases. In the first phase (called the reconstruction phase), the encoder network, decoder network and the z-layer weights are updated to minimize the reconstruction error. Then in the second phase (the adversarial phase), the discriminator network weights are updated to tell the real samples (sampled from the Gaussian) from the fake/generated samples (coming from the encoder output). This is done by minimizing Equation 7.3. Also in the second phase, the encoder and the z-layer, which act as the generator in the adversarial training, are updated to confuse the discriminator network. That is, the encoder and the z-layer weights are updated to minimize Equation 7.4.

Variants of Adversarial Autoencoder (AAE)

Makhzani et al. also demonstrate multiple variants of the adversarial autoencoder for clustering and dimensionality reduction. In the clustering variant, shown in Figure 7.2, the encoder network is connected to the z-layer and the y-layer. The y-layer, which is a softmax layer, is a probability distribution over the cluster labels. There is an additional adversarial network (which we will refer to as "Categorical Discriminator") for regularizing such that the cluster prediction match a Categorical distribution (i.e., one-hot encoding).

Each training iteration of the clustering variant of AAE consists of three phases. The first is the reconstruction phase, similar to the earlier configuration, updates the encoder network, the z-layer, the y-layer, and decoder network weights to minimize a reconstruction error. The second phase is the Gaussian adversarial phase which is done in the same way as the original adversarial autoencoder configuration. The third is the categorical adversarial phase. In the context of this phase, the Categorical discriminator represents the D network; whereas the encoder and y-layer constitute the generator G network. In this phase, the weights of the D network are updated to minimize Equation 7.3. Then the weights of the G network are updated to minimize Equation 7.4. In Equations 7.3 and Equation 7.4, the real samples are now sampled from a Categorical distribution (i.e., p_{data} in Equation 7.3). The fake samples come from $p(v)$ which is the encoder and the z-layer.

In the dimensionality reduction variant shown in Figure 7.3, a lower dimensional representation is learned. To do this, the cluster architecture is modified by first multiplying $\vec{y}$ by an $n \times m$ weight matrix W_c (where n is the number of clusters and m is the dimensionality of $\vec{z}$). The result is referred to as a "clusterhead" which has lower dimension than $\vec{y}$ but same as $\vec{z}$. In order to combine $\vec{y}$ and $\vec{z}$, the clusterhead is added to the latent variables $\vec{z}$ to give a lower dimension representation. This representation is fed to the decoder network. They also

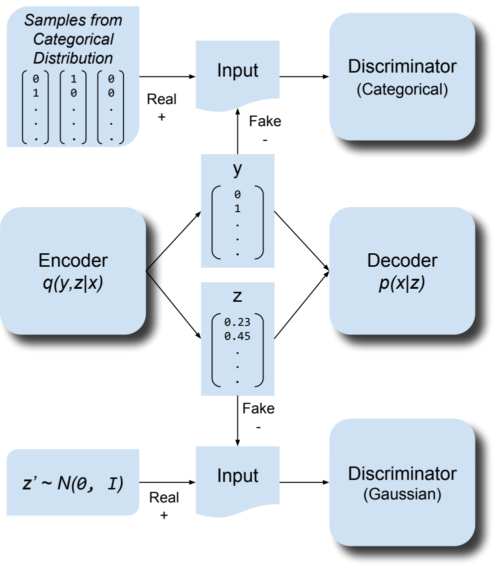


Figure 7.2: Adversarial autoencoder architecture for clustering.

add a loss function that linearly penalizes the distance between cluster heads if this distance is less than some threshold η and zero otherwise.

II-Loss

The aim of ii-loss, introduced in Chapter 6, is to learn a neural-network-based representation that facilitates open set recognition. II-loss achieves this by ensuring the representation fulfills two properties: (P1) instances from the same class are close to each other; (P2) instances from different classes are further apart. The intuition is that as the known classes are further apart from each other, unknown class instances will have more space to occupy between them making the task of recognizing unknown class instances easier.

Given training data, consisting of set of instance X and labels Y, and a neural network g, the learned representation of instance $\vec{x}$ is $\vec{z} = g(\vec{x})$. The neural network is trained to minimize ii-loss which defined as:

$$\textit{ii-loss} = \textit{intra\_spread} - \textit{inter\_sparation} \tag{7.5}$$

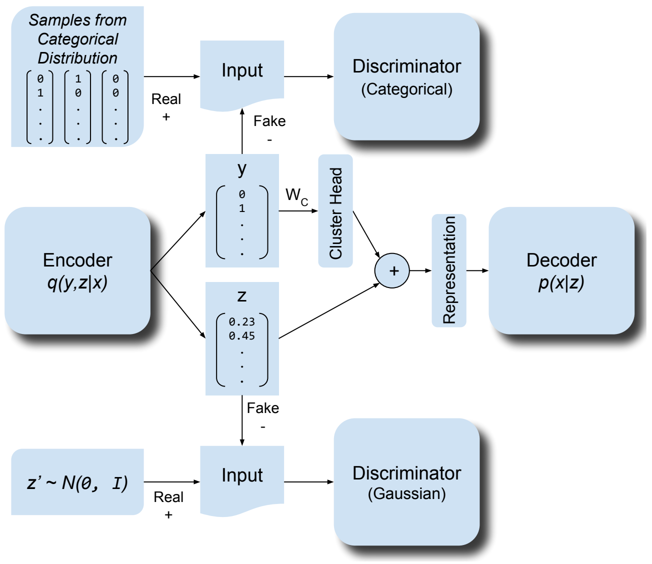


Figure 7.3: Adversarial autoencoder architecture for dimensionality reduction.

The first component of the ii-loss, *intra_spread*, aims to minimize the distance the projection $\vec{z}$ has from the instance's class center.

$$intra\_spread = \frac{1}{N}\sum_{j=1}^{K}\sum_{i=1}^{|C_j|}\|\vec{\mu_j} - \vec{z_i}\|_2^2 \tag{7.6}$$

where $|C_j|$ is the number of training instances in class C_j, N is the number of training instances, K is the number of known classes, and μ_j is the mean of class C_j :

$$\vec{\mu_j} = \frac{1}{|C_j|}\sum_{i=1}^{|C_j|}\vec{z_i} \tag{7.7}$$

The second term of the ii-loss, *inter_sparation*, maximizes the distance between the class centers.

$$inter\_sparation = \min_{\substack{1\le m\le K \\ m+1\le n\le K}} \|\vec{\mu_m} - \vec{\mu_n}\|_2^2 \tag{7.8}$$

7.3.2 Unsupervised Open Set Recognition

The continuous latent variables $\vec{z}$ learned by an AAE does not directly try to fulfill the two properties of ii-loss which have shown to be useful for open set recognition. The original $\vec{z}$

in AAE is used to learn a representation for reconstruction, and it is regularized to follow a normal distribution by the adversarial network so that it can be used for the generative model. For example in Figure 7.5a we see that all instances are projected together into the same region. Although the dimensionality reduction variant of AAE adds a loss function for separating clusters, the loss term is set to zero after the distance between clusters gets larger than some threshold value. Additionally, the dimensionality reduction variant of AAE does not try to make the clusters compact. As a result, the separation between the clusters is small; leaving a smaller space for the unknown instances to occupy.

