Evaluation Techniques for the Applications of Machine Learning in Cybersecurity

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy by

Said Sulaiman Al-Riyami

March 2022

Dedication

This thesis is dedicated to my mother Raya Al-Riyami and the memory of my father Sulaiman Al-Riyami. Also I would like to dedicate the thesis to my wife, Ibtisam Alsalmi, who has been a wonderful support throughout my research.

Acronyms

ANN Artificial Neural Network.

AUC Area Under The Curve.

CNN Convolutional Neural Network.

FN False Negative.

FP False Positive.

IoT Internet of Things.

NB Nave Bayes.

NIDS Network Intrusion Detection System.

ROC Receiver Operating Characteristic.

RSA Rivest Shamir Adleman.

SVM Support Vector Machines.

TN True Negative.

TP True Positive.

Glossary

Binary Classification A type of machine learning classification problem, where the task is to classify the data into two groups.

Cryptanalysis A process of studying cryptographic systems to look for weaknesses or leaks of information.

Cryptojacking A type of cybercrime where a criminal secretly uses a victim's computing power to mine cryptocurrency.

Malware Stands for malicious software. It is any type of software that is intended to harm or hack the user.

Multiclass Classification A type of machine learning classification problem, where the task is to classify the data into more than two groups.

Network Intrusion Detection System A system that monitor the computer network traffic in order to detect any intrusion.

Prime Factorisation A way of expressing a number as a product of its prime factors.

Acknowledgements

I want to express my special thanks and gratitude to my primary supervisor Dr Alexei Lisitsa for his continued support, discussions, and guidelines during the PhD journey. I would also like to thank my secondary supervisor, Prof. Frans Coenen, for his constant feedback and support throughout the time spent completing the work. My deepest gratitude to both for the time and effort given under their supervision that added a lot to my experience. I will carry this valuable experience for the rest of my professional life.

I want to extend my gratitude to my wife, Ibtisam Alsalmi, for her limitless support and patience throughout the whole journey. Also, I express my thanks to my brothers, sisters, family, and friends for their support and valuable prayers.

I am extremely grateful to the Oman government for providing me with this opportunity. Finally, my thanks go to all the people who have supported me to complete the research work directly or indirectly.

Abstract

The number of internet users is on the rise and more and more parts of our lives depend on the internet. However, there is also an increase in online threats. The newly discovered malware are in millions every year. This makes the manual process of detecting and defending against the attacks harder, thus, we need a more automated way to do so. Machine learning (ML) can help the automation by finding zero-day attacks or new malware.

The work presented in this thesis is concerned with the evaluation methods used to measure the performance of the machine learning applications for cybersecurity. The current evaluation methods do not take into consideration the rapid change nature of security applications. For that, we need other evaluation methods that give a deeper understanding of the ML model designed for the application of cybersecurity. There are the two most-known methods of evaluation: handout set and k-Fold cross-valuation. These methods are designed to give a general understanding of the model performance to select the best model that can perform well in the deployment stage. However, various types of tasks create various challenges for these methods. In some cases, we needed more specialised methods.

First, we need to check if the result generated by the model is just a matter of overfitting because of the data generated from the same environment, for example, an overfitting by the computer network architecture. Second, the threats to cybersecurity are consistently changing. Therefore, we need to measure how much we need to update our model, how often, and what kind of threats there are—for example, the constant changes in malware. Finally, some problems can generate an almost infinite amount of data. Therefore, we need to look for different evaluation methods for these problems such as cryptography problems.

In this work, we propose different ways of tackling these problems. For the first type, we propose the use of the Cross-Datasets Evaluation method. We will take an example from Network Intrusion Detection System (NIDS). The data will be generated from two different computer networks. We propose five different ways of splitting the data based on the duration (time) and data size for the second type. We will use the method with malware detection and propose a new deep learning model for that. For the last type, we propose the use of a generator for the evaluation. We will take the RSA algorithm security as an example. By proposing those kinds of methods, we have a better and more practical way of evaluating the cybersecurity applications of machine learning.

List of Publications

Here are the list publication that have been published during the research:

1. **Said Al-Riyami**, Frans Coenen, and Alexei Lisitsa. “A Re-evaluation of Intrusion Detection Accuracy: Alternative Evaluation Strategy. “Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018. APA

2. **Said Al-Riyami**, Alexei Lisitsa, and Frans Coenen. “An efficient way to deal with algorithmically generated data in deep learning”, “Proceedings of the 2019 Emerging Technology Conference”, Editors: M.K. Bane and V. Holmes, ISBN 978-0-9933426-4-6

3. **Said Al-Riyami**, Alexei Lisitsa, and Frans Coenen. “Cross-datasets evaluation of machine learning models for intrusion detection systems”, “Proceedings of the Sixth International Congress on Information and Communication Technology”, Editors: Yang, X.-S., Sherratt, S., Dey, N., Joshi, A. (Eds.), ISBN 978-981-16-2101-7 (Due: August 21, 2021)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

The work presented in this thesis is focused on finding different techniques to evaluate the applications of machine learning for cybersecurity. Cybersecurity is important in the modern era.

The number of internet users and the amount of time spent on the internet are growing. According to the Data Reportal's report [40], the estimated number of internet users worldwide is 4.80 billion as of July 2021. The same report indicated an increase in a quarter of a billion users in a year. The other estimation from Internet World Stats [93] is 5.1 billion users as of March 2021. In 2015, the number of internet users was estimated to be around 2 billion worldwide users. The growth in the number of users within a short period is astonishing. The increase may be because of the advancements in the connections (5G network, fibre optics, and internet satellite), the decrease in the cost of the internet access and mobile phone.

On the other hand, the amount of time spent on the internet is on the increase. The 2020 COVID-19 pandemic shifted many people to working online. According to [27], the average content consumption over on the internet is six hours and 59 minutes, an increase of 15% compared with 2019. This is the highest increase since 2012. Furthermore, the report predicted that the average daily use would surpass 8 hours by 2022.

The number of internet users are dwarfed by the number of Internet of Things (IoT) devices connected to the internet. Juniper Research reported 35 billion IoT devices in 2020 and is predicted to reach 83 billion by 2024 [86]. In the US, the average number of connected devices by households was 10 in 2020 [87]. It is predicted that the IoT-connected devices will generate 79.4 Zettabytes of data in 2025 [10].

The increase in the number of users and IoT devices with the time spent on the internet will increase the surface of the attack. Not only that, but use cases are also increasing. For example, the car we use might also be connected to the internet (or might have autonomous driving capability). So an attack on those applications might cost us not only privacy but also a life.

The number of attacks and malware are also rapidly increasing. In recent years, the number of malware and intrusion attacks have escalated dramatically. Kaspersky reports that in 2020 it detected on average 360,000 new malicious files a day [38]. The AV-Test Institute has reported a similar result [49]. The McAfee Labs Threats Report of June 2021 stated that it detected 10.68 million new ransomware and 2.51 million new ransomware in the first quarter of 2021 [50]. Cybersecurity Ventures predicted that ransomware would cost victims $10 billion (USD) in 2021 and $265 billion (USD) in 2031 [28]. SonicWall lab reported 2.5 billion malware attacks, 32.2 million IoT malware attacks, 304.7 million ransomware attacks, 51.1 million cryptojacking attacks, and 2.5 trillion intrusion attempts in the first half of 2021 [1].

One of the most used intrusion detection methods is *signature-based* detection, which uses knowledge of intrusion in the form of signatures, which can be easily checked against monitored traffic (Network-based Intrusion Detection System - NIDS) or CPU instruction (Host-based Intrusion Detection System - HIDS). The major issue with this method is the availability of signatures in the context of the constantly growing number of attacks; the set of signatures must be constantly updated, and this requires expert knowledge. The automation of the detection procedures can partially alleviate this issue using machine learning methods, particularly supervised machine learning.

For years, traditional machine learning algorithms have been used for intrusion detection systems (IDS). For example, decision tree learning [94], random forest [35], logistic regression [89], Linear Support Vector Machine (SVM) [88], k-nearest neighbours algorithm (k-NN) [26] and others. But recently, the use of deep learning models has become popular in cybersecurity applications. Several architectures can be used for deep learning, including Artificial Neural Networks (ANN), Autoencoders (AE) [42], Convolution Neural Networks (CNN) [44] and Recurrent Neural Networks (RNN) with different cells, like Long Short-Term Memory (LSTM) [36] and Gated Recurrent Unit (GRU) [13].

There are many applications of machine learning in cybersecurity that cover the defensive and offensive parts of the area. The applications include intrusion detection system (IDS), malware detection, cryptography, stenography, spam detection, etc (more about the applications in Chapter 2). The work of this project focuses on evaluation methods of the intrusion detection system (IDS), malware detection and cryptography.

The conventional way of conducting research in cybersecurity (intrusion detection system in particular) using machine learning are as follows:

- Use a known dataset in cybersecurity.
- Use any machine learning algorithm or a variation of it.
- Follow holdout or k-fold cross-validation to train and test.
- Report the results and compare them with other results of the same dataset.

The typical output of such a process has an accuracy of 97% and above [62, 65, 95]. Such results are questionable and have been discussed in several publications [32, 79, 75]. The issues with such results can be summarised as follows:

- We do not know if the results are just a case of overfitting.
- Does the model learn something about the intrusion in general, or just for a particular setting used in training?
- How can we evaluate the quality of the dataset used?
- What are other insights we can learn from the trained model?
- What about the practicality of such a model in real-world applications?

The current method of evaluation is not designed directly to answer those questions and they are essential for cybersecurity applications of machine learning. Therefore, this thesis wants to explore different ways to evaluate machine learning models in cybersecurity applications.

The remaining structure of the introductory chapter is as follows: In Section 1.2 the motivation for the research is discussed in more detail. In Section 1.3, the main research question and the subquestions are given. Section 1.4 outlines the research methodology that is used to answer the research questions. Section 1.5 is about the research contribution. Section 1.6 lists the main contributions of the research. Section 1.6 outlines the publications that have been published during the work of the research. Section 1.7 outlines the structure of the rest of the thesis.

1.2 Research Motivation

As discussed, the number of internet users increases and our lives become more dependent on the internet. There is also an increase in the amount of Internet of Things (IoT) devices. On the other hand, the number of threats is increasing. The number of newly discovered malware is very high, and we do not know how many there are that have not been discovered. Therefore, the process of detection by using signature-based methods might not be sufficient today. For that, we need to automate the process of finding the intrusion and the use of machine learning is promising for such tasks.

Machine learning algorithms have shown promising results in computer vision and Natural Language Processing (NLP). Thus, they should be a good tool to use in cybersecurity applications. However, the area of cybersecurity in general, and intrusion detection in particular, has special characteristics compared with other areas. The main one is that it suffers from concept drifting [96]. Concept drift means that the trained model does not adjust with the changes that happen in the target data over time. The concept drift in cybersecurity has several reasons as follows:

- Constant changes in the intrusion. This includes the changes in malware and the type of attacks. The attacker also constantly tries to avoid getting detected. The goal of the attacks is to keep changing to evade detection. Furthermore, the model should not only try to detect a known attack but also a zero-day (unknown) attack. Finally, the model also should constantly be updated with new intrusions.
- Changes in computer traffic. This is due to new or updated protocols, different ways of generating the traffic (desktop, mobile, or IoT devices). It also includes the behaviour of users or organisations over time.

The current ways to evaluate the applications of machine learning in cybersecurity is similar to other areas. In this thesis, we want to explore more ways to evaluate the machine learning models in different cybersecurity applications that give us more insights about the intrusion, ML model and the dataset that have been used.

We can summarise the motivations of this thesis as follows:

1. Our lives become more dependent on the internet and the threats are rapidly increasing.
2. We need to automate the process of finding the intrusion, and the use of machine learning is promising.

3. The current ways of evaluating the machine learning application in cybersecurity are insufficient due to the characteristics of cybersecurity like concept drift.

4. We need to explore more ways of evaluating the machine learning model in cybersecurity that yields more insight about the intrusions.

1.3 Research Question

Given the above motivation for the work in this thesis, we are more focused on exploring different techniques of evaluating the machine learning models in cybersecurity applications. Therefore, the main research question is:

"How can we evaluate different machine learning cybersecurity applications that give more insights compared with the current methods?"

The current methods use of Holdout and K-fold cross-validation. These methods are generic and not designed to go deeper for understanding the ML models in cybersecurity.

To answer the main question, we need to consider answering the following subquestions:

1. How can we evaluate the change in the computer network environment effect in the trained model?

2. What is the effect on the performance of the model when the network environment changed?

3. What groups of attacks are easier and harder to detect?

4. How can we evaluate how often we need to retrain our model for malware detection?

5. What is a suitable criteria for retraining the model? Time-based (wait for a fixed duration) or size-based (wait for a number of samples)?

6. What are the groups of malware that require more frequent model retraining, and what groups require less?

7. How can we evaluate the model in algorithmically generated data problems like the one in cryptanalysis?

1.4 Research Methodology

To answer the above research question and subquestions, we start with an examination of the current evaluation methods. First, we will train a model and tune the hyperparameters to have a similar or better result than previously reported results from the same dataset. Then we will introduce three different directions of evaluation from the current one: (1) cross-datasets, (2) different ways of data splitting, and (3) generator evaluation. For the cross-datasets, we focused on network-based intrusion detection systems. For data-splitting evaluation, we focused on malware detection. For the use of generator evaluation, we focused on the area of cryptanalysis. While those methods can be used in different applications, they are more suitable for the context of those applications. Therefore, more details about each application will be discussed in detail in the following chapters.

In cross-dataset evaluation, the data used in training is generated from a different computer network than the testing data. Therefore, the learning of the intrusion should be transferred from one computer network to another. We will use different machine learning algorithms to train and test. We will have two different datasets that share the same objective (designed for the network-based intrusion detection system NIDS) but are generated from two different network architectures. For the experiment, we use two public datasets : NSL-KDD [62] and gureKDD [61]. We will train the model in one dataset as a whole and test in the other one, then, we will swap them. The goal is to find out how much the known machine learning models learn about the intrusion in general as opposed to the intrusion in particular network settings. And we will understand which dataset performs better. Finally, we will consider the binary and multiclass classifications.