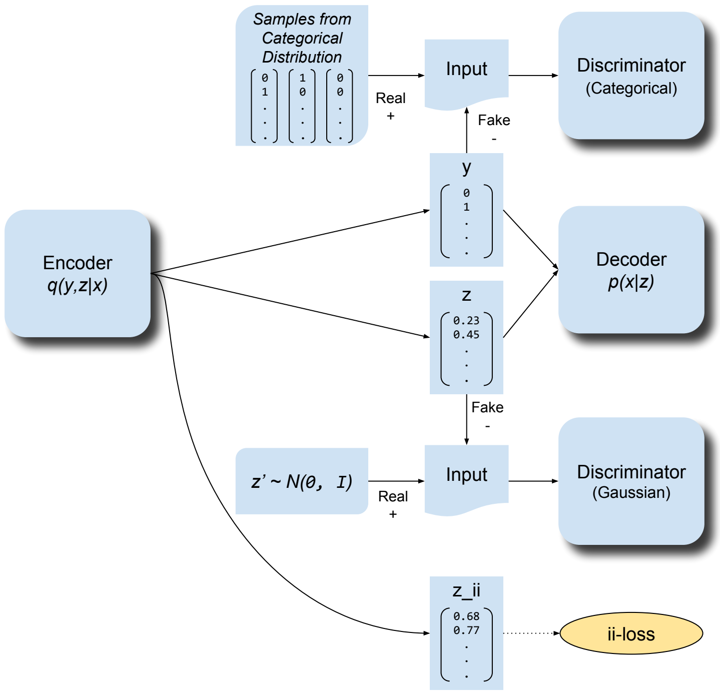
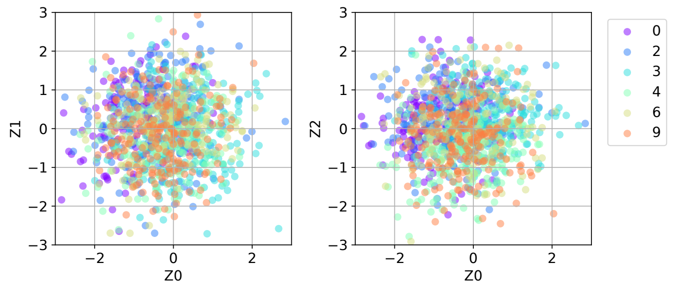
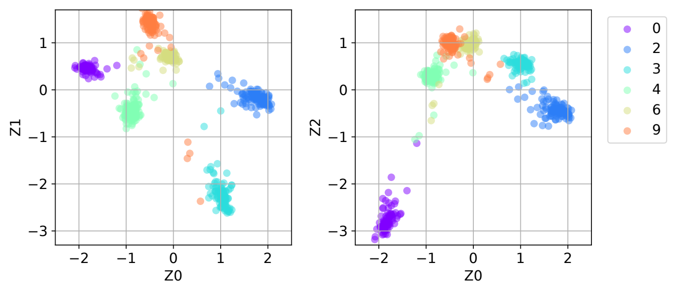


Figure 7.4: AAE-II: architecture for unsupervised open set recognition.

We propose combining ii-loss with AAE to learn a continuous representation for facilitating open set recognition. Since the adversarial regularization works in contrary to ii-loss and as a result, degrades open set recognition performance; we propose to learn a different continuous representation $\vec{z\_ii}$. In $\vec{z\_ii}$, instances from the same cluster are closer to each other while instances from different clusters are further apart, this is shown in Figure 7.5b. We will refer to this new representation $\vec{z\_ii}$, shown in Figure 7.4, as a z_ii-layer. The z_ii-layer shares the same encoder network as the z-layer and the y-layer. However, it is not given as input to the decoder as shown in Figure 7.4.



(a) z-layer



(b) z_ii-layer

Figure 7.5: Scatter plot of the 15 dimensional z-layer and z_ii-layer for 1000 random instances. The legend indicates the true label of the points.

The original ii-loss, in Chapter 6, was used for supervised open set recognition, where the training class labels are used for calculating ii-loss. In this case, however, the open set recognition task at hand is unsupervised. Hence, the objective of ii-loss needs to be changed accordingly. We propose to use ii-loss to project similar instances from the same cluster closer together while projecting different clusters further apart. We can use the same ii-loss formulated in Equations 7.5-7.8. However, C_j in Equations 7.6 and 7.7 in this case refers to a cluster label instead of class labels. A cluster label (or cluster-id) of an instance is computed as $cluster\_id = \underset{1\leq j\leq K}{\text{argmax}}\, y[j]$ where K is the number of clusters (i.e., the dimension of the y-layer).

The training of the AAE-II is carried out in two stages as shown in Algorithm 7.1. In the first stage (lines 2-8), we train the AAE network with updates from the reconstruction phase, gaussian adversarial phase and categorical adversarial phase. During this training, we monitor the percent of change in cluster membership among the training instances during each training iteration. We

Algorithm 7.1: Training AAE-II

```
Input :
        $X$: Training data and labels
1  // Stage One
2  for number of stage one iterations do
3      Sample a mini-batch $X_{batch}$ from $X$
4      Update Encoder, z-layer, y-layer, and Decoder to minimize reconstruction loss,
         Equation 7.1.
5      Update Gaussian Discriminator to minimize Equation 7.3 where
         $p(v) = L_z(Enc(X_{batch}))$ and $p_{data} = \mathcal{N}(0, I)$
6      Update Categorical Discriminator to minimize Equation 7.3 where
         $p(v) = L_y(Enc(X_{batch}))$ and $p_{data} = \mathcal{C}at()$
7      Update Encoder and z-layer to minimize Equation 7.4 where $p(v) = L_z(Enc(X_{batch}))$
8      Update Encoder to y-layer to minimize Equation 7.4 where $p(v) = L_y(Enc(X_{batch}))$
9  end
10 // Stage Two
11 for number of stage two iterations do
12     Sample a mini-batch $X_{batch}$ from $X$
13     Update Encoder, z-layer, y-layer, and Decoder to minimize reconstruction loss,
         Equation 7.1.
14     Update Gaussian Discriminator to minimize Equation 7.3 where
         $p(v) = L_z(Enc(X_{batch}))$ and $p_{data} = \mathcal{N}(0, I)$
15     Update Categorical Discriminator to minimize Equation 7.3 where
         $p(v) = L_y(Enc(X_{batch}))$ and $p_{data} = \mathcal{C}at()$
16     Update Encoder and z-layer to minimize Equation 7.4 where $p(v) = L_z(Enc(X_{batch}))$
17     Update Encoder to y-layer to minimize Equation 7.4 where $p(v) = L_y(Enc(X_{batch}))$
18     Update Encoder and z_ii-layer to minimize ii-loss, Equation 7.5
19 end
20 Calculate cluster means and save as part of the model.
```

L_z denotes the z-layer
L_y denotes the y-layer
$Enc()$ denotes the encoder network
$Cat()$ denotes a categorical distribution

stop the first stage when the change in cluster membership converges to a value less than 1% or when a pre-set training iteration is reached. In the second stage (lines 11-18), we add the z_ii-layer to the network and start updating the z_ii-layer and the encoder network to minimize ii-loss. We also continue the reconstruction, gaussian adversarial, and categorical adversarial phases as in the first stage. After the training is completed, the cluster means(centers), at the z_ii-layer, are calculated for each cluster using all the training instances and stored as part of the model.

Outlier Score and Threshold Estimation

To measure the degree to which a test instance $\vec{x}$ is an outlier, we get the z_ii-layer output ($\vec{z\_ii}$) for this instance and then measure how far this vector is from each cluster mean. The distance from the closest cluster center (mean) is then used as an outlier score.

$$outlier\_score(\vec{x}) = \min_{1 \leq j \leq K} \left\| \vec{\mu}_j - \vec{z\_ii} \right\|_2^2 \tag{7.9}$$

where $\vec{z}$ is the output of the z_ii layer for input $\vec{x}$, and $\vec{\mu}_j$ is the mean of cluster j.

To determine whether a test instance is an outlier, a threshold value on the outlier score is needed. We estimate this threshold based on the outlier score of the training data. An assumption is made on how much of the training data to consider as outliers using the contamination ratio. For example, if the contamination ratio is set to be 2%, then the 98 percentile largest outlier score of the training data is set as the threshold value. During testing any instance that has an outlier score higher than the threshold is predicted to be an outlier.

7.4 Evaluation

7.4.1 Dataset

We evaluated our proposed approach on three datasets. The first dataset is the Microsoft Malware Challenge Dataset [1] which consists of 10868 labeled disassembled windows malware samples from 9 families. In case of our experiment, we use 10260 samples from this dataset since our disassemble file parser was not able to properly process the remaining files. The majority class has 2936 samples whereas the minority class has 34 samples with the median class size of 1012 samples.

The second dataset used in the following evaluation is from the Android Malware Genome Project [67]. The dataset consists of Android malware apps from many families. However, many of these families only have few samples. Therefore in our evaluation, we use only families that have at least 40 samples to be able to split the dataset into training, validation, and test and have enough samples. After removing the smaller classes, the dataset has 986 samples. The majority class in this dataset has 309 samples whereas the minority class has 46 samples with the median class size being 69. To evaluate the applicability of the proposed approach to domains outside malware classification, we also evaluate it on the MNIST handwritten digit dataset.

The malware features used in both malware dataset are based on function call graph features proposed in Chapter 4. In both cases, we use minhash to cluster the functions into 64 clusters based on the unigram of the function opcodes. The cluster-ids are then used to label the functions, and the vertex and edge features are extracted.

7.4.2 Simulating Open Set Dataset

For evaluating our proposed approach, we simulate open set datasets from the original evaluation datasets. All three of the original datasets are labeled; however, we do not use the labels in training the evaluated approaches. The labels are only used to create the open set datasets and calculating performance metrics.

We start by randomly selecting K classes whose instances are going to be present in the training set. In other words, the training set is made up of instances from K data distributions; we will refer to these as known or training data distributions. In case of the MS and Android datasets, we choose 75% of instances from the K classes to be part of the open set training set and add the remaining to the open set test set. All the instances that are not part of the K classes are added to the open set test set. In case of the MNIST dataset, we select all the instances from the K classes in the original MNIST training dataset to be part of the open set training set. All the instances from the original MNIST test set are used in the open set test dataset (i.e., this contains the K training classes and the other classes that are not part of the training set). In all cases the classes labels are not used for training, they are only used for calculating unsupervised open set recognition performance metrics.

7.4.3 Evaluated Approaches

Unsupervised open set recognition has similarities with anomaly/outlier detection if one considers the instances from an unknown data distributions as being outliers. It is important to show how an outlier detection technique performs for unsupervised open set recognition. Therefore we evaluate the approach proposed in contrast to Isolation Forest [98]. We also evaluate how well clustering-based outlier detection performs in an unsupervised open set scenario. Similar to what is proposed by Rieck et al. [18], we use KMeans-based approach where the distance from cluster centroid is used for detecting unknown data distribution instances.

In case of the Android and MS datasets, both the KMeans-based and Isolation Forest are given the malware FCG features as input features. However, in case of the MNIST dataset, the KMeans-based approach and Isolation Forest understandably do not perform well when given the raw image pixels as input features. So instead we first use an autoencoder to transform the pixels into an intermediate representation and then feed this intermediate representation to the two algorithms.

We also evaluate our proposed approach implemented using TensorFlow. The source code of our implementation is publicly available [1]. The hyper-parameters for the neural network are available in Appendix B. In case the Android and MS dataset the FCG features are given as input features to our approach. Whereas in case of MNIST dataset we take the raw images as input.

7.4.4 Detecting Instances from Unknown Data Distributions

We evaluate our proposed approach, and the other two methods mentioned in Section 7.4.3 on the task of detecting instance from unknown data distributions. For these evaluations, three open set datasets are randomly created from each the three datasets as discussed in Section 7.4.2. We then run 10 experiments for each of the three open set dataset, which means we have a total of 30 experiments for each original dataset. The objective of these experiments is to perform unsupervised open set recognition. In other words, discriminating test instances that belong to the known data distributions (i.e., the data distributions present in the training set) from instances that belong to unknown data distributions (i.e., the data distributions not seen in the training set).

We report the results of these experiments first using Area Under the Curve (AUC), which measures the unsupervised open set recognition performance independent of the outlier threshold. We use a t-test to measure the statistical significance of the difference in performance AUC values.

[1] https://github.com/shrtCKT/unsupervised_opennet

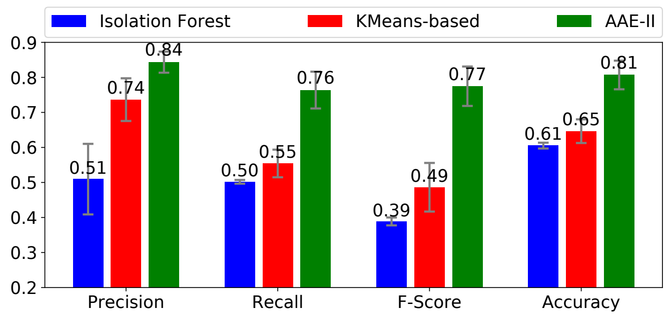
Table 7.1: 30-run average AUC of discriminating instances of unknown data distributions from instances of known data distributions. When calculating the AUC, the positive label represented instances from unknown data distributions. The underlined average AUC values are higher with statistical significance (p-value less than 0.05 with a t-test) compared to the values that are not underlined on the same row. The average AUC values in **bold** are the largest average AUC values in each row.