In the splitting evaluation, we propose five different ways to split the data. This evaluation aims to understand the behaviour of the malware detector models over time to detect different types of malware. This allows us to understand many aspects, like how much we need to retrain our models and which types of malware required constant updates, and if our model is sufficient to generalise malware detection in the foreseeable future. Each way of splitting time-labelled data is describing how we can split training and testing data. We give the following names for the five split methods: (1) normal flow, (2) forward accumulative with the whole testing, (3) forward accumulative with splits testing, (4) backward accumulative with the whole testing, and (5) backward accumulative with split testing. Each way of splitting will provide different insights. Then, we have two different criteria of splitting: (1) time and (2) size-based. In time-based splitting, all splits are equal in terms of duration. In size-based ones, all splits are equal in terms of the number of samples. This will help us understand which criteria are better to retrain the model over time. In the end, we will try to make a model that performs better in this type of evaluation.

The evaluation by the generator is designed when the data used to train the model can be generated algorithmically. In this type of evaluation, data generation will run simultaneously when the model training runs. This process will run as much as we want. Such an evaluation technique is suitable for cryptanalysis problems. We will take a ML attack on the RSA algorithm as a case study for this type of evaluation method. The input will be the public values of the RSA and try to predict the private values. The data can be fed to the model in a binary or decimal format.

Figure 1.1 visualises the summary of different evaluation techniques that we will use in this thesis. We cover three applications of machine learning in cybersecurity, but the same evaluation methods can be used in other areas.

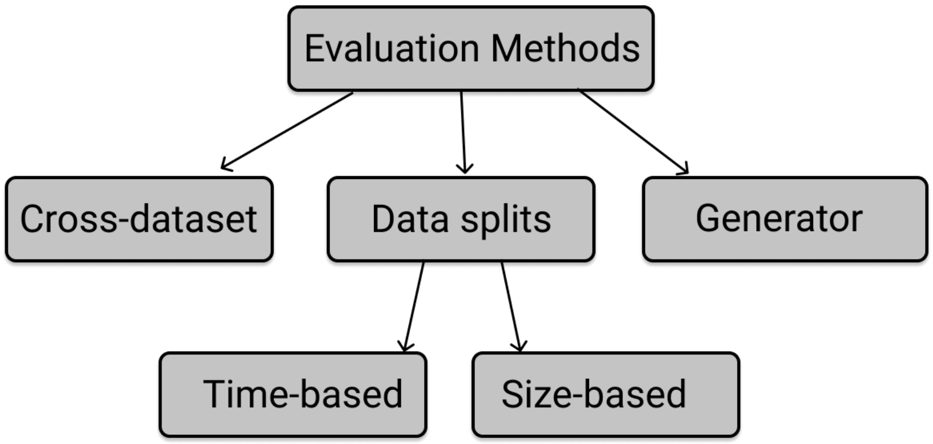


Figure 1.1: Project Overview

1.5 Research Contributions

The main contribution of the thesis is providing several ways to evaluate machine learning models in cybersecurity applications and demonstrating them with various datasets.

In the first part, we followed the conventional way to evaluate the ML models to match or outperform the current reported results in different NIDS datasets. We are designing and tuning the hyperparameters of a deep learning architecture to achieve such results. As we question those high results, we propose a different way to evaluate the same models by using cross-dataset evaluation. The results showed that all models failed to learn. After investigating the problem, we found mismatch in the definitions in one of the features. This required a recreation of the dataset to improvement the performance. And after we used the recreated dataset, the learning can of the intrusion can be transfer for the binary classification problem. Then, we want to test the same method with the multiclass classification. The problem is that one of the datasets that we use does not provide classifications of the attacks. This required a work to map the attacks into the classes thus allows the training and testing with cross-dataset evaluation method. The results show that the model just learned the normal label and failed to learn from the attacks (except the denial of service attacks — DoS). We also show a framework for one potential application of this method is to be used for automatic signature generation.

For the second part, we provided different evaluation strategies by giving five different ways of splitting the data for the training and testing and two different criteria of splitting - the total is 10 different ways of evaluation from this method. We examine these evaluations in malware detection with two known deep learning models. We analyse the results for different types of malware. We followed this by providing our own designed model that gives better results compared to the examined models.

In the last part, we design a way to evaluate a problem where the data for the problem can be generated algorithmically—the example of such data can be found in cryptanalysis. We examine the RSA algorithm security with this method. The work of this part is not to break RSA but to give it as a case study with the method. Nevertheless, some results of this part might be used as the start of an attack based on ML methods.

For reference, the proposed evaluation methods in the thesis, with the chapter numbers, are as follows:

1. Cross-dataset evaluation (Chapter 3 and 4)

2. Splitting evaluation (Chapter 5, 6, and 7)

3. Generator evaluation (Chapter 8)

1.6 Publications

During the work in this research, we published three peer-reviewed papers. Here is the list with an overview of each one:

1. **Said Al-Riyami, Frans Coenen, and Alexei Lisitsa. "A Re-evaluation of Intrusion Detection Accuracy: Alternative Evaluation Strategy." Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018. APA**. This paper examined the conventional ways of evaluating machine learning models in network-based intrusion detection systems. It provided matched or exceeded results from current known results with three different NIDS datasets. Then it proposed the use of the cross-dataset evaluation technique. When the same machine learning models are examined with the cross-datasets method, all models fail. The work of this paper is presented in Chapters 3 and 4.

2. **Said Al-Riyami, Alexei Lisitsa, and Frans Coenen. "An efficient way to deal with algorithmically generated data in deep learning", "Proceedings of the 2019 Emerging Technology Conference", Editors: M.K. Bane and V. Holmes, ISBN 978-0-9933426-4-6**. This paper proposes the use of generator evaluation to evaluate the algorithmically generated data problem. The paper examines resource consumption compared with a fixed list method. The paper gives the RSA algorithm as a case study. The work of this paper can be found in Chapter 8.

3. **Said Al-Riyami, Alexei Lisitsa, and Frans Coenen. "Cross-datasets evaluation of machine learning models for intrusion detection systems", "Proceedings of Sixth International Congress on Information and Communication Technology", Editors: Yang, X.-S., Sherratt, S., Dey, N., Joshi, A. (Eds.), ISBN 978-981-16-2101-7**. This paper is a follow-up to the first. The paper provided a fix that improves the performance of the ML models in NIDS with cross-dataset evaluation. The paper goes further to examine the multiclass classification. The work of this reported in this paper can be found in Chapter 3 and 4.

The work on splitting evaluation has not been published yet (Chapters 5, 6, and 7). The dataset that has been used to examine the evaluation method was recently published. The paper is in preparation.

1.7 Structure of Thesis

The thesis is organised as follows:

- **Chapter 2:** presents a review of the current evaluation method of machine learning applications in cybersecurity. The terms and description of this chapter will be the foundation for the rest of the thesis.

- **Chapter 3:** examines the conventional evaluation method in ML with three different NIDS datasets. We will match or exceed the reported results on these datasets and question the evaluation method used. We propose the use of a different method called cross-dataset evaluation. The result is that all the models failed. We provided a fix that improves the performance.

- **Chapter 4:** examines multiclass classification with cross-dataset evaluation. The results show the models learn the normality of the traffic, not the actual intrusion.

- **Chapter 5:** proposes five different ways to split the dataset for training and testing. We used a recent malware dataset and two known deep learning models. The splits on this chapter are based on the duration. We analyse the results based on different types of malware.

- **Chapter 6:** similar to the previous chapter, but the criteria of the splitting are based on the size (number of samples), not the duration.

- **Chapter 7:** proposes a designed deep learning model that performs better in different splitting-based evaluation methods. We compared the result with the previous two chapters.

- **Chapter 8:** provides the proposal of the generator evaluation that is suitable for the algorithmically generated data problems. The chapter compares the method with the use of a finite list. The chapter provides the RSA algorithm as a case study.

- **Chapter 9:** summarises the research, the main finding, and suggestions for future work.

1.8 Thesis Repository

Many results were generated during the work of this research. A big part of the work is not included in this text, including the detailed results, codes, figures and machine learning models. We will include them in a repository for easy access at the following URL:

`https://drive.google.com/drive/folders/1X6BgffCbG5bdJ4pUuAZRypbZugtlgCXd?usp=sharing`

Chapter 2

Evaluation Methods of ML Models in Cybersecurity

2.1 Chapter Overview

This chapter discusses the evaluation methods used in machine learning for application in cybersecurity. The term *'evaluation'* includes several underlying concepts like how to split the data, what is included in training and testing, and what the scoring metrics are. The chapter will also give an overview of the main machine learning algorithms and the datasets used. It will also highlight the main hardware and software tools used.

2.2 Evaluation

In the context of this thesis, we refer to the term evaluation as

The steps that are taken in the development stage of the machine learning model starting from processing of the dataset ending with performance metrics of the model to understand the performance of the model in the deployment stage.

We try to have a model that has less bias and is more accurate in the real world. There might be other goals of evaluation, like model selection and hyperparameter tuning of the model. Note that we are dealing with a classification task of supervised machine learning. In the development of the stage of machine learning, there are many steps that we can follow. The main ones that affect the evaluation are:

- How do we decide the training and testing phase of the evaluation process?
- How do we split the datasets?
- Which splits will go to the training and which ones for the testing?
- After the training, what are the scoring metrics used to measure the model's performance?

We can look at the process of evaluation from four different dimensions:

1. **Stages**: There are two main phases of creating the model: (1) Training (where the model learns from part of the data) and (2) Testing (where the model tries to predict based on unseen data). Typically, we want to check how the model is performing during the testing phase.
2. **Data-Splitting Methods:** How we can split the dataset? Furthermore, which part of the data is used in training, validation (if any) and testing. There are two popular methods: The holdout method and K-fold cross-validation. We will discuss the splitting in more detail later.
3. **Scoring Metric**: is used to measure the result (Accuracy, Confusion Matrix, Precision, Recall, Precision, F1 score, AUC, Loss, Root Mean Squared Error (RMSE), etc.). We can also have external metrics like processing time (How long it takes to train, retrain and deploy the model) and the model's size (and how many parameters it contains).
4. **Overfitting and Underfitting**: The results from the training stage should be close to the results of the testing stage. Overfitting tends to memorise the training data and fails in the real world. Underfitting tends to not learn from the training data.

More details of these evaluation dimensions and the option that we can choose during this process will follow.

2.2.1 Stages of Evaluation

At each stage of evaluation, the data used is different. The distinct separation of each stage is the goal of using the data. Each stage will serve a purpose in the process of creating a model. The list of the stages is as follows:

1. **Training**: The data used in the training stage aims to develop the model and make it learn from the distribution of the data. In a typical setting, this stage will contain most of the data from the dataset. It can take 70% or more of the data, depending on the size of the dataset.

2. **Testing**: The goal of this stage is to understand how the model will perform in the deployment. The data from this stage should not be used in training the model. Instead, it should be an independent validator for the performance of the trained model.

Some datasets will define what part of the dataset should be used for the training and testing.

2.2.2 Data-Splitting Methods

The idea of data splitting is how to distribute the data between the training and testing phase. The typical goal of the splitting method is to maximise the learning from the training phase while keeping a more realistic performance measure from the testing one. There are two main ways to split the data: (1) Holdout method and (2) K-fold cross-validation method.

(1) Holdout Method

In this method, we split the data into different parts; each has its purpose. The figure 2.1 shows a simple illustration of this method. The following is common to split the dataset [34, 7]:

- **Training set**: Part of the data that the model will use to learn from during the training phase. The model will learn the distribution of the data just from this portion of the data. This will represent most of the data from the dataset. The bigger the dataset, the more percentage of the dataset is given to the training set.

- **Testing set**: The data that will be used to test the performance of the model after the training phase. This will be done after we finished the training phase of the model. We want to avoid using any part of this data in the training phase, including the hyperparameter tuning of the model. The reason being is to avoid creating any bias.

- **Dev (development) set**: also known as the *'validation set'* or *'holdout set'*. This part is used during the training phase to tune the hyperparameters of the model. The distribution of the validation set should be the same as the distribution of the testing set [55]. This set helps to avoid overfitting while tuning hyperparameters by giving feedback on how the change affects the performance. We stop the training phase when the performance of the training set is as good as the validation set performance. In other words, we stop the training when the error in the validation set start to increase [7].

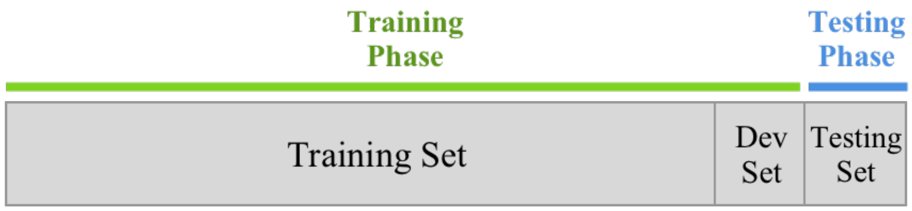


Figure 2.1: Simple illustration of Holdout method

(2) K-fold Cross-Validation Method

Originated from [31]. The following steps are how it works:

1. Divide the dataset into K number of fold or group.
2. Make one group a validation group and the rest of the groups a training set. Calculate the performance score.
3. Repeat Step 2 until all groups are used as a validation set.
4. The result is the average of all scores.

Figure 2.2 shows an illustration of this method when the number of folds is equal to five (K=5). The testing phase with k-fold has two variations. Some implementations will consider the results of the validation set to evaluate the model performance. This is normally done when we have a limited amount of data. Other implementations will have a separate testing set for the final evaluation of the model to prevent data leaks from training [18].