	FPR	Isolation Forest	KMeans-based	AAE-II
MNIST	100%	0.619(±0.085)	0.841(±0.078)	**0.898**(±0.029)
	10%	0.008(±0.003)	0.031(±0.013)	**0.058**(±0.010)
MS	100%	0.547(±0.167)	0.568(±0.107)	**0.829**(±0.065)
	10%	0.014(±0.017)	0.011(±0.009)	**0.027**(±0.017)
Android	100%	0.716(±0.050)	0.789(±0.016)	**0.817**(±0.139)
	10%	0.011(±0.002)	0.026(±0.009)	**0.029**(±0.016)

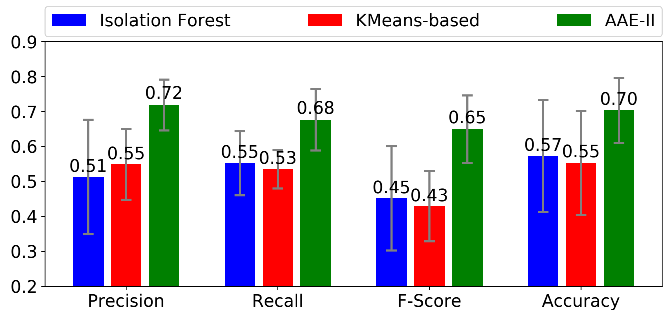
Table 7.1 shows the 30-run average AUC. As shown in Table 7.1, our proposed approach (AAE-II) has significantly higher average AUC for MNIST and MS dataset both up to 100% False Positive Rate (FPR) and up to 10% FPR.

In case of the Android dataset, the KMeans-based approach has higher average AUC up to 100% FPR over AAE-II although the improvement is not statistically significant. When considering AUC up to 10% FPR AAE-II has a higher value. Again the improvement is not statistically significant.

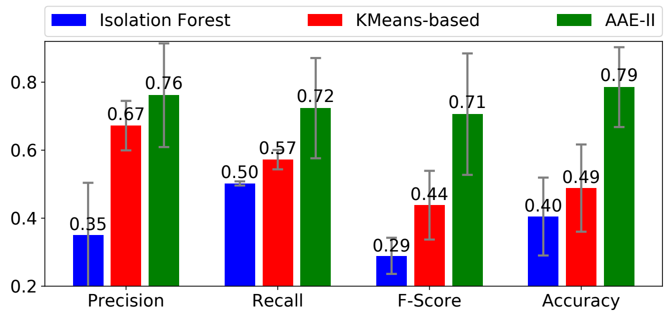
We now present the result of these experiment from a different perspective where a threshold value is estimated. The threshold is then used to predict whether an instance belongs to known data distribution or it does not. We present the results of this prediction in Figure 7.6. In all three datasets, our proposed approach records statistically significant improvement in F-Score and Accuracy compared to the other approaches.



(a) MNIST Dataset



(b) MS Challenge Dataset



(c) Android Genom Dataset

Figure 7.6: Average Precision, Recall, F-Score and Accuracy of 30-runs. The metric calculations for precision recall and f-score included the averaging known and the unknown and further averaged over 30 experiment runs.

7.5 Discussion

One of the questions we had when combining ii-loss with the clustering variant of AAE was the effect of ii-loss on the clustering performance of AAE. To quantify this effect we report here the clustering performance of the experiments in Section 7.4.4. Additionally, we report clustering performance of the same networks trained without ii-loss. We report the clustering performance results in terms of the cluster homogeneity as defined by Rosenberg and Hirschberg [101]. The homogeneity values have a range of 0.0 to 1.0. A homogeneity of 1.0 indicates that a cluster is purely made up of instances from a single class, which is a desirable result. For the homogeneity computation, elements in cluster i are assigned the label of the training instance that maximizes the probability of belonging to that cluster. As shown in Table 7.2, the clustering performance with and without ii-loss remain similar. We conducted a t-test to assess whether the difference in the clustering performances is significant. The results from the t-test showed that the differences were not statistically significant since the p-values from all three rows in the table were higher than 0.24. Therefore, the effect of ii-loss on the clustering performance is negligible.

Table 7.2: 30-run average cluster homogeneity. The average Homogeneity values in **bold** are the largest average Homogeneity values in each row. We used a t-test to determine whether the difference in performance were significant and in all three datasets the differences were not statistically significant.

	AAE	AAE-II
MNIST	**0.834**(±0.049)	0.815(±0.076)
MS	0.771(±0.051)	**0.779**(±0.044)
Android	0.756(±0.040)	**0.757**(±0.040)

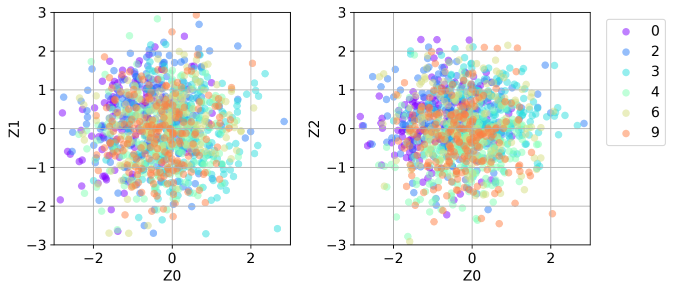
In our implementation the use a separate z_ii-layer was driven by the realization that the original z-layer of AAE was not separating the instances from the different clusters. We showed this in Figure 7.5 where we plotted the z-layer and z_ii-layer projections of 1000 random data instances. We show these figures again in Figure 7.7, but this time the plots contain both instances from known data distributions, and instance from unknown data distributions. We see in Figures 7.7a and 7.7b that the projections from both known and unknown overlap for the most part. Hence, it is difficult to discriminate them. On the other hand, ii-loss pushes the projections of different clusters further apart; so when an instance from unknown data distribution is encountered, the network projects them in the space between the clusters. This effect can be seen in Figure 7.7c and 7.7d.

To understand the effect of the number of clusters (the dimension of the y-layer in Figure 7.4) and dimension of the z_ii layer (we will refer to as z_ii-dim) on the open set performance, we conducted experiments on the three original datasets. Similar to what we did in Section 7.4.4, we randomly create 3 open set datasets from each original dataset as discussed in Section 7.4.2. We then run 10 experiments for each of the three open set dataset, which means we have a total of 30 experiments for each original dataset at a given value of z_ii-dim and number of clusters. The results of these experiments are presented using the 30-run average Area Under the Curve (AUC), which measures the unsupervised open set recognition performance in identifying instances from unknown data distributions.

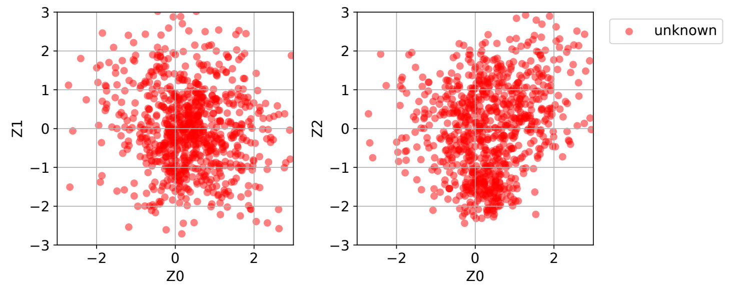
The figures on the right in Figure 7.8 show the effect of using different z_ii dimension on the average AUC (each point is the average of 30 experiments). The figures indicate that the performance of our approach relatively remains the same for different z_ii dimension values for all three datasets. This is good because it shows the open set performance is not sensitive to the z_ii dimension. Hence, it is one less hyper-parameter to we need to tune. As long as one chooses the dimensions of z_ii that is not too small the proposed approach should work well.

The figures on the left in Figure 7.8 show the effect of using a different number of clusters on the average AUC (each point is the average of 30 experiments). We expected the open set recognition performance (i.e., average AUC) to increase as the number of clusters increases since the clusters become more homogeneous(pure) as the number of clusters increase. The results of MS and Android datasets seem to agree with our expectation as seen in Figures 7.8b and 7.8b. In case of MNIST dataset, however, average AUC decreases as the number of clusters increase, which is the opposite of what we expected to happen.

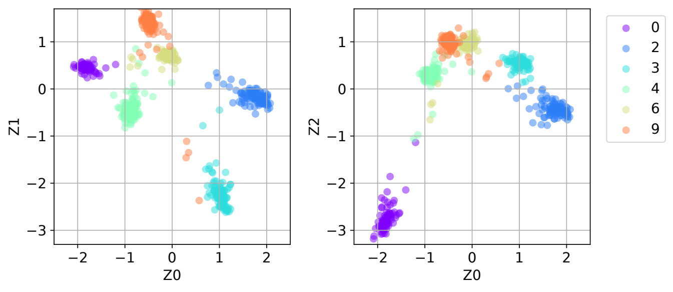
Our first hypothesis in trying to explain the unexpected result (i.e., the decrease in open set recognition performance with an increase in cluster number for MNIST dataset) was that it might be because the cluster quality(homogeneity) was not improving with an increase in the number of clusters. However, we found out that was not the case; cluster homogeneity in the experiments increases with an increase in the number of clusters. Our second hypothesis was that ii-loss was not separating the clusters well (hence explaining in poor open set recognition performance) when the number of clusters increased. We found out that inter-cluster separation did decrease as the number of clusters increased, however, this was the case for the MS and Android datasets as well, but in those two datasets, the open set recognition improved with an increase in the number of clusters. Therefore this does not explain these results. At this moment we are not able to find the answer for this unexpected result and leave this off as future work. The conclusion we can draw from these experiments is that the optimal cluster number (i.e., the dimension of the y-layer) for a good open set recognition performance differs from dataset to dataset and need to be tuned more carefully.



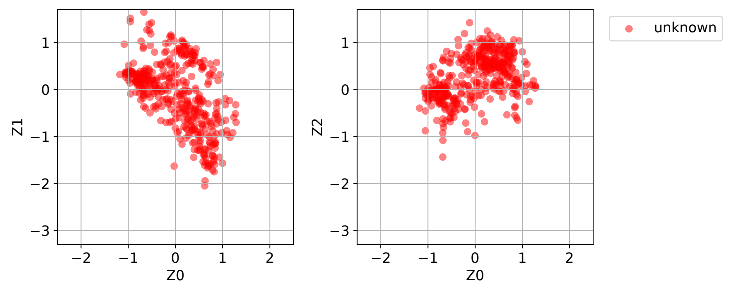
(a) z-layer projections
for instances from known data distribution.



(b) z-layer projections
for instances from unknown data distribution.

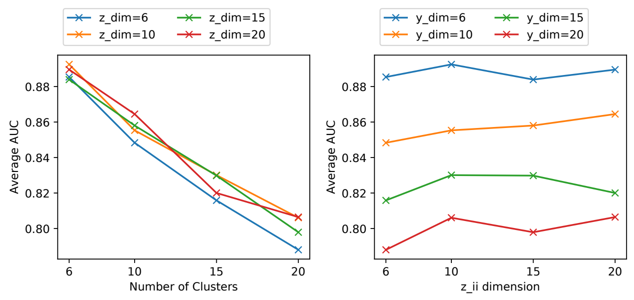


(c) z_ii-layer projections
for instances from known data distribution.

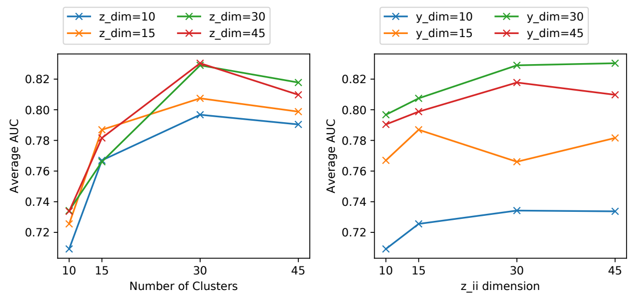


(d) z_ii-layer projections
for instances from unknown data distribution.

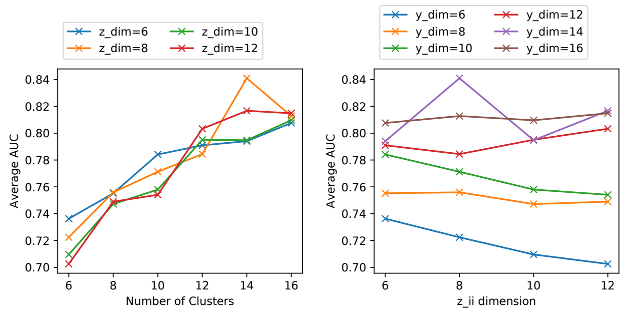
Figure 7.7: Scatter plot of the 15 dimensional z-layer and z_ii-layer for 1000 random instances from known and unknown data distributions. The legend indicates the true label of the points.



(a) MNIST Dataset



(b) MS Challenge Dataset



(c) Android Genom Dataset

Figure 7.8: Effect of cluster size and z dimension on performance.

7.6 Conclusion

In this chapter, we presented the problem of unsupervised open set recognition where the task is to identify instances in the test set that are not from the training data distributions. Our approach learns a representation for facilitating open set recognition in an unsupervised setting. In this representation, instances from the same cluster are projected close together while instances from different clusters are projected further apart.

We compare our approach with closely related outlier detection methods and show that our approach gives a statistically significant improvement on three publicly available datasets. Two of the evaluation datasets are from the malware domain. To show the applicability of our approach to domains different from malware, we evaluate our approach on an image dataset as our third dataset. Our results also show that typical anomaly or outlier detection methods such as Isolation Forest do not perform well when used for unsupervised open set recognition tasks.

Although the proposed approaches do perform better than other methods, there is still much room for improvement. For example, a more robust outlier threshold estimation approach can be explored.