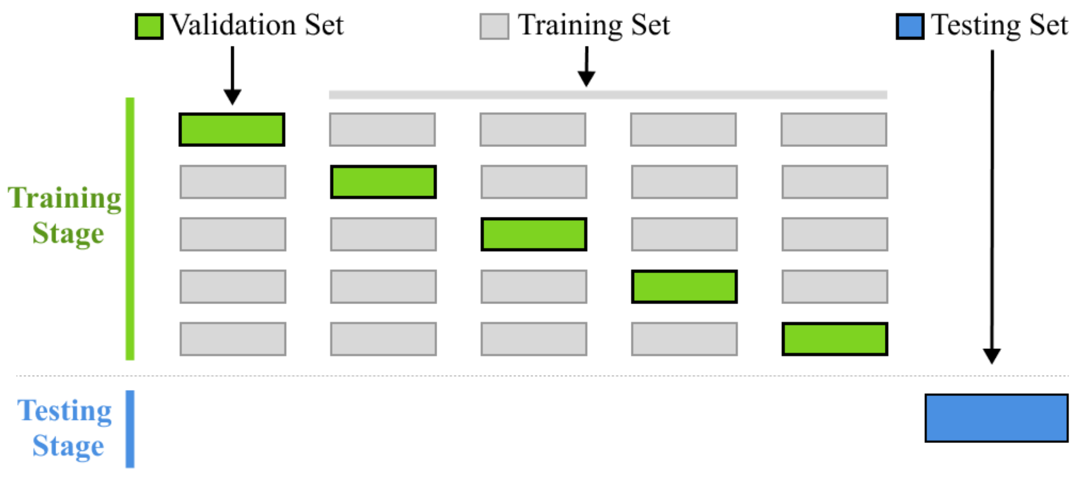


Figure 2.2: Simple illustration of K-fold cross-validation. K=5

There are different variations of the k-fold cross-validations. The following are some:

1. *Stratified K-Fold cross validator*: Each fold will have an equal balance of target/label values. This type is suitable for imbalanced data.

2. *Leave-one-out cross-validation (LOOCV)*: When the number of the fold k is equal to the number of the sample of the dataset n [63].

3. *Leave p-out cross-validation (LpOCV)*: Uses several p-observations as the validation set and the remainder of the data as a training set [74].

4. *Repeated Stratified K-Fold cross validator*: Repeats the k-fold several times and randomises the data in the fold for each iteration.

5. *Time Series cross-validation*: The K-fold validation assumes the data are independent and identically distributed (i.i.d.), so the time series data is not valid for this assumption. For that, there are other variations of cross-validation design for the time series, which will be discussed next.

Time Series Cross-Validation

We want to provide an overview of the time series cross-validation methods, as they have some similarities with the proposed splitting evaluations in later chapters.

The k-fold validation in time series can be categorised into two main categories: (1) Rolling Cross-Validation with a sliding window and (2) Rolling Cross-Validation with an expanding window. Figure 2.3 illustrates both types of Rolling Cross-Validation. In both methods, the validation set is fixed. The change is in how to deal with the training fold size.

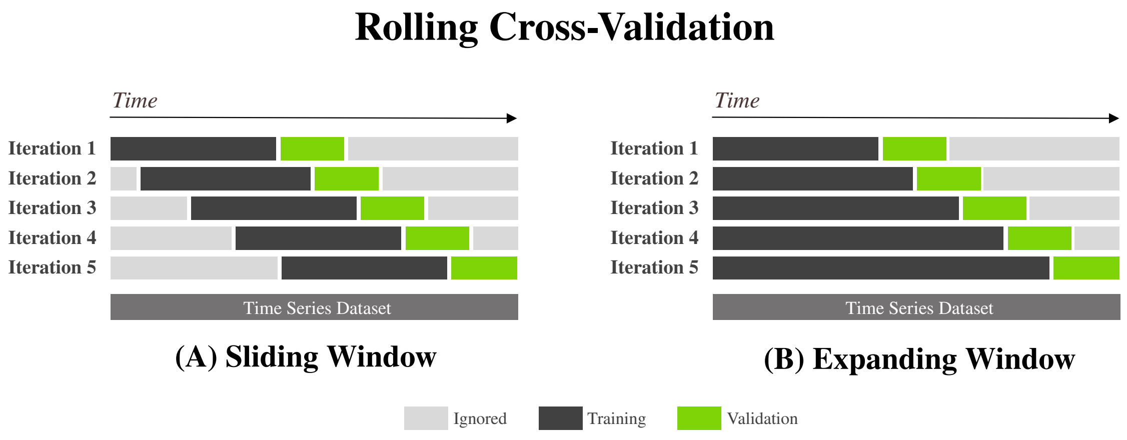


Figure 2.3: Rolling Cross-Validation: (A) Sliding Window, (B) Expanding Window

More details are as follows:

- *Rolling Cross-Validation with a sliding window:* The size of the training fold in this type is fixed for all the iterations. In each iteration, we slide forward in time. The fixing of the window size will give an equal evaluation for all the iterations since all have an equal validation size.

- *Rolling Cross-Validation with an expanding window:* The size of the training fold in this type expands with each iteration. The method will give an understanding of the effect of accumulating more data in training over time.

There is also the concept of a gap. The idea of the gap is to give some space between the training and the validation set. The goal is to increase the independence between the training and validation set [11]. Figure 2.4 illustrates the use of the gap in rolling cross-validation.

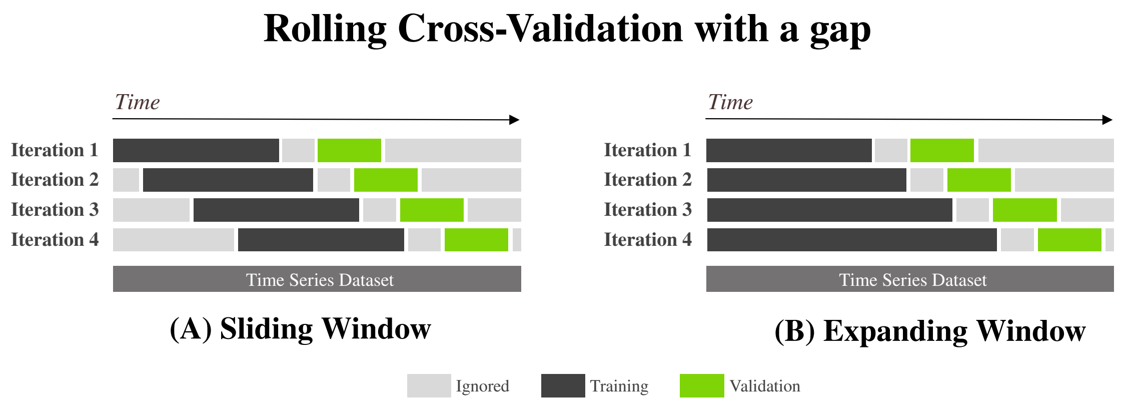


Figure 2.4: Rolling Cross Validation with a gap: (A) Sliding Window, (B) Expanding Window

One downside of Rolling Cross Validation is that not all the data is utilised for the training.

2.2.3 Scoring Metrics

There are many scoring metrics to measure the performance of a machine learning model classifier. For our work, the following sections describe the relevant metrics. In general, if the score is 50% or less, that might indicate the result of learning is just a purely random guess.

Confusion Matrix

A visualised table consists of the actual performance values of the classifier in a certain order. The table compares the model prediction with the ground truth/label. Table 2.1 shows confusion matrix values in a table. Assuming that we have a binary classification problem for an attack presence, the table will consist of the following four values:

1. *True Positive (TP)*: The number of times that the model correctly predicted that there is an attack.

2. *False Positive (FP)*: Also known as type I error. This is a false alarm. The model predicated there is an attack, but there is no attack.

3. *False Negative (FN)*: Also known as type II error. There is an attack, but the model did not predict that there is an attack.

4. *True Negative (TN)*: There is no attack in the data, and the model prediction is the same.

Table 2.1: Confusion Matrix for Binary Classification

	Predicted condition		
Actual condition		Positive	Negative
	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

Accuracy

This might be the most used metric. The accuracy calculates the ratio of the total number of correct predictions by the total number of predictions. Equation (2.1) shows how it is calculated.

$$Accuracy = \frac{TruePositive + TrueNegative}{TotalPrediction} \tag{2.1}$$

Precision

Also known as positive predictive value. The precision metric tries to answer the question: what proportion of positive identifications was actually correct? [16]. Equation (2.2) shows how it is calculated.

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} \tag{2.2}$$

Recall

Also known as sensitivity or true positive rate. Recall metric tries to answer the question: what proportion of actual positives was identified correctly? [16]. Equation (2.3) shows how it is calculated.

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} \tag{2.3}$$

Specificity

Also known as false positive rate. Equation (2.4) shows how it is calculated.

$$Specificity = \frac{FalsePositive}{FalsePositive + TrueNegative} \tag{2.4}$$

F1 Score

Also known as F-measure or balanced F-score. It is a harmonic mean of precision and recall. Equation (2.5) shows how it is calculated.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.5}$$

Area Under the Curve (AUC)

Also known as *'area under the ROC Curve'*. The receiver operating characteristic curve (ROC) shows the performance of the classifier by plotting Recall (true positive rate) and Specificity (false positive rate) in a curve with a moving threshold that covers all possible classification thresholds. AUC is the area under the ROC curve. A higher value of AUC indicates a better-performing classifier. The result of 1.0 of AUC means a perfect score. Figure 2.5 shows an example of the ROC, a classifier that scores an AUC of 0.79.

Multiclass Averages

For a multiclass classification problem, there are different ways to calculate the average of the score [59]:

- **Micro Average**: Calculate metrics globally by counting the total true positives, false negatives and false positives.
- **Macro Average**: Calculate metrics for each label and find their unweighted mean. This does not take label imbalance into account.
- **Weighted Average**: Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label).

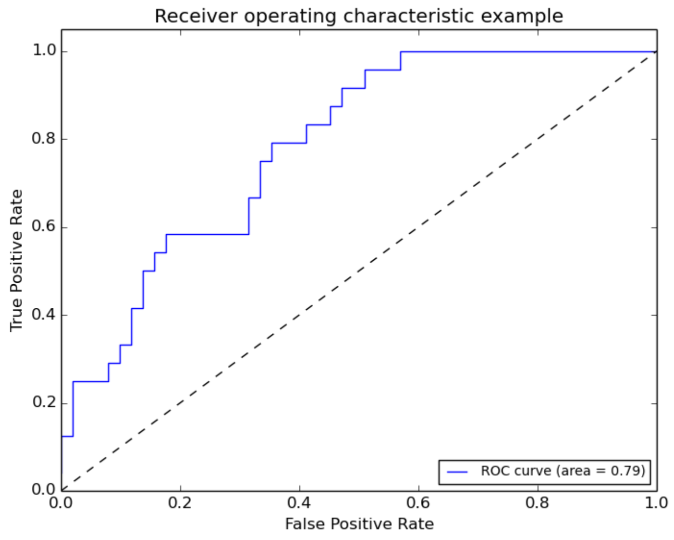


Figure 2.5: ROC curve of with AUC=0.79 [67]

2.2.4 Overfitting and Underfitting

Overfitting is when the model performs well in the training set but does not in the validation set. This might be due to high variance and low bias in the model error. Underfitting is when the model cannot learn from the training set. This might be due to high bias and low variance in the model. We want the model to generalise the distribution of the data. This concept is referred to as Bias-Variance Tradeoff. Figure 2.6 is an illustration between the bias and the variance with the model complexity chart.

2.3 Machine Learning Algorithms

Machine learning (ML) is the study of computer algorithms that improve automatically through experience and by using data, [53]. The three main types of machine learning are:

1. **Supervised learning**: The learning in this type of problem is from labelled data.
2. **Unsupervised learning**: The data used to learn from this type is not labelled.
3. **Reinforcement learning**: The learning in this type is based on the feedback given to the agent in a certain environment after acting.

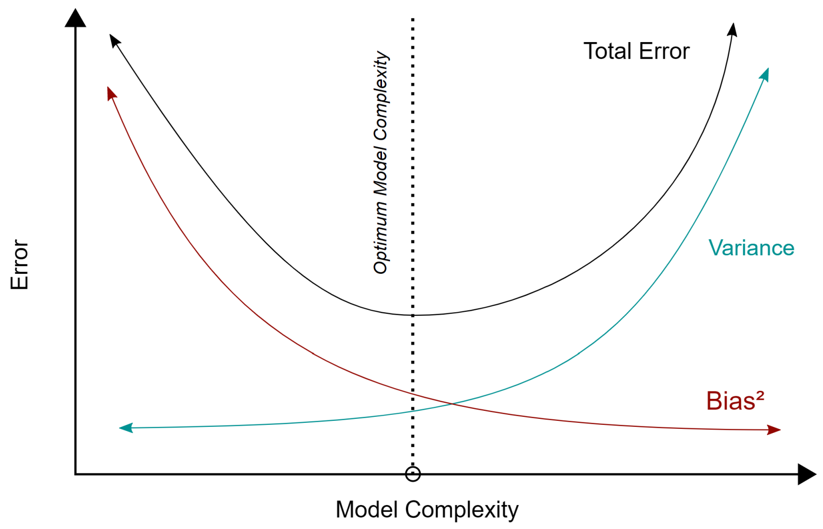


Figure 2.6: Bias and variance as a function of model complexity [6]

Supervised learning can have two main types of problems, as follows:

- **Classification**: This type of problem tries to predict discrete class labels. For example: does an image have a picture of a cat?
- **Regression**: This type of problem tries to predict a continuous quantity. For example, predicting a stock price.

This thesis deals with supervised learning, in particular, a classification problem. A classification problem can have two labels. In this case, it is called a *binary classification problem*. On the other hand, a *multiclass classification problem* is deal with more than two labels.

Recently, the use of deep learning (usage of artificial neural networks with a higher number of layers than usual) has become popular in areas such as computer vision, speech recognition, natural language processing, audio recognition, social network filtering, machine translation, bioinformatics and so on. This has happened because of advancements in the processing power available, particularly in the Graphics Processing Unit (GPU), and the increase in the amount of data we have. The major advantage is the model's ability to abstract information from the data by drilling down into the data to a greater extent than was previously possible.

Many machine learning algorithms are used in this thesis. Some are from the classical ML algorithm, and others are from deep learning models. In this section, we will give an overview of the main algorithm used.

2.3.1 Decision Tree (DT)

A type of supervised machine learning algorithm that follows a flowchart-like structure called nodes to make prediction. The decision tree node consists of three types:

1. *Root:* A starting node that does not have a parent node. The root node will give two child nodes. The splitting between the child nodes is based on a condition.