Chapter 8

Conclusions

Anti-malware vendors receive a large number of suspected malware files daily. To cope with this influx of malware files, machine learning techniques, such as classification and clustering, are used to group similar malware into families. Grouping malware into families allows human analysts to examine representative samples from each group instead of examining all file. The effectiveness of the machine learning techniques heavily relies on how much discriminative information the features extracted from these files carry. Since it is difficult to collect training samples from each malware family, the classification and clustering systems are forced to operate in an open set scenario where they are faced with instances from never before seen families. In this dissertation, we worked on extracting useful malware features and proposed the malware classification and clustering systems capable of operating in an open set scenario.

In Chapters 3 and 4 we explored malware feature extraction. In Chapter 3 we evaluated malware features used in past works. In Chapter 4 we proposed a linear time approach for extracting function call graph (FCG) features. We overcame the performance overhead associated with FCG based features by using a novel technique to convert FCG representation into a vector representation. Our approach not only gives better classification accuracy compared past FCG based approach but also scales better. In both chapters, we evaluate the proposed features under a closed set assumption.

We shifted our attention to an open set scenario in Chapter 5. We proposed a method for extracting features from the output of a close set classifier to help identify unknown malware families. An outlier detector is trained on the features extracted from the output of the classifier to determine if an instance is from unknown family. When evaluated on two malware datasets, our proposed approach is more accurate when compared to other open set recognition algorithms.

In Chapter 6 we presented a neural-network-based representation for addressing the supervised open set recognition problem. In this representation instances from the same class are close to each other while instances from different classes are further apart, resulting in statistically significant performance improvement in open set recognition when compared to other approaches on three datasets from two different domains.

We extended the ii-loss presented in Chapter 6 by combining it with Adversarial Autoencoders (AAE) to address the unsupervised open set recognition problem in Chapter 7. Our work attempts to learn a representation for facilitating open set recognition in an unsupervised manner. Our evaluations show that the proposed approach gives improved open set recognition performance compared to other approaches.

8.1 Limitations and Future Work

The malware features used in this work were extracted through static analysis. It has been shown that the use of more advanced binary packers by malware authors can make disassembling, which is required for extraction these static analysis features, difficult. One way to handle this is to use dynamic analysis to run the packed program and dump process memory image once the malware has unpacked itself. After that, the feature extraction approaches proposed in this work can be applied the process code segment dump.

In the open set recognition approaches presented in Chapters 6 and 7 we estimate the outlier threshold based on a contamination ratio hyper-parameter. The contamination ratio specifies what percent of the training data to consider as outliers, based on that, an outlier threshold is then estimated. A more robust approach can be explored similar to the way Openmax [81] computes a probability over K+1 classes (where the K classes represent the known classes, and the last class represents the unknown class).

Appendix A

Evaluation Network Architectures for Chapter 6

We evaluate 4 networks: ii, ce, ii+ce, and Openmax. All four networks have the same architecture upto the fully connected z-layer. In case of the MNIST dataset, the input images are of size (28,28) and are padded to get an input layer size (32,32) with 1 channel. Following the input layer are 2 non-linear convolutional layers with 32 and 64 unites (filters) which have a kernel size of (4,4) with a (1,1) strides and SAME padding. The network also has max polling layers with kernel size of (3,3), strides of (2,2), and SAME padding after each convolutional layer. Two fully connected non-linear layers with 256 and 128 units follow the second max pooling layer. Then a linear z-layer with dimension of 6 follows the fully connected layers. In the case of ii+ce and ce networks, the output the z-layer are fed to an additional linear layer of dimension 6 which is then given to a softmax function. We use Relu activation function for all the non-linear layers. Batch normalization is used though out all the layers. We also use Dropout with keep probability of 0.2 for the fully connected layers. Adam optimizer with learning rate of 0.001, beta1 of 0.5, and beta2 of 0.999 is used to train our networks for 5000 iterations. In case of the Openmax network, the output of the z-layer are directly fed to a softmax layer. Similar to the Openmax paper we use a distance that is a weighted combination of normalized Euclidean and cosine distances. For the ce, ii, and ii+ce we use contamination ratio of 0.01 for the threshold selection.

The open set experiments for MS Challenge dataset also used a similar architectures as the four networks used for MNIST dataset with the following differences. The input layer size MS Challenge dataset is (67,67) with 1 channel after padding the original input of (63,63). Instead of the two fully connected non-linear layers, we use one fully connected layer with 256 units. We use dropout in the fully connected layer with keep probability of 0.9. Finally the network was trained using Adam optimizer with 0.001 learning rate, 0.9 beta1 and 0.999 beta2.

We do not use a convolutional network for the Android dataset open set experiments. We use a network with one fully connected layer of 64 units. This is followed by a z-layer with dimension of 6. For ii+ce and ce networks we further add a linear layer with dimension of 6 and the output of this layer is fed to a softmax layer. In case of Openmax the output of the z-layer is directly fed to the softmax layer. For Openmax we use a distance that is a weighted combination of normalized Euclidean and cosine distances. We use Relu activation function for all the non linear layers. We used batch normalization for all layers. We also used Dropout with keep probability of 0.9 for the fully connected layers. We used Adam optimizer with learning rate of 0.1 and first momentem of 0.9 to train our networks for 10000 iterations. For the ce, ii, and ii+ce we use contamination ratio of 0.01 for the threshold selection.

The closed set experiments use the same set up as the open set experiments with the only difference coming from the dimension of the z-layer. For the MNIST dataset we used z dimension of 10. For the MS and Android datasets we use z dimension of 9.

Appendix B

Evaluation Network Architectures for Chapter 7

Microsoft Challenge Dataset

For the Microsoft Malware Challenge Dataset, we used two non-linear layers of 250 hidden units with ReLU activation function for the encoder, decoder, gaussian discriminator, and categorical discriminator networks. For the output layer of the decoder, we used Sigmoid activation function. We used 30 linear units for the z_ii-layer and z-layer. The dimensionality of the y-layer was set to 30 (i.e., 30 clusters). Batch normalization was used in all layers and dropout was not used in any layer. We used Adam Optimizer for the reconstruction loss, adversarial losses, and ii-loss. The learning rate of 0.001 and beta1 of 0.9 was used for the first 1000 training iterations of the first stage after which we change the learning rate to 0.0001. We ran the first training stage for 3000 iterations and the second stage for 2000 iterations. We used a batch size of 512.

For the evaluation of the kmeans-based approach we used the same number of clusters (i.e. 30 clusters). For all the evaluated approaches we used a contamination ratio of 0.05 for the outlier threshold estimation.

Android Dataset

For the Android Genome Project Dataset, we used a similar architecture as the Microsoft dataset with the following differences. We used a single hidden layer with 100 units and a ReLU activation function for the encoder, decoder, gaussian discriminator, and categorical discriminator networks. We used 14 clusters and 12 linear units for the z_ii-layer and z-layer. We used learning rate of 0.001 for the first 3000 training iterations of the first stage after which we change the learning rate to 0.0001. The beta1 of the Adam Optimizer for reconstruction loss was fixed at 0.9 while it was fixed at 0.1 for the other loss functions. We ran the first training stage for 3000 iterations and the second stage for 1000 more iterations. We used a batch size of 256.