2. *Internal node*: Similar to a root node, but it has a parent node.

3. *Leaf:* This will be the final node in the tree which will contain the prediction.

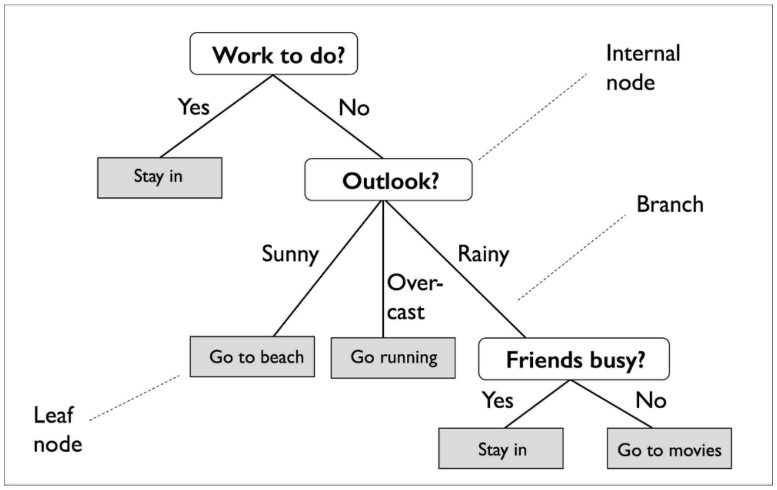


Figure 2.7: Decision tree learning example [66]

Figure 2.7 shows a simplified version of decision tree learning. There are variations of machine learning algorithms that use decision tree learning with ensemble learning. Here are some of them:

1. **Random Forest (RF)** [35]: a type of bagging ensemble algorithm. Instead of having one tree, several subtrees will be created. The decision will be based on voting between the trees.

2. **AdaBoost with DT** [29]: stands for *Adaptive Boosting*. It is a type of boosting ensemble algorithm. It builds a strong classifier by combining multiple poorly performing classifiers so that you will obtain high accuracy [54].

3. **Gradient boosting Trees** [30]: a type of boosting ensemble algorithm. Similar to AdaBoost in terms of build based on weaker classifiers, but it differs in the method.

2.3.2 Artificial Neural Network (ANN)

The artificial neural network (ANN) consists of three types of layers: input layer, hidden layers and output layer. The input layer is linked with the input features X and the output layer is linked with the target value y. Each layer consists of several nodes. The nodes in the hidden layer are connected with the previous and next layers. The link between the nodes consists of bias and weight. Typically, at the end of the hidden layer, there is an activation function. The learning and updating of the weights in the network will happen in a process called backpropagation. The learning and updating are done in several passes in the dataset called an *epoch*. A network with many hidden layers is typically referred to as *deep learning*. Figure 2.8 shows a simplified illustration of multilayer neural network architecture.

There are other architectures of the neural network as follows:

- **Convolutional Neural Network (CNN/ConvNet)**[46]: designed to learn from visual images. A typical CNN architecture will consist of Convolutional layers, Pooling layers, and Fully connected layers.

- **Recurrent Neural Network (RNN)**: designed to learn from sequential data, like time series data. The typical RNN architecture might have a vanishing gradient problem. To resolve such a problem, different types of network cells proposed like Long short-term memory (LSTM) [36] and Gated Recurrent Units (GRU) [13]

2.3.3 Other

Other machine learning algorithms used in the thesis include Logistic Regression, Naive Bayes, Linear Support Vector Machine (SVM) [88], K-Nearest Neighbours Algorithm (k-NN), and Discriminant Analysis.

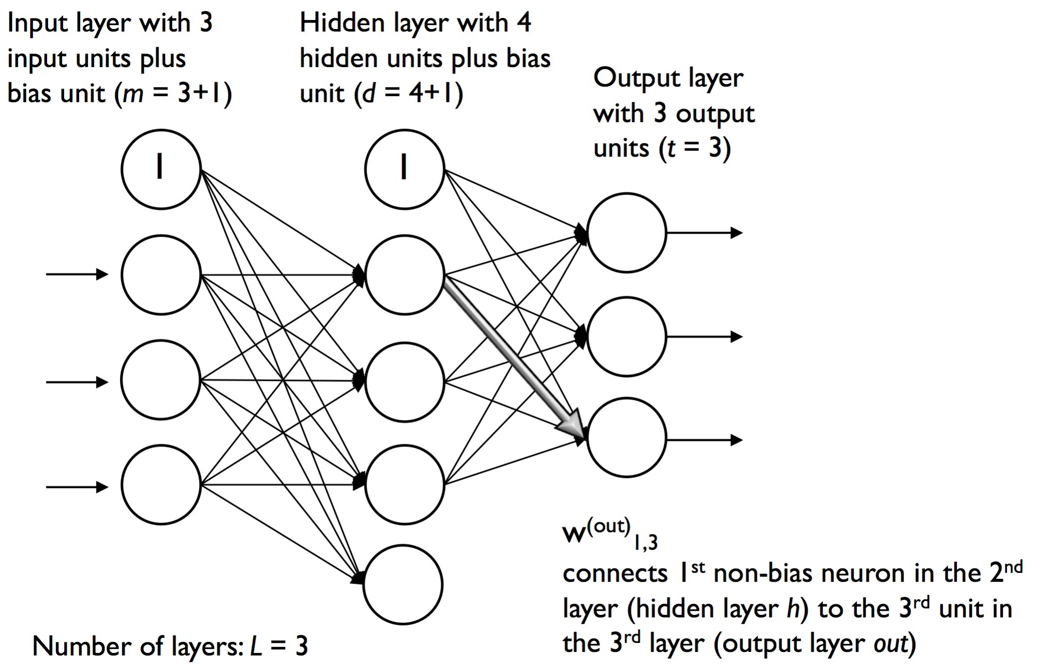

Input layer with 3 input units plus bias unit (m = 3+1)
Hidden layer with 4 hidden units plus bias unit (d = 4+1)
Output layer with 3 output units (t = 3)
1
1
$w^{(out)}_{1,3}$
connects 1st non-bias neuron in the 2nd layer (hidden layer h) to the 3rd unit in the 3rd layer (output layer out)
Number of layers: L = 3

Figure 2.8: Multilayer neural network architecture [54]

2.4 Application of ML in Cybersecurity

Cybersecurity applications can be categorised into two broad categories: offensive and defensive security. And the application of machine learning in cybersecurity will follow the same categories. There are many examples of the application of machine learning in cybersecurity. Some are:

1. **Spam Detection:** A common use of machine learning is to catch spam emails [17].

2. **Intrusion Detection System (IDS):** With different types: Host Intrusion Detection (HIDS), Network Intrusion Detection (NIDS), or Web Application Intrusion Detection. It is used to detect the intrusion by learning form their behaviour [3].

3. **Penetration Testing:** Use machine learning to find a vulnerability in the system [52].

4. **Steganography:** Uses machine learning to detect hidden stenography data or hidden

malware in the deep learning model by using stenography [12, 90].

5. **Malware detection:** Sses machine learning for the detection of malware [4].

6. **Adversarial Machine Learning:** Tries to find vulnerabilities in machine learning models [70].

In this thesis, we consider in three application areas: Network Intrusion Detection System, Malware Detection, and Cryptanalysis.

2.5 Datasets

During the work of this thesis, we used the following public datasets:

1. **NSL-KDD** [62]: This is a fixed version of a famous KDD Cup dataset. It is based on computer network traffic.

2. **gureKDD** [61]: A newer version of KDD dataset built from scratch. It is based on computer network traffic.

3. **Kyoto+2006** [80]: A network dataset that started from 2006 and ends in 2015. It contains normal and malicious network traffic

4. **SOREL-20M** [33]: Largest available public malware dataset as of writing. It was published at the end of 2020. It contains 20 million samples of malware.

2.6 Implementation

Here is a description of the hardware and software that we used to implement the experiment in the project. Different experiments might use a different tool.

2.6.1 Hardware

Most of the experiments are implemented in the workstation with the following configurations:

- CPU: AMD Ryzen$^{\text{TM}}$ Threadripper (12 cores and 24 threads).

- RAM: 64 GB.

- GPU 1: NVIDIA TITAN V (12 GB HBM2 memory).
- GPU 2: NVIDIA® GeForce® RTX 2080 Ti (11 GB GDDR6 memory).

Some experiments are executed in the Amazon AWS cloud service in the cases when there is a constraint with the workstation resources.

2.6.2 Software

The programming language that we used is Python. And the main tools and libraries that were used are:

- *pandas*: Used to read the datasets and preprocess the training.
- *Scikit-learn*: Contains many classical machine learning algorithms.
- *TensorFlow and Keras*: Used to create deep learning models.
- *PyTorch*: Also used to create deep learning models.
- *Matplotlib*: to make different visualisations.

2.7 Summary

This chapter provides the reader with an overview of different aspects of evaluation in machine learning models. First, the chapter explained the four different dimensions of the evaluation process: (1) stages, (2) data-splitting methods, (3) scoring metrics, and (4) overfitting and underfitting. Then, it gives an overview of the handout method and k-fold cross-validation. Next, the chapter highlighted the main concepts of machine learning that is a foundation for the following chapters. There is a list of various applications of machine learning in cybersecurity. The chapter ends by giving the dataset used and the hardware and software used for implementation. The next chapter will start with the first method of evaluation, called cross-dataset evaluation with a binary classification problem.

Chapter 3

Cross-Datasets Evaluation in Binary Classification

3.1 Chapter Overview

This chapter will discuss a common method used to evaluate the machine learning models in cybersecurity applications. The chapter focus on the Network Intrusion Detection System (NIDS) application with the binary classification problem. The chapter lists the best results in different datasets and tries to match or obtain better results. The next part of the chapter will introduce the cross-dataset evaluation method, examine the same machine learning models with the new method, and provide the analysis for the results.

3.2 Intrusion Detection System (IDS)

An intrusion detection system (IDS) is a common method used in defensive security. IDS can be software or hardware that tries to find an intrusion in the system. We can categorise IDS based on the target system as follows:

1. **Network-based Intrusion Detection System (NIDS)**: The data type here is computer network traffic [76].

2. **Host-based Intrusion Detection System (HIDS)**: The data type here is CPU instructions [47].

3. **Web Application Intrusion Detection System (WAIDS)**: The data type here can be HTTP messages or DOM objects in the browser [57].

We can also categorise IDS based on how it works, as follows [21]:

1. **Signature-based IDS**: The IDS have previous knowledge of a specific intrusion pattern in the system. Such a system needs a constant update on the signature to include newly discovered intrusions. The advantage of such a system is that it has a less False Positive rate. The drawback is that it cannot detect zero-day attacks.

2. **Anomaly-based IDS**: The IDS will check the normality of the network or the system with a statistical method. Once it detects a shift from the normality baseline, it will give an alert. It can also understand the behaviour of the attacks. The advantage of such a system is that the fact it can detect zero-day attacks. However, such a system is known for high False Positive alerts.

3. **Policy-based IDS**: The system work based on defined security rules. Once any security rule is violated, it will give an alert.

4. **Hybrid IDS**: In practice, we want the IDS to include different methods of detection.

The work present in this chapter and the next chapter is focused on a Network-based Intrusion Detection System (NIDS). The dataset consists of computer network traffic.

3.3 Current Evaluation method in NIDS

The use of NIDS with the machine learning can be approached as Supervised Learning and Unsupervised Learning. Supervised Learning deals with the data that contains the labels and tries to map the input X to the target output y. For example, the data can contain the label of the intrusion or normal traffic. Unsupervised Learning does not have a label. Typically, it tries to cluster the data into different categories.

Most of the publications on machine learning in intrusion detection follow the same procedure when evaluating the performance:

- Select an available public dataset.
- Use the dataset and follow one of the methods mentioned in data-splitting methods. This includes holdout or k-fold cross-validation methods.

- Report the performance results by using one or more scoring metrics.

The drawback of this setting is that we do not know how much the model learns about the intrusion compared to the particular computer network environment. It might be just a result of the overfitting of a particular dataset that has been created within a particular computer network architecture with a particular way to label the intrusion, as the use of the honeypot. This may make the reported results about the accuracy of the intrusion detection overly optimistic and suspicious looking. However, most of the reported results reach 99% precision [69], which is the discussion of this section.

3.3.1 Experiment with the method

The starting point of the investigation is to examine the current evaluation method. Then, we will follow the holdout set method to split the data. We will use three known NIDS datasets with different machine learning algorithms.

Most of the time, the use of traditional machine learning algorithms for NIDS obtain higher results than deep learning algorithms. For that, we want to demonstrate that we can use most of the deep learning models and try to reach the same performance of classical machine learning or even more. This section aims to show that obtaining a high-performance model is not hard based on the current method of evaluating the models in NIDS.

3.3.2 Methodology

We will use three different datasets: Kyoto2006+ [80], NSL-KDD [62], and gureKDD [61]. We will check models accuracy that has been reported in those datasets in the literature. Then, we will run several machine learning algorithms. We will do hyperparameter tuning that gives the best results.

The training and testing are done within the same dataset. The split is around 80% for training and 20% for testing (mostly given by the dataset). Then, we will calculate the performance only based on the result of the testing part. The performance can be measured by the ratio of the total number of correct predictions by the total number of predictions. But to avoid any Accuracy Paradox that might be because of the distribution in imbalance data, we use the F1 score. The F1 will balance between Precision and Recall (check Chapter 2).

The models that we used are Decision Tree (DT) [94], Random Forest (RF) [35], Logistic Regression (logit) [89], K-NN [26], Artificial Neural Network (ANN), and Recurrent Neural

Network (RNN) with Long Short-Term Memory (LSTM) [36] and with Gated Recurrent Unit (GRU) [13].

Kyoto+2006 Dataset

Kyoto 2006+ is a dataset that has been generated for NIDS [80][81]. They used honeypot traffic as a source of ground truth about the malicious activities. The dataset contains 22 features. There are three classes of labels for this dataset: normal, known attack, unknown attack. We will convert the problem to be binary, where 0 means normal traffic and 1 means malicious traffic. We will follow the same split and preprocessing of this dataset to the one reported in for comparison [2].