For the evaluation of the kmeans-based approach we used the same number of clusters (i.e., 10 clusters). For all the evaluated approaches we used a contamination ratio of 0.01 for the outlier threshold estimation.

MNIST

For the MNIST dataset, we used a convolutional network in the encoder. We first pad the input images of size (28,28) to get an input layer size (32,32) with 1 channel. Following the input are two convolutional layers with 32 and 64 units, (4,4) kernel size with a stride of (1,1) and SAME padding. We use max-polling of (3,3) kernel, (2,2) stride size, and SAME padding after each convolutional layer. The decoder network uses the same architecture with deconvolutional layers of 64 and 32 units. A linear layer was used for the output layer of the decoder. For the Gaussian and Categorical discriminator networks, we used two hidden layers with 1000 units. We use ReLU activation function for all non-linear layer. The dimensionality of the y-layer was set to 6 (i.e., 6 clusters) and 15 linear units were used in the z_ii-layer and z-layer. Batch normalization was used in all layers and dropout was not used.

Adam Optimizer with a learning rate of 0.001 was used for the first 3000 training iterations of the first stage after which the learning rate is changed to 0.0001. The beta1 of the Adam Optimizer for the reconstruction loss was fixed at 0.9 while it was fixed at 0.1 for the other losses. We ran the first training stage for 3000 iterations and the second stage for an additional 7000 iterations. We used a batch size of 512.

Appendix C

List of Publications

The following are the publications that have been produced during the course of this Ph.D. research.

Published works

Hassen, Mehadi, and Carvalho, Marco M, and Philip K. Chan, "Malware Classification Using Static Analysis Based Features." *In the proceedings of IEEE Symposium Series on Computational Intelligence (IEEE SSCI 2017)*, 2017.

Hassen, Mehadi, and Philip K. Chan. "Scalable Function Call Graph-based Malware Classification." *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY2017).* ACM, 2017.

Hassen, Mehadi, and Philip K. Chan. "Learning to Identify Known and Unknown Classes: A Case Study in Open World Malware Classification." *In the proceedings of The 31th International FLAIRS Conference*, 2018.

Submitted for review

Hassen, Mehadi, and Philip K. Chan. ”Learning a Neural-network-based Representation for Open Set Recognition.” *arXiv preprint arXiv:1802.04365* (2018). Submitted for review.

Work in progress

Hassen, Mehadi, and Philip K. Chan. ”Unsupervised Open Set Recognition using Adversarial Autoencodes.”

Bibliography

[1] Microsoft malware classification challenge (big 2015). https://www.kaggle.com/c/malware-classification, 2015. [Online; accessed 27-April-2015].

[2] Giovanni Vigna. Antivirus isn't dead, it just can't keep up. http://labs.lastline.com/lastline-labs-av-isnt-dead-it-just-cant-keep-up.

[3] Matthew G Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE, 2001.

[4] Guanhua Yan, Nathan Brown, and Deguang Kong. Exploring discriminatory features for automated malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 41–61. Springer, 2013.

[5] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM, 2009.

[6] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[7] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

[8] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, (2):32–39, 2007.

[9] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[10] Anubis - malware analysis for unknown binaries. https://anubis.iseclab.org.

[11] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[12] Av-test malware statistics. http://www.av-test.org/en/statistics/malware/.

[13] Rafiqul Islam, Ronghua Tian, Lynn Batten, and Steve Versteeg. Classification of malware based on string and function feature selection. In *Cybercrime and Trustworthy Computing Workshop (CTC), 2010 Second*, pages 9–17. IEEE, 2010.

[14] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 3422–3426. IEEE, 2013.

[15] Xin Hu, Kang G Shin, Sandeep Bhatkar, and Kent Griffin. Mutantx-s: Scalable malware clustering based on static features. In *USENIX Annual Technical Conference*, pages 187–198, 2013.

[16] Swathi Pai, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. Clustering for malware classification. *Journal of Computer Virology and Hacking Techniques*, pages 1–13, 2016.

[17] Gideon Creech and Jiankun Hu. A semantic approach to host-based intrusion detection systems using contiguousand discontiguous system call patterns. *Computers, IEEE Transactions on*, 63(4):807–819, 2014.

[18] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[19] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

[20] Barbara G Ryder. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, (3):216–226, 1979.

[21] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[22] Ming Xu, Lingfei Wu, Shuhui Qi, Jian Xu, Haiping Zhang, Yizhi Ren, and Ning Zheng. A similarity metric method of obfuscated malware using function-call graph. *Journal of Computer Virology and Hacking Techniques*, 9(1):35–47, 2013.

[23] Orestis Kostakis, Joris Kinable, Hamed Mahmoudi, and Kimmo Mustonen. Improved call graph comparison using simulated annealing. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1516–1523. ACM, 2011.

[24] Deguang Kong and Guanhua Yan. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1357–1365. ACM, 2013.

[25] Lei Xu and Erkki Oja. Improved simulated annealing, boltzmann machine, and attributed graph matching. In *Neural Networks*, pages 151–160. Springer, 1990.

[26] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5:1–3, 2005.

[27] Stavros D Nikolopoulos and Iosif Polenakis. A graph-based model for malware detection and classification using system-call groups. *Journal of Computer Virology and Hacking Techniques*, pages 1–18, 2016.

[28] Carey Nachenberg, Jeffrey Wilhelm, Adam Wright, and Christos Faloutsos. Polonium: Tera-scale graph mining and inference for malware detection. 2011.

[29] Acar Tamersoy, Kevin Roundy, and Duen Horng Chau. Guilt by association: large scale malware detection by mining file-relation graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge Discovery and Data Mining*, pages 1524–1533. ACM, 2014.

[30] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[31] Jack W Stokes, John C Platt, Helen J Wang, Joe Faulhaber, Jonathan Keller, Mady Marinescu, Anil Thomas, and Marius Gheorghescu. Scalable telemetry classification for automated malware detection. In *Computer Security–ESORICS 2012*, pages 788–805. Springer, 2012.

[32] L. Nataraj and B. S. Manjunath. Spam: Signal processing to analyze malware [applications corner]. *IEEE Signal Processing Magazine*, 33(2):105–117, March 2016.

[33] Lakshmanan Nataraj, S Karthikeyan, Gregoire Jacob, and BS Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, page 4. ACM, 2011.

[34] Lakshmanan Nataraj, Vinod Yegneswaran, Phillip Porras, and Jian Zhang. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 21–30. ACM, 2011.

[35] William W Cohen. Fast effective rule induction. In *Proceedings of the twelfth international conference on machine learning*, pages 115–123, 1995.

[36] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning Data Mining, Inference, and Prediction.* Springer Series in Statistics. Springer, second edition, 2008.

[37] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[38] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

[39] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[40] Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware classification with recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1916–1920. IEEE, 2015.

[41] Alex Graves. *Supervised sequence labelling*. Springer, 2012.

[42] 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*, volume 96, pages 226–231, 1996.

[43] Anand Rajaraman, Jeffrey D Ullman, Jeffrey David Ullman, and Jeffrey David Ullman. *Mining of massive datasets*, volume 1. Cambridge University Press Cambridge, 2012.

[44] Pang-Ning Tan, Michael M. Steinbach, and Vipin Kumar. *Introduction to data mining*. 2006.

[45] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 309–320. ACM, 2011.

[46] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. On the detection of anomalous system call arguments. In *Computer Security–ESORICS 2003*, pages 326–343. Springer, 2003.

[47] Sandeep Bhatkar, Abhishek Chaturvedi, and R Sekar. Dataflow anomaly detection. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[48] Peng Li, Hyundo Park, Debin Gao, and Jianming Fu. Bridging the gap between data-flow and control-flow analysis for anomaly detection. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 392–401. IEEE, 2008.

[49] Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting intrusions through system call sequence and argument analysis. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):381–395, 2010.

[50] 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*, volume 2, 2011.

[51] Ultimate packer for executables(upx). http://upx.sourceforge.net.

[52] Mehadi Hassen and Philip K. Chan. Scalable function call graph-based malware classification. In *7th Conference on Data and Application Security and Privacy*, pages 239–248. ACM, 2017.

[53] Weka 3. http://www.cs.waikato.ac.nz/ml/weka/.

[54] Josh Attenberg, Kilian Weinberger, Anirban Dasgupta, Alex Smola, and Martin Zinkevich. Collaborative email-spam filtering with the hashing trick. CEAS, 2009.

[55] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

[56] An in-depth look into the win32 portable executable file format. https://msdn.microsoft.com/en-us/magazine/cc301808.aspx.

[57] Ida pro. https://www.hex-rays.com/products/ida.

[58] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image and Vision computing*, 27(7):950–959, 2009.

[59] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[60] Shohei Hido and Hisashi Kashima. A linear-time graph kernel. In *2009 Ninth IEEE International Conference on Data Mining*, pages 179–188. IEEE, 2009.

[61] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 183–194. ACM, 2016.

[62] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web*, pages 271–280. ACM, 2007.

[63] Austin Appleby. Murmurhash3. https://github.com/aappleby/smhasher, 2008.

[64] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[65] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[66] P. Chan. *An Extensible Meta-Learning Approach for Scalable and Accurate Inductive Learning*. PhD thesis, Department of Computer Science, Columbia University, New York, NY, 1996.

[67] Android malware genome project. http://www.malgenomeproject.org/.

[68] Adagio. https://github.com/hgascon/adagio.

[69] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. Efficient algorithms for mining outliers from large data sets. In *ACM Sigmod Record*, volume 29, pages 427–438. ACM, 2000.

[70] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. Lof: identifying density-based local outliers. In *ACM sigmod record*, volume 29, pages 93–104. ACM, 2000.

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

[72] Ji Zhang. Advancements of outlier detection: A survey. *ICST Transactions on Scalable Information Systems*, 13(1):1–26, 2013.

[73] W. J. Scheirer, A. de Rezende Rocha, A. Sapkota, and T. E. Boult. Toward open set recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(7):1757–1772, 2013.

[74] Paul Bodesheim, Alexander Freytag, Erik Rodner, Michael Kemmler, and Joachim Denzler. Kernel null space methods for novelty detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3374–3381, 2013.

[75] Paul Bodesheim, Alexander Freytag, Erik Rodner, and Joachim Denzler. Local novelty detection in multi-class recognition problems. In *2015 IEEE Winter Conference on Applications of Computer Vision*, pages 813–820. IEEE, 2015.

[76] Lalit P Jain, Walter J Scheirer, and Terrance E Boult. Multi-class open set recognition using probability of inclusion. In *European Conference on Computer Vision*, pages 393–409. Springer, 2014.

[77] Libsvm-openset. https://github.com/ljain2/libsvm-openset, 2017.

[78] Knfst. https://github.com/cvjena/knfst, 2017.

[79] Thomas G. Dietterich. Steps toward robust artificial intelligence. *AI Magazine*, 38(3):3–24, 2017.

[80] Abhijit Bendale and Terrance Boult. Towards open world recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1893–1902, 2015.

[81] Abhijit Bendale and Terrance E Boult. Towards open set deep networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1563–1572, 2016.

[82] Enrique G Ortiz and Brian C Becker. Face recognition for web-scale datasets. *Computer Vision and Image Understanding*, 118:153–170, 2014.

[83] Lalit P Jain, Walter J Scheirer, and Terrance E Boult. Multi-Class Open Set Recognition Using Probability of Inclusion. pages 393–409, 2014.

[84] Qing Da, Yang Yu, and Zhi-Hua Zhou. Learning with augmented class by exploiting unlabeled data. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[85] ZongYuan Ge, Sergey Demyanov, Zetao Chen, and Rahil Garnavi. Generative openmax for multi-class open set classification. *arXiv preprint arXiv:1707.07418*, 2017.

[86] Douglas O Cardoso, Felipe França, and João Gama. A bounded neural network for open set recognition. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–7. IEEE, 2015.

[87] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[88] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[89] Ethan Rudd, Andras Rozsa, Manuel Gunther, and Terrance Boult. A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world solutions. *IEEE Communications Surveys & Tutorials*, 2017.

[90] Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, and Brendan Frey. Adversarial autoencoders. *arXiv preprint arXiv:1511.05644*, 2015.

[91] Mnist hand written digit dataset. http://yann.lecun.com/exdb/mnist/.

[92] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[93] He Zhang and Vishal M Patel. Sparse representation-based open set recognition. *IEEE transactions on pattern analysis and machine intelligence*, 39(8):1690–1696, 2017.

[94] Walter J Scheirer, Lalit P Jain, and Terrance E Boult. Probability models for open set recognition. *IEEE transactions on pattern analysis and machine intelligence*, 36(11):2317–2324, 2014.

[95] Hanxiao Wang, Xiatian Zhu, Tao Xiang, and Shaogang Gong. Towards unsupervised open-set person re-identification. In *Image Processing (ICIP), 2016 IEEE International Conference on*, pages 769–773. IEEE, 2016.

[96] Mehadi Hassen and Philip K Chan. Learning a neural-network-based representation for open set recognition. *arXiv preprint arXiv:1802.04365*, 2018.

[97] Mahito Sugiyama and Karsten Borgwardt. Rapid distance-based outlier detection via sampling. In *Advances in Neural Information Processing Systems*, pages 467–475, 2013.

[98] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 413–422. IEEE, 2008.

[99] Dantong Yu, Gholamhosein Sheikholeslami, and Aidong Zhang. Findout: finding outliers in very large datasets. *Knowledge and Information Systems*, 4(4):387–412, 2002.

[100] Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160*, 2016.

[101] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*, 2007.