NSL-KDD and gureKDD Datasets

The KDD cup 1999 dataset is one of the most-known datasets for NIDS. Different variations were created based on this dataset because of a reported problem [51][48]. We will use NSL-KDD and gureKDD datasets. NSL-KDD is a refined version of KDD [84]. The gureKDD dataset is generated from scratch with the same purpose as KDD [61].

3.3.3 Results

Table 3.1 shows the results of the three datasets.

Table 3.1: Evaluation with the current strategy - F1 Score

Algorithm	Kyoto+	NSL-KDD	gureKDD	NSL-KDD (multi)
DT	97.64%	99.46%	99.92%	99.36%
RF	98.19%	99.56%	99.94%	99.30%
logit	93.57%	93.69%	97.34%	95.27%
k-NN	97.16%	99.15%	99.92%	98.07%
ANN	98.13%	99.24%	99.46%	98.25%
LSTM	97.91%	99.21%	99.42%	98.45%
GRU	98.03%	99.17%	99.31%	98.40%

Kyoto+2006

The best accuracy result in [2] is ≈84.15%. And after we calculated their F1 score, we got ≈87.56%. Another reported work with the same datasets can be found in [77]. They used a stack of Artificial Neural Network (ANN) layers. Their best reported result showed when they used 1000 hidden neurons with an F1 Score of ≈97.50%.

After we experiment with different models, the result we obtained is shown in Table 3.1. We reach an F1 score of ≈98.13% by using Artificial Neural Network (ANN) model. The best result we got is ≈98.19% with a Random Forest (RF). Both results are better than the reported results.

KDD Variation Datasets Experiments

Many studies were done by using the NSL-KDD dataset. For example, the paper [62] reported an accuracy of 99% for both binary and multiclass classification by using SVM. This paper [65] reached 99.6% with K-NN. The paper [95] reported accuracy of 97.09% by using a Recurrent Neural Network (RNN).

After we experiment with different models, the result we obtained is shown in Table 3.1. For binary classification of both NSL-KDD and gureKDD, most of the models give ≈99% of the F1 score. We also show that this result can be obtained for the multiclass problem. To demonstrate that, we used 40 classes of different attacks, including a normal traffic label. The result is shown in Table 3.1. You can notice that the results are around ≈98%-99%.

3.3.4 Analysis

Table 3.1 shows that our result matches the current reported results or, sometimes, exceeded them by using the same evaluation strategy. In addition, we notice that traditional machine learning algorithms show similar results to deep learning models.

The most important part is that those results are very high. Such results are suspicious for a real model and practical use. Therefore, we need a different way to test machine learning models for NIDS. In the next section, we will propose one way of doing that.

3.4 Cross-datasets Evaluation

As we can see from the previous section, we can achieve high results. Many models created achieved 99% of the F1 score. Based on the current way of assessing the NIDS models, can we stop here and use the learned model in NIDS? Does the model we created represent attack understanding learning? Or is it just an overfitting based on the dataset representing a certain type of computer network with a particular setting?

To address these questions, we propose another evaluation strategy. We propose using two different datasets that have the same domain but a different distribution of traffic. Both should share the same purpose and features but are generated from two different computer networks. In this way, we will make sure that the model does not learn just the distribution of traffic in the network but should generalise the understating of what we want to find. In our problem, we want to learn the abstract behaviour of the attack that allows us to find the attack in a different network or find a new zero-day attack with the same understanding.

The cross-datasets evaluation method tries to make the evaluation of ML-based NIDS more practical. The idea is to use two different datasets generated from two different computer networks. We train in one dataset and test the model in the other (and swap between them) with the same set of features. This gives a better understanding of the actual performance of the machine learning models when the computer network changes. That allows us to see how our model can adapt to the change in network architecture. By using this method, we also gain insight into the quality of the datasets used in training. If we get high performance, the dataset used in training contains more information about the intrusions. Figure 3.1 shows an illustration of the method.

We use the F1 score as a metric for evaluating the performance. This will help avoid the accuracy paradox that might be there because of the imbalance of the data [92].

Cross-Dataset Evaluation has two main advantages, as follows:

1. **Evaluating the model practicality**: Any model should perform well when the computer network changes.

2. **Evaluating the quality of the dataset**: The performance of the model in the testing dataset indicates the quality of the training dataset.

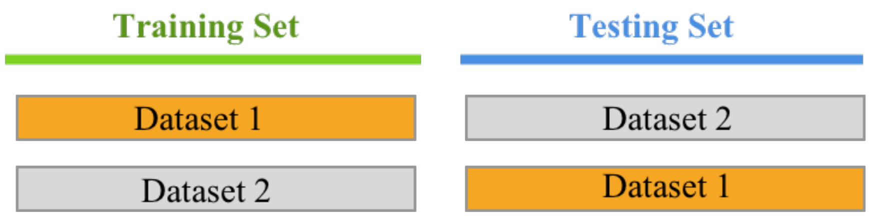


Figure 3.1: Cross-Dataset Evaluation Method

3.4.1 Methodology

We will use NSL-KDD and gureKDD to experiment with the cross-datasets evaluation method. Both datasets represent different computer networks, which will give us a different distribution of traffic and the same set of features. Furthermore, both datasets are available for public download. The training will be done on one dataset and the testing in the other, and vice versa. We take one dataset as a whole in the training and another as a whole in the testing. We can use a small portion of the training set for validation during the training phase.

We will follow the same process as before, where we will use traditional machine learning and different deep learning models. The problem will be binary classification, but the idea can also be applied to multiclass problems (next chapter). The importance of the F1 score will show in the results. For that, we only consider the F1 score as a performance indicator.

3.4.2 Results and Discussion

During the model training, all of them achieved an accuracy of ≈99%. But we will only report the results during the testing phase. Table 3.2 shows the results when NSL-KDD is used for training and gureKDD is used for testing. Table 3.3 shows the result when we use gureKDD for training and NSL-KDD for testing.

Table 3.2: New Evaluation Strategy - NSL-KDD(training), gureKDD(testing)

Algorithm	Accuracy	F1 Score
Random Forest	97.65%	36.08%
ANN	80.85%	9.59%
LSTM	87.21%	17.5%

Table 3.3: New Evaluation Strategy - gureKDD(training), NSL-KDD(testing)

Algorithm	Accuracy	F1 Score
Random Forest	52.71%	3.62%
ANN	52.16%	5.19%
LSTM	52.78%	7.40%

As you can notice from the results obtained, all models give a very low F1 score. The importance of the F1 score is shown in Tables 3.2 and 3.3. Even though the accuracy in some cases is high, the results of the F1 score are still low.

From the conventional way to evaluate the performance, the model gets a very high F1 score. However, when the same model is evaluated with a different strategy, the models fail to adapt. The change in the computer network traffic makes the results of the models just random guesses. In this case, the model did not carry out real learning of intrusions. We need a further investigation into the performance drop shown in these results (next section).

This way to measure the performance of NIDS models will not only benefit the way we create the machine learning models but also the datasets that we have and the way we created them. From this angle, a good dataset should give trained model a better results once it is tested with different dataset. Furthermore, it should represent attack/malicious understanding and adapt to the change in the network traffic.

The general way to use the method is to train in a certain dataset and test in another dataset that has been generated from different network traffic. However, we can have different variants of the cross-datasets evaluation method as follows:

1. When we have categories of attack shared, and we want to evaluate the performance of finding each category of attack. In other words, the multiclass classification problem. The work in the next chapter will focus on this method.

2. Evaluating the finding of a zero-day attack. Can the model created in one dataset detect an unseen attack from another dataset? In other words, we want to see the attacks that are not shared between two datasets and test the machine learning models for such attacks.

3. Multiple dataset learning: we can also make the model train from multiple datasets and test it in one or more datasets.

We should avoid training and testing our model from traffic or data on the same computer network, even if the datasets have different names. If the result is below 50% of the F1 score, this means the model cannot adapt to the new dataset. If above 50%, it means that the model is starting to learn something about the intrusion. Since the results of the F1 score is 36% or less, this means that the learner has learned less than a random guess.

3.5 Investigation and Correction

This section is an investigation into the drop of F1 score results with the experiments in cross-dataset evaluation. We will use the feature importance to understand the effect of each feature in the datasets. We used tree-based classification methods. We added AdaBoost Classifier, Gradient Boosting Classifier beside the use of Random Forest Classifier.

The initial results of each model by using cross-datasets evaluation are shown in table 3.4. The table shows the results of a different number of estimators.

GradientBoost result is better than other models reported in the previous section [69], but still, it fails when considering the F1 score. All F1 score is less than 38%. This means that the models still do not learn about the intrusion.

		NSL-KDD → gureKDD		gureKDD → NSL-KDD	
Model Type	Estimators	Accuracy	F1 Score	Accuracy	F1 Score
Random Forest	500	96.87%	29.67%	52.73%	3.62%
	1000	97.13%	31.48%	52.72%	3.62%
	2000	96.30%	26.23%	52.73%	3.61%
	5000	96.31%	26.32%	52.73%	3.61%
	10000	96.28%	26.19%	52.73%	3.61%
AdaBoost	500	74.60%	13.76%	53.40%	6.73%
	1000	89.20%	26.17%	52.97%	4.98%
	2000	85.57%	21.49%	52.97%	5.14%
	5000	88.76%	23.66%	53.28%	6.49%
	10000	87.28%	21.46%	53.38%	6.98%
GradientBoost	500	97.27%	37.55%	97.27%	37.57%
	1000	97.25%	28.72%	97.28%	28.50%
	2000	97.41%	27.38%	97.41%	27.55%
	5000	97.15%	24.98%	97.37%	27.08%
	10000	97.37%	27.05%	97.28%	26.21%

Table 3.4: Result before the fix (Train → Test).

The comparison of the feature importance between the AdaBoost Classifier and GradientBoost Classifier shown in Figure 3.2 (NSL-KDD (training) → gureKDD (testing)) and Figure 3.3 (gureKDD (training) → NSL-KDD (testing)).

3.5.1 Feature Importance with Random Forest

We will use the Random Forest algorithm as another tool for feature importance investigation. This investigation aims to find the feature importance of the best and worst subtrees in the testing phase and analyse them. The detailed steps are as follows:

1. Run 5000 estimators/trees in the Random Forest Classifier for both the datasets as training. So, we have two Random Forest Classifiers, and each one has 5000 estimators/trees.
2. Check the performance estimators/trees inside Random Forest Classifier independently in the testing set using the F1 score.
3. Report the top features each of 10 best and worst performing estimators combined. The top feature is the sum of the importance value of each feature that estimators group.
4. Analyse each feature in both datasets.

3.5.2 Feature Importance Results

The Random Forest algorithm Feature Importance method results will be reported here in two parts: (1) NSL-KDD as a training set and gureKDD as a training set.

NSL-KDD (training) $\rightarrow$ gureKDD (testing)

- **Best Features**: diff_srv_rate, dst_host_srv_diff_ host_rate, dst_bytes, serror_rate, srv_serror_rate, dst_host_count, srv_rerror_rate, src_bytes dst_host_same_src_port_rate, and service.
- **Worst Features**:src_bytes, dst_host_srv_serror_rate , dst_bytes , diff_srv_rate , dst_host_count , serror_rate , srv_count , protocol_type , num_failed_logins and service.

gureKDD (training) $\rightarrow$ NSL-KDD (testing)

- **Best Features**: dst_host_srv_diff_host_rate, wrong_fragment, duration, diff_srv_rate, dst_host_serror_rate, src_bytes, same_srv_rate, dst_host_same_src_port_rate, dst_bytes, and count.

- **Worst Features**: wrong_fragment, src_bytes, dst_host_srv_diff_host_rate, dst_bytes, service, diff_srv_rate, count, protocol_type, dst_host_same_srv_rate, and srv_count.

There are three features common to all models listed above. The features are diff_srv_rate (% of connections to different services), dst_bytes (number of data bytes from destination to source), and src_bytes (number of data bytes from source to destination).

3.5.3 Analysis

After running the experiment and analysing the features, we found that there is a mismatch between both datasets on *service* feature naming. NSL-KDD contains 70 distinct labels of service, while gureKDD contains 24 labels. For example, the HTTP protocol service labelled as 'http' in gureKDD, but it has four different labels in NSL-KDD based on the used port number (http, http_2784, http_443, http_8001). We conjectured that this was the reason for the dramatic drop in performance in cross-dataset validation results reported in [69]. To check the conjecture, we modified the datasets to have a compatible service feature naming.

The service feature was labelled differently in both datasets. There is no standard to label the service feature. To match the label of this feature, we need the port number (sending and responding) and the protocol used in the transport layer. Since that pieces of information are not available in NSL-KDD, we decided to match the gureKDD to the one in NSL-KDD. Each service needs to be mapped carefully since there is no documentation from the original KDD on the meaning of each label. The service feature is more straightforward to change based on the specifications in [15]. Other labels need more understanding of the connection to decide on the meaning. The output of this process is a new version of gureKDD that is compatible with NSL-KDD. The new version is generated from the raw data of the gureKDD. The full details of the changes that we made with the new version of gureKDD can be found in the repository [check section 1.8].

3.5.4 New Results

After the changes are done to the gureKDD dataset, we run several machine learning algorithms. The machine learning models that we used are Decision tree learning, Random forest [35] (with a different number of trees), AdaBoost [29], Gradient boosting [30], Logistic Regression (LR), Naive Bayes, Linear Support Vector Machine (SVM) [88], K-Nearest

Neighbours Algorithm (k-NN), Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis(QDA), Artificial Neural Networks (ANN), Convolutional neural network (CNN), Long short-term memory (RNN LSTM) [36] and Gated Recurrent Units (RNN GRU) [13].

This change makes the performance increase dramatically in both datasets. The results of this fix show in Table 3.5 with different machine learning models.

	NSL-KDD → gureKDD		**gureKDD → NSL-KDD**		**Average**	
	Accu.	F1	Accu.	F1	Accu.	F1
Decision Tree	97.93%	98.25%	89.35%	88.10%	93.64%	93.18%
Random Forest (50)	99.27%	99.39%	93.74%	93.09%	**96.51%**	96.24%
Random Forest (100)	99.21%	99.33%	93.34%	92.60%	96.28%	95.97%
Random Forest (300)	**99.47%**	**99.55%**	92.36%	91.41%	95.91%	95.48%
AdaBoost (100)	98.60%	98.83%	94.07%	93.67%	96.34%	**96.25%**
GradientBoost (50 dev)	99.07%	99.22%	89.16%	87.55%	94.11%	93.39%
GradientBoost (50 expo)	99.28%	99.39%	89.36%	87.70%	94.32%	93.55%
LR (lbfgs)	95.92%	96.61%	79.47%	73.29%	87.69%	84.95%
Gaussian NB	42.46%	5.14%	79.28%	75.31%	60.87%	40.23%
Complement NB	31.62%	0.12%	79.15%	80.81%	55.38%	40.46%
Linear SVC	91.63%	93.20%	86.15%	83.95%	88.89%	88.57%
k-NN	95.73%	96.42%	89.73%	88.59%	92.73%	92.51%
QDA	45.26%	17.10%	64.09%	40.51%	54.67%	28.81%
LDA	92.05%	93.57%	85.22%	83.33%	88.64%	88.45%
ANN Model	98.31%	98.59%	91.99%	91.13%	95.15%	94.86%
CNN Model	95.12%	96.00%	**95.35%**	**95.11%**	95.23%	95.56%
LSTM Model	95.89%	96.62%	89.48%	87.98%	92.69%	92.30%
GRU Model	99.09%	99.23%	90.75%	89.80%	94.92%	94.52%
Best Result	99.47%	99.55%	95.35%	95.11%	96.51%	96.25%

Table 3.5: Results of testing of all models after the fix in binary classification (Train → Test).

3.5.5 Analysis

Fixing the service label shows how important this feature is in the learning process. The previously reported result [69] mostly did not reach over 10% of the F1 score with the cross-dataset evaluation method. Now we can gain more insight into the models and the datasets used.

When we use NSL-KDD for the training, most of the models obtain over 95% of F1 scores in the testing. Fifteen classifiers out of 18 get over 90%. The best model here is the Random Forest classifier with 300 trees, which gets 99.55% of the F1 score.

When we use gureKDD for the training, the results differ from each classifier. Only six models out of 18 get over 90% of testing F1 score. The best classifier in the scenario is the Convolutional neural network (CNN) with an F1 score of 95.35%.

The best average model from both scenarios was AdaBoost (96.25% of F1 score in testing). Most of the time-based decision classification methods (decision tree, Random Forest Classifier, AdaBoost Classifier, Gradient Boosting Classifier) perform better in both scenarios.

As for the datasets, the NSL-KDD is a better dataset to learn from compared with gureKDD. Interestingly, gureKDD contains 18 times the number of connections of NSL-KDD but a smaller number of distinct attacks.

For the feature importance, we notice that src_bytes shows before the fix and after the fix as one of the top 10 important features (—more details in the next part).

Feature Importance after the correction

We analyse feature importance by using Random Forest, as we did before. When NSL-KDD used for training, the top features of best and worst 10 trees as follows:

- **Best**: diff_srv_rate, src_bytes, dst_host_same_src_port_rate, dst_bytes, dst_host_rerror_rate, flag, count, service, protocol_type, and dst_host_srv_rerror_rate.

- **Worst**: dst_host_count, dst_bytes, same_srv_rate, srv_serror_rate, diff_srv_rate, src_bytes, dst_host_srv_count, dst_host_srv_rerror_rate, count, and protocol_type.

And when we use gureKDD as training set, the results as follows:

- **Best**: src_bytes, dst_bytes, dst_host_serror_rate, dst_host_same_srv_rate, count, duration, dst_host_srv_diff_host_rate, dst_host_srv_serror_rate, dst_host_count, and protocol_type.
- **Worst**: diff_srv_rate, count, dst_host_diff_srv_rate, flag, dst_host_count, duration, logged_in, dst_host_same_src_port_rate, src_bytes, and protocol_type.

There are three common features in the top 10 from all the above models. The features are src_bytes (number of data bytes from source to destination), count (number of connections to the same host as the current connection in the past two seconds) and protocol_type (type of the protocol, e.g. tcp, udp, etc.).

3.6 Summary

For a long time, we treated the problem of network intrusion detection systems (NIDS) without concern for the practicality of the result in the real world. The research is done so that we train and test our model in the same dataset and just calculate the accuracy. But, we argue that this way of thinking is impractical. So, we spent the first section where we use different datasets with different models. We showed that it is not difficult to reach a very high accuracy. But we need to go deeper and understand the effect of the change in the computer network. We propose the use of the cross-dataset evaluation method. We need to use two different datasets that are generated from two different computer networks. We use one dataset as a whole for training and the one for testing and vice versa. This way of evaluation is better to take out the effect of specific computer network architecture into the results. The result shows that all models perform badly, with less than 40% F1 score and most of them less than 10%. The rest of the chapter investigates reasons for such a drop in performance. It shows that the ways to preprocess the service feature in both datasets mismatch. The fixed result in a jump in the performance.

The next chapter will use the same cross-dataset evaluation method but for the multiclass classification problem.

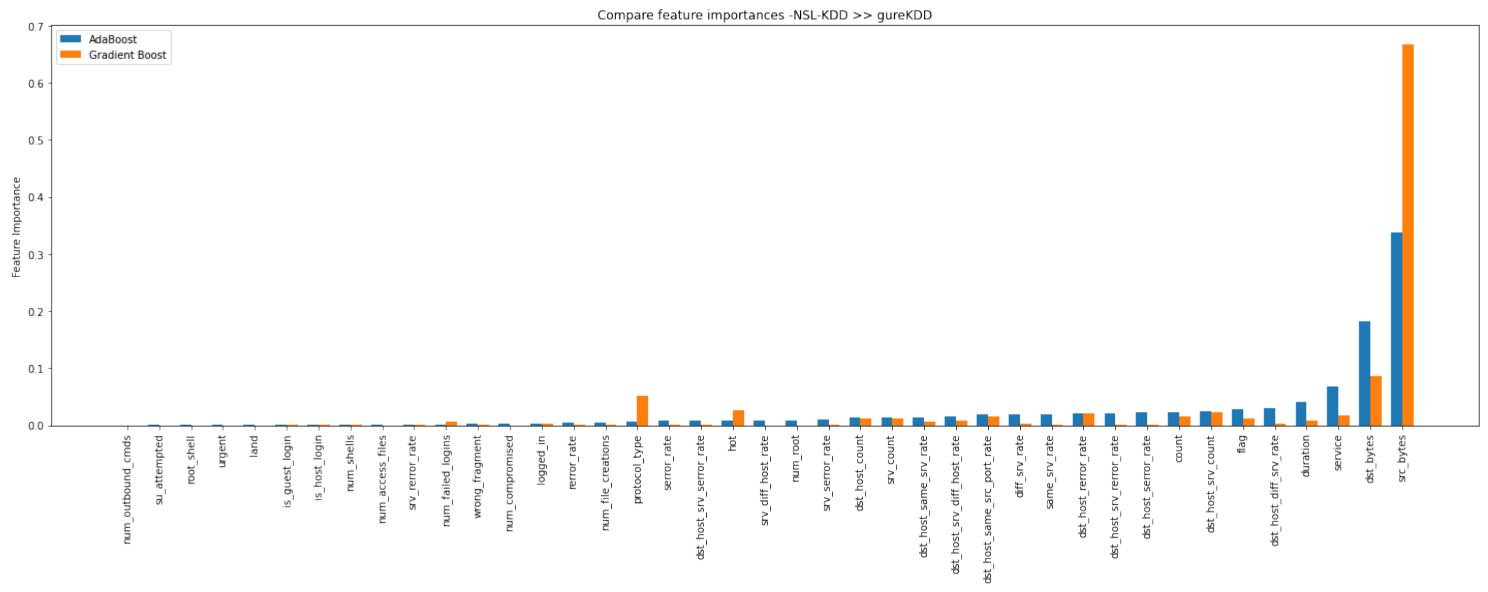


Figure 3.2: Feature Importance comparison between AdaBoost and GradientBoost. (NSL-KDD (training) → gureKDD (testing))

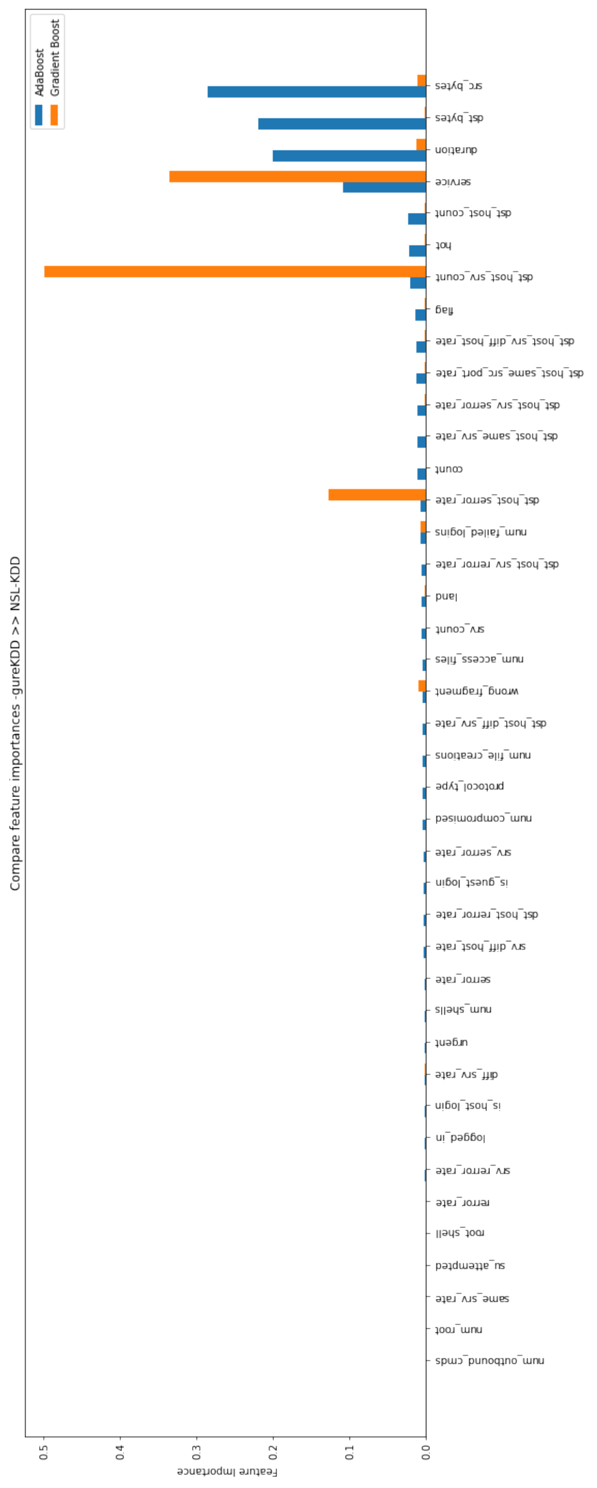


Figure 3.3: Feature Importance comparison between AdaBoost and GradientBoost. (gureKDD (training) → NSL-KDD (testing))

Chapter 4

Cross-datasets Evaluation in Multiclass Classification

4.1 Chapter Overview

The last chapter introduces the cross-datasets evaluation method and experiments with the binary classification problem. This chapter will use the same evaluation method but with a multiclass classification problem. The multiclass classification problem deals with a model rather than two target labels. We will use the same datasets as the previous chapter but require further preprocessing to make them compatible. Finally, the chapter will end with an example application for such a method: signature generation.

4.2 Multiclass Classification

4.2.1 Problem Overview

In this section, we investigate cross-datasets evaluation for multiclass classification in the intrusion detection task. The multiclass classification problem is when we have more than two types of labels. For example, in a network intrusion detection system (NIDS), the binary classification means intrusion or not intrusion as a label. On the other hand, the multiclass classification in NIDS might have further details about the attacks, like the group of the intrusion or the specific name of the attack.

The idea of the cross-datasets evaluation method is to use two different datasets generated in two different computer network settings. We use the first dataset in the training of the machine learning models and use the other in testing. Then, we reverse the process by swapping the training and testing datasets. The method allows one to evaluate the quality of the detection models and the quality of the datasets used. The method used in the last chapter with the binary classification problem. At the beginning of the experiment, the results were very poor. However, a proposed fix was implemented that increased the performance. The fix is on the gureKDD [61] dataset that makes it match the NSL-KDD dataset [84].

4.2.2 Datasets

There are several labelled datasets used to evaluate network intrusion detection systems. For example, NSL-KDD is a refined version of the popular KDD Cup 1999 datasets. The dataset can be used in binary classification (normal, attack) or multiclass classification (5 attack categories or the name of the exact attack) [84].

We will use the same datasets as in the last chapter: NSL-KDD [84] and gureKDD [61]. Both datasets were generated from two different networks. NSL-KDD is a refined version from a famous KDD cup 1999 dataset, which reported having some problems [51][48]. The NSL-KDD dataset comprises 148,517 connections with 40 distinct attacks. The gureKDD dataset created from scratch followed the specifications of the KDD cup 1999 dataset. The gureKDD datasets contains 2,759,494 connections with 28 distinct attacks. NSL-KDD dataset contains more distinct attacks but fewer connections.

Dataset Labels

The labels on NSL-KDD and gureKDD are for a different type of attacks. The problem is that those labels do not match. There are several attacks in both datasets. The following are the labels available for each dataset:

- **NSL-KDD**: there are 28 labels. The following is the labels: ['normal', 'warezclient', 'dict', 'warezmaster', 'teardrop', 'syslog', 'land', 'guest', 'imap', 'phf', 'multihop', 'rootkit', 'ftp-write', 'loadmodule', 'eject', 'format', 'format_clear', 'ffb', 'ffb_clear', 'anomaly', 'perlmagic', 'format-fail', 'perl_clear', 'spy', 'eject-fail', 'warez', 'dict_simple', 'load_clear']

- **gureKDD**: there are 40 labels as follows: ['normal', 'neptune', 'warezclient', 'ipsweep', 'portsweep', 'teardrop', 'nmap', 'satan', 'smurf', 'pod', 'back', 'guess_passwd', 'ftp_write', 'multihop', 'rootkit', 'buffer_overflow', 'imap', 'warezmaster', 'phf', 'land', 'loadmodule', 'spy', 'perl', 'saint', 'mscan', 'apache2', 'snmpgetattack', 'processtable', 'httptunnel', 'ps', 'snmpguess', 'mailbomb', 'named', 'sendmail', 'xterm', 'worm', 'xlock', 'xsnoop', 'sqlattack', 'udpstorm']

To run the cross-datasets evaluation method for multiclass, we need to match both datasets labels. We can remove data that have no matching label in the other datasets. But we prefer not to lose any data to compare the results with those from binary classification. Another option we have is to map the attacks label into the category of the attacks. Since the original KDD datasets give a specification for grouping the attacks, we decided to follow the same grouping/categories.

4.2.3 Mapping into five categories

Based on the original KDD datasets [83], there are five different labels (Normal and four groups of attack) as following:

- **Normal**: Normal traffic.
- **DoS**: Denial of Service attacks, e.g. syn flood
- **R2L** R2L: unauthorised access from a remote machine, e.g. guessing password;
- **U2R** U2R: unauthorised access to local superuser (root) privileges, e.g., various "buffer overflow" attacks;
- **Probing** probing: surveillance and other probing, e.g., port scanning.

The grouping process was not straightforward since mismatch or lack of the documentation. It required an understanding of each attack before it categorised it. The final list of groups is as following:

- **DoS**: 'back', 'land', 'neptune', 'pod', 'smurf', 'teardrop', 'mailbomb', 'apache2', 'processtable', 'udpstorm', 'syslog'.
- **R2L**: 'ftp_write', 'ftp-write', 'guess_passwd', 'imap', 'multihop', 'phf', 'spy', 'warezclient', 'warezmaster', 'warez', 'sendmail', 'named', 'snmpgetattack', 'snmpguess', 'xlock', 'xsnoop', 'worm', 'dict', 'dict_simple', 'guest'.

- **U2R**: 'buffer_overflow', 'loadmodule', 'perl', 'perl_clear', 'perlmagic', 'rootkit', 'http-tunnel', 'ps', 'sqlattack', 'xterm', 'eject', 'eject-fail', 'ffb', 'ffb_clear', 'format', 'format-fail', 'format_clear', 'load_clear'.
- **Probing**: 'ipsweep', 'nmap', 'portsweep', 'satan', 'mscan', 'saint'.

4.2.4 Machine Learning Algorithms

In this work, we tried to include a wider range of machine learning models. This allows us to compare the performance of each model. The machine learning models that we used are Decision tree learning, Random forest [35] (with a different number of trees), AdaBoost [29], Gradient boosting [30], Logistic Regression (LR), Naive Bayes (NB), Linear Support Vector Machine (SVM) [88], K-Nearest Neighbours Algorithm (k-NN), Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA), Artificial neural networks (ANN), Convolutional neural network (CNN), Long short-term memory (RNN LSTM) [36] and Gated Recurrent Units (RNN GRU) [13].

4.2.5 Results and Discussion

Tables 4.1 and 4.2 summarise the results of testing the F1 score for NSL-KDD, respectively, with gureKDD used as training sets. The results are shown for each class of the dataset. There are also micro, macro, and weighted averages [explanation in Chapter 2].

Learning normality of the traffic gets the highest results on both flows of datasets, followed by the DoS attacks. The rest of the groups (R2L, U2R, and Probe) was difficult for all models to detect. The worst label to detect is Probe, even though the Probe label is the third largest sample in both datasets (after normal and DoS labels).

Tree-based models perform well in this task most of the time. If we consider macro average and weighted average as a metric, then tree-based models will win here. The k-NN model was not at the top of the list for the binary classification, but this task performs better.

Learning from NSL-KDD gives better results than gureKDD in all metrics of best results. This might be because the NSL-KDD contains a more distinct number of attacks than gureKDD, even though that gureKDD is a larger dataset (2,759,485 data points compared with 148,517 in NSL-KDD).

						Averages		
ML Models	Normal	DoS	R2L	U2R	Probe	Micro	Macro	Weighted
DT	92.68%	88.30%	9.73%	0.55%	21.40%	85.07%	42.53%	88.58%
RF (50)	99.36%	97.79%	0.37%	4.84%	61.59%	97.18%	52.79%	97.56%
RF (100)	98.14%	94.74%	0.00%	5.08%	46.64%	93.99%	48.92%	95.01%
RF (300)	99.41%	94.79%	0.00%	5.41%	41.61%	94.03%	48.24%	95.45%
AdaBoost(100)	89.13%	56.27%	14.55%	9.38%	9.27%	59.96%	35.72%	68.68%
GradientBoost	99.23%	92.79%	45.95%	3.39%	34.82%	91.90%	55.24%	94.13%
Gaussian NB	6.90%	74.88%	0.94%	0.02%	19.37%	58.43%	20.42%	45.78%
Comp. NB	72.21%	96.78%	4.99%	0.00%	0.16%	80.25%	34.83%	84.56%
Linear SVC	89.35%	70.16%	3.69%	0.00%	10.61%	66.19%	34.76%	76.67%
k-NN	94.73%	98.30%	7.49%	34.41%	74.71%	95.27%	61.93%	96.24%
QDA	57.98%	9.06%	1.36%	0.11%	18.25%	42.23%	17.35%	29.28%
LDA	87.18%	4.54%	32.71%	2.84%	6.25%	35.47%	26.70%	38.44%
ANN	92.68%	88.30%	9.73%	0.55%	21.40%	85.07%	42.53%	88.58%
CNN	89.90%	13.30%	15.47%	2.30%	4.72%	54.03%	25.14%	44.48%
LSTM	75.31%	21.67%	16.38%	6.24%	8.64%	49.15%	25.65%	43.35%
GRU	98.85%	88.41%	23.61%	7.11%	15.98%	87.30%	46.79%	91.07%
Best Result	99.41%	98.30%	45.95%	34.41%	74.71%	97.18%	61.93%	97.56%

Table 4.1: Results of testing F1 score - NSL-KDD as training set

						Averages		
ML Models	Normal	DoS	R2L	U2R	Probe	Micro	Macro	Weighted
DT	91.99%	92.10%	42.66%	17.89%	75.52%	89.55%	64.03%	89.10%
RF (50)	92.42%	91.83%	0.64%	0.00%	77.39%	89.80%	52.46%	88.31%
RF (100)	92.20%	91.51%	0.43%	0.00%	72.90%	89.26%	51.41%	87.65%
RF (300)	91.69%	91.64%	0.27%	0.00%	67.63%	88.70%	50.25%	86.93%
AdaBoost(100)	81.94%	87.30%	20.23%	0.00%	41.40%	77.85%	46.18%	78.33%
GradientBoost	91.78%	88.21%	23.08%	1.27%	48.94%	86.35%	50.66%	84.55%
Gaussian NB	73.94%	10.34%	6.08%	7.13%	21.66%	41.75%	23.83%	44.30%
Linear SVC	90.19%	90.92%	5.94%	0.87%	23.67%	84.90%	42.32%	81.87%
k-NN	90.67%	87.08%	27.19%	16.00%	64.93%	86.01%	57.17%	85.21%
QDA	81.46%	1.75%	22.16%	0.98%	20.50%	52.35%	25.37%	45.40%
LDA	89.56%	87.90%	20.12%	1.14%	6.44%	82.42%	41.03%	79.18%
ANN	92.56%	82.20%	0.00%	0.00%	0.00%	82.82%	34.95%	77.57%
CNN	92.28%	92.21%	4.18%	0.00%	19.78%	86.70%	41.69%	83.00%

LSTM	91.51%	88.74%	14.20%	0.00%	57.15%	86.39%	50.32%	85.15%
GRU	90.79%	88.18%	14.99%	0.00%	46.45%	85.62%	48.08%	83.58%
Best Result	92.56%	92.21%	42.66%	17.89%	77.39%	89.80%	64.03%	89.10%

Table 4.2: Results of testing F1 score - gureKDD as training set

Using two different datasets generated from two different computer networks should give us a clearer picture of how much we should trust the model. And since we want to understand the behaviour of the attack, different attacks in both datasets will pose a more challenging task.

The result of both classification problems (binary and multiclass) tend to give a higher result for detecting the normality. This might give an insight for a better intrusion detection system based on normality detection (as a variant of anomaly-based intrusion detection based on machine learning).

Our result indicates that when we use two different datasets, the dataset that contains more distinct attacks will perform better than one with more connections. The observation is that the NSL-KDD provides more attacks but fewer connections than gureKDD. Using NSL-KDD as a training set gives a better result on the testing set across most models. This reflects the quality of the learning set. We speculate that more attacks in the dataset will provide better learning for the model, but this observation needs further investigation.

Based on our work here, deep learning models do not give an advantage when compared with classical machine learning models like tree-based ones. And by using tree-based models, we have the advantage of the ease of model interpretation.

The highest F1 score reached is 99.41% (NSL-KDD $\rightarrow$ gureKDD) with the normality label (92.56% in gureKDD$\rightarrow$ NSL-KDD). The second label is DoS attack with 98.3% (NSL-KDD $\rightarrow$ gureKDD) and 92.21% in gureKDD$\rightarrow$ NSL-KDD. All models struggle with the other three groups of the attacks.

4.3 Application: Automation of Signature Generation

A signature-based intrusion detection system has an advantage of speed compared with the anomaly-based IDS. One challenge in signature-based IDS is the generation of the signature. The speed of the changes in the malware and attacks required more automation on the side of signature generation. It is also important for the signature to be correct.

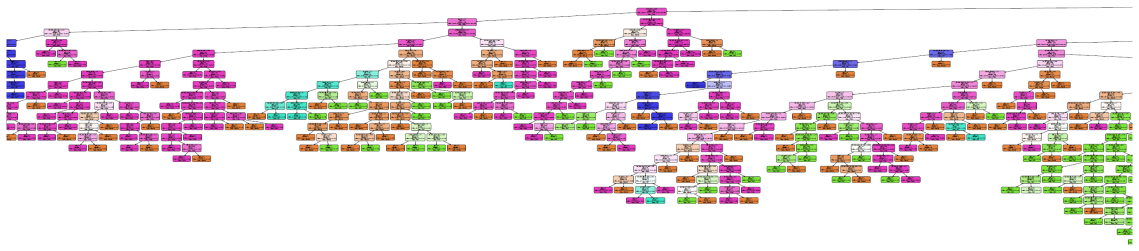


Figure 4.1: A demonstration of how a decision tree looks like after NSL-KDD training
.

Cross-dataset evaluation might help in the automation pipeline of signature generation. For example, we can use the labels that achieve high results (Normality and DoS attacks) and generate the signature out of the decision tree (DT) or random forest (RF) models.

The pipeline of such a system might look like the following:

1. Use two computer networks to generate the labelled data. Both networks should be with a different architecture.

2. Fit a decision tree model with the data in a different network and test them with the other data from the other network.

3. Extract the rules from the tree model as signatures and Verify the signatures by experts.

Signature-based IDS works based on the intrusion signatures. We can also take the blacklist approach, allowing only the normal behaviour of the network and blocking the other activities. We can generate signatures for the normality instead of the intrusions.

Figure 4.1 shows a demonstration of how the decision graph looks like from a decision tree when NSL-KDD is used in training. We can get 314 rules from the tree to use as a signature after the verification. The number of rules out of each label are as follows: Normal (163 rules), DoS (48 rules), R2L (33 rules), U2R (5 rules), and Probe (65 rules). Code 4.1shows examples of a normal signature generated from the NSL-KDD dataset extracted from the decision tree model. The full list of the rules can be found in the thesis repository (Check section 1.8).

Code 4.1: Example of Rules Extracted from the Decision Tree

```
1- WHEN dst_host_serror_rate <= 0.9750000238418579 && dst_host_srv_count <=
    164.5 && dst_bytes <= 609.0 && dst_bytes <= 90.5 && service_ftp_data <=
    0.5 && dst_host_srv_diff_host_rate <= 0.014999999664723873 && rerror_rate
     <= 0.9950000047683716 && dst_host_same_srv_rate > 0.014999999664723873
    && protocol_type_tcp <= 0.5 && service_urp_i <= 0.5 && service_ecr_i <=
    0.5 && dst_host_count > 2.5 && dst_host_count <= 253.0 &&
    srv_diff_host_rate <= 0.004999999888241291 && service_domain_u <= 0.5 &&
    dst_host_same_srv_rate <= 0.7899999916553497 && srv_count > 3.5 &&
    dst_host_diff_srv_rate > 0.044999999925494194 &&
    dst_host_same_src_port_rate > 0.16499999910593033 && service_other > 0.5
    THEN Y=Normal (probability=1.0) (values=[[1.5419524 0.,0.,0.,0.,]])
2- WHEN dst_host_serror_rate > 0.9750000238418579 &&
    dst_host_srv_diff_host_rate <= 0.014999999664723873 && flag_SH <= 0.5 &&
    logged_in > 0.5 && srv_count <= 1.5 && dst_bytes > 500.5 && dst_bytes >
    1550.5 THEN Y=Normal (probability=1.0) (values=[[2.3129286 0,0.,0.,0.,]])
3- WHEN dst_host_serror_rate <= 0.9750000238418579 && dst_host_srv_count <=
    164.5 && dst_bytes <= 609.0 && dst_bytes <= 90.5 && service_ftp_data <=
    0.5 && dst_host_srv_diff_host_rate <= 0.014999999664723873 && rerror_rate
     <= 0.9950000047683716 && dst_host_same_srv_rate > 0.014999999664723873
    && protocol_type_tcp > 0.5 && service_other <= 0.5 && service_imap4 <=
    0.5 && dst_host_srv_count <= 2.5 && service_private <= 0.5 &&
    dst_host_diff_srv_rate <= 0.5349999964237213 && src_bytes <= 4.5 && land
    <= 0.5 && same_srv_rate > 0.5 && service_telnet > 0.5 THEN Y=Normal (
    probability=1.0) (values=[[3.46939289 0.,0.,0.,0. ]])
```

4.4 Summary

The applicability of the machine learning-based intrusion detection systems should be given more attention in the research. Using the cross-datasets evaluation method is one approach to measure the practicality of detection classifiers. This chapter is an extension of the last chapter. The work here shows the use of the cross-datasets evaluation method in the multiclass classification. The experiment results show the models can achieve good results in the normal and DoS attack labels. It also shows that using the NSL-KDD dataset is better for learning than the gureKDD dataset, even though the gureKDD is a bigger dataset. The chapter ended with the application of the cross-datasets evaluation method in the automation of signature generation. The next chapter will explore the splitting evaluation method over time for malware detection.

Chapter 5

Malware Detection Performance Over The Time - Time-Based Splits

5.1 Chapter Overview

In this chapter, we will propose a new way of evaluating the performance of machine learning models for the malware detection task. The goal of the evaluation is to have a better way to evaluate the practicality of the model in real-world scenarios. We will divide the malware dataset into several splits. All splits are equal in the time duration, regardless of the size. This evaluation method will allow us to understand the model in the future and past malware, leading to designing a better malware detection model. We will test two different machine learning models and analyse the results. The analysis will include the behaviour of different malware groups.

5.2 Introduction

Over time, the changes in malware dynamics make it a challenging task for the machine learning model to keep up to date. The process will require obtaining new malware data with high-quality labels and retraining the detection model. However, we need to understand the detection model performance over time. The understanding will include training the model in a certain time frame and testing the model on future and past data.

The conventional way of evaluating machine learning in malware detection is not to check the model adaptability over time. As we saw in previous chapters, the normal procedure is to divide the dataset into training and testing splits. Each split has a role in the process. The training split is used to train the machine learning model to detect the malware. The testing part is used to test the performance of the model on unseen data. There is a small portion of the training call validation set used to hyperparameter tune the model during the development of the model to avoid overfitting. Check Figure 2.1 from the earlier chapter for illustration.

In this work, we suggest a different way of splitting the dataset that considers the time dimension of the malware discovery in the security labs.

This chapter aims to test how the detection model created now with a specific time frame will perform for unseen malware from the past and future. Such a question will make us understand how often we need to train the model.

We want to check the following:

- How much does the accuracy get affected over time without retraining the model?
- Can a model that learned from a certain time frame data perform well on future and past malware data? And in which direction does is perform better? And in which direction it is performing better?
- How much does the accuracy get affected in different malware groups over time?
- Which group of malware needs more retraining and which one less?

Answering such questions will give a better understanding of how much and how often we need to retrain our model. It will also allow us to develop a better machine learning model to adapt to future and past malware. This chapter aims to test how the detection model created now with a specific time frame will perform for unseen malware from the past and future. Such a question will make us understand how often we need to train the model.

5.3 Methodology

In this section, we will discuss the following questions: what dataset to use? How will the dataset be divided? What are different flows/strategies to train and test the splits? What are the machine learning models that we will use? Furthermore, what are the performance metrics to measure the performance?

5.3.1 SOREL-20M Dataset

SOREL-20M [33] is a recent dataset with many malware samples of 20 million files with pre-extracted features and metadata. The goal of the dataset is to become a benchmark dataset for malware research like other famous datasets in image classification or natural language processing (CIFAR [43], ImageNet [22], or Stanford Sentiment Treebank [78]). Sophos and ReversingLabs provide the dataset. The dataset contains 9,762,177 malware and 9,962,820 benign samples. There is also a label for each malware group. There are 11 groups: Adware, Flooder, Ransomware, Dropper, Spyware, Packed, Crypto Miner, File Infector, Installer, Worm, and Downloader.

The authors of the dataset suggest the splitting of the dataset into three different parts: Training (7,596,407 samples), Validation (962,222 samples), and Testing (1,360,622 samples). They provided the timestamp of each split. The separation between those splits are not random but based on time. The beginning of the dataset is used for the training, followed by the validation and testing parts. The goal of the split is to benchmark different models by the researchers. For that, they also provided two base models that have been train in the dataset: LightGBM [39] and Base model based on Feed-Forward Neural Network ((FFNN) model (smaller version of Aloha model [72])).

5.3.2 Multiclass Classification - Malware Groups

The type of problem that we are dealing with is a multiclass classification one. There are ten different types of malware and benign labels. The description of how the classes are derived can be found in [72]. The malware classes are as follows:

1. **Adware:** A type of malware that hides in the computer to show an advertisement.
2. **Flooder:** It will convert many computers to a botnet that the attacker uses to send a massive amount of data to a certain target.
3. **Ransomware:** A type of malware that will encrypt the data in the computer and then ask for a ransom to recover the data or the system.

4. **Dropper:** A type of malware that is designed to avoid detection by downloading another type of malware once it is activated.

5. **Packed:** A type of malware designed to avoid detection and reverse engineering by using compression.

6. **Crypto Miner:** Also known as cryptojacking, a type of malware that uses the computer's resources to mine cryptocurrencies.

7. **File Infector:** A type of malware that attached itself with computer programs, like games or word processors.

8. **Installer:** A type of malware that acts as a download manager to download and install another malware during the installation of a software.

9. **Worm:** A type of malware that replicates and spreads itself to other computers.

10. **Downloader:** A type of malware that infects the computer and waits for an internet connection to download another malware.

Figure 5.1 shows the occurrence of each malware group in the dataset over time and Figure 5.2 shows the heatmap of each group monthly.

5.3.3 Splitting the dataset

Our goal is not to provide a better deep learning model that gives better results but to investigate the model performance over the timeline. For that, we will not use the recommended split by the dataset authors. We will split the dataset into almost equal time intervals. The window time duration is around three months (except the last split with two more weeks). In total, we will have nine splits. Any sample that contains a null value will be removed from training and testing. The number of samples of each split vary significantly, but we will argue that we will leave it as it stands. We argue here that we want to preserve the natural occurrence of malware in the lab. Sometimes there will be a massive spike of malware; other times, less. For example, split 1 represent only 1.37% of the training size, while split 8 represent 25.87%. The complete timestamp of the splits with the proportion can be found in Table 5.1. There is also a validation part in each split. The validation will be used during the training to optimise the model. The time of validation

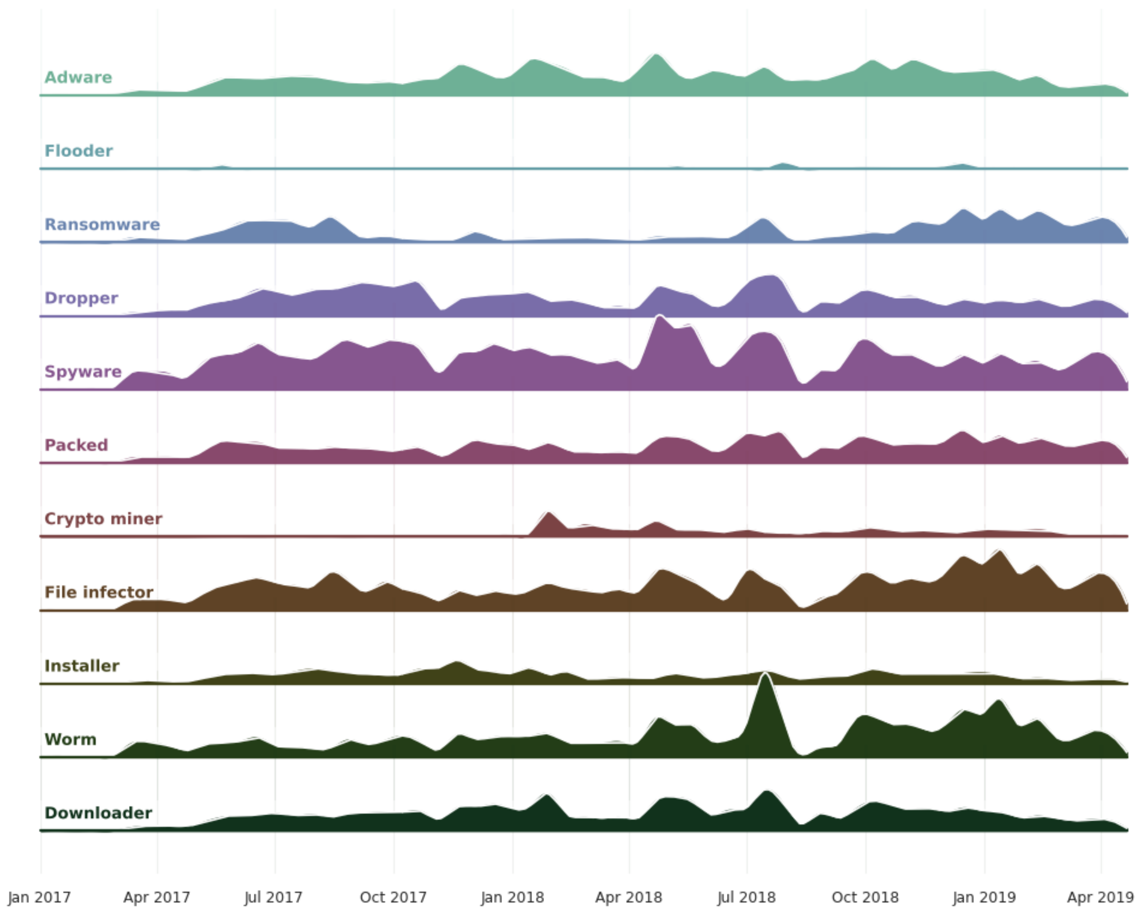


Figure 5.1: Timeline of each Malware Group in the dataset

started from the time of the validation to the end of the split. The size of the validation split is the last seven days of each split.

5.3.4 Distribution of malware groups in each split

Table 6.1 shows the distribution of each malware group in each split of the time-based split. Figure 5.3 shows the heat map of the distribution in each split. We can notice that flooder and crypto-miner contain the least amount of samples compared with the rest of the malware groups. The distribution of labels will be helpful later as part of analysing each group.

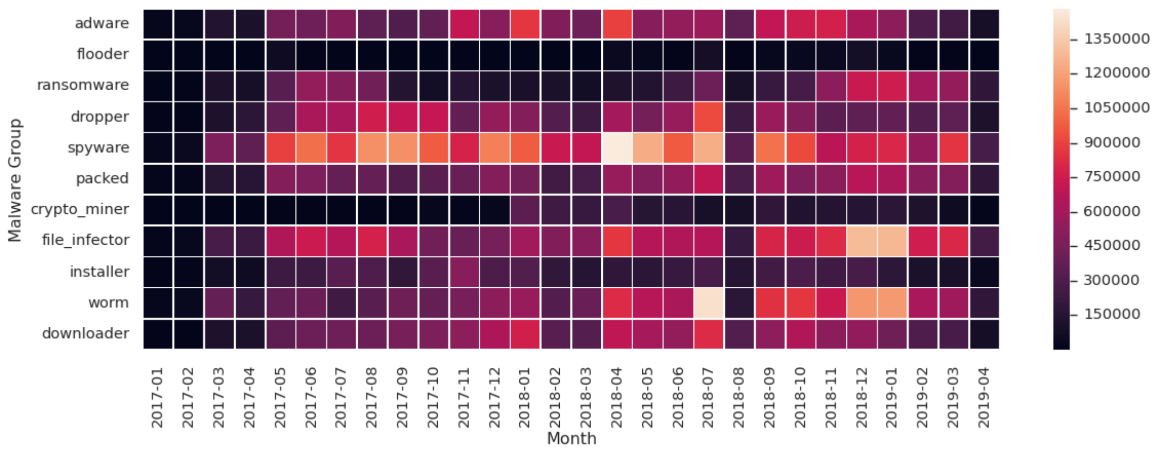


Figure 5.2: Heatmap of malware group by month

Table 5.1: Dataset Time-Based Splits with size.

	Start	Timestamp	Validation	Timestamp	Training Size	Validation Size	Train %
Split1	01/01/2017	1483228800	24/03/2017	1490313600	246,676.00	37,285.00	1.37%
Split2	01/04/2017	1491001200	23/06/2017	1498172400	774,597.00	89,490.00	4.29%
Split3	01/07/2017	1498863600	23/09/2017	1506121200	2,160,131.00	92,956.00	11.96%
Split4	01/10/2017	1506812400	24/12/2017	1514073600	957,925.00	97,869.00	5.30%
Split5	01/01/2018	1514764800	24/03/2018	1521849600	933,257.00	88,468.00	5.17%
Split6	01/04/2018	1522537200	23/06/2018	1529708400	1,525,367.00	471,072.00	8.45%
Split7	01/07/2018	1530399600	23/09/2018	1537657200	2,672,843.00	497,246.00	14.80%
Split8	01/10/2018	1538348400	24/12/2018	1545609600	4,671,596.00	373,302.00	25.87%
Split9	01/01/2019	1546300800	27/03/2019	1553644800	4,115,300.00	721,569.00	22.79%

Table 5.2: Malware Groups Distribution of each split in Time-Based.

Split	adware	flooder	ransomware	dropper	spyware	packed	crypto_miner	file_infector	installer	worm	downloader
1	184512	2811	125985	127734	534556	185850	2023	315799	96652	441494	130271
2	972537	64251	987679	1169641	2298606	1127514	18310	1608180	533912	994814	871662
3	1162758	11886	1064320	2093876	3131371	1079549	29083	2052370	843583	1008045	1277188
4	1593560	17496	328172	1650683	2850762	1243438	65837	1272670	1151790	1358975	1634972
5	1769849	10749	282093	1049918	2423160	973675	825128	1581283	656552	1289879	1435612
6	1947002	73013	492888	1611290	3697490	1579919	602462	2157260	571013	2109077	1858007
7	1645675	115601	721998	1729057	2624781	1578611	381660	1659173	686650	2445162	1670281
8	2148282	157632	1523961	1206377	2378347	1672346	423776	2859102	821311	2762371	1707007
9	1164245	53212	2081749	1163206	2466075	1812219	363694	3102783	429094	2564519	1081829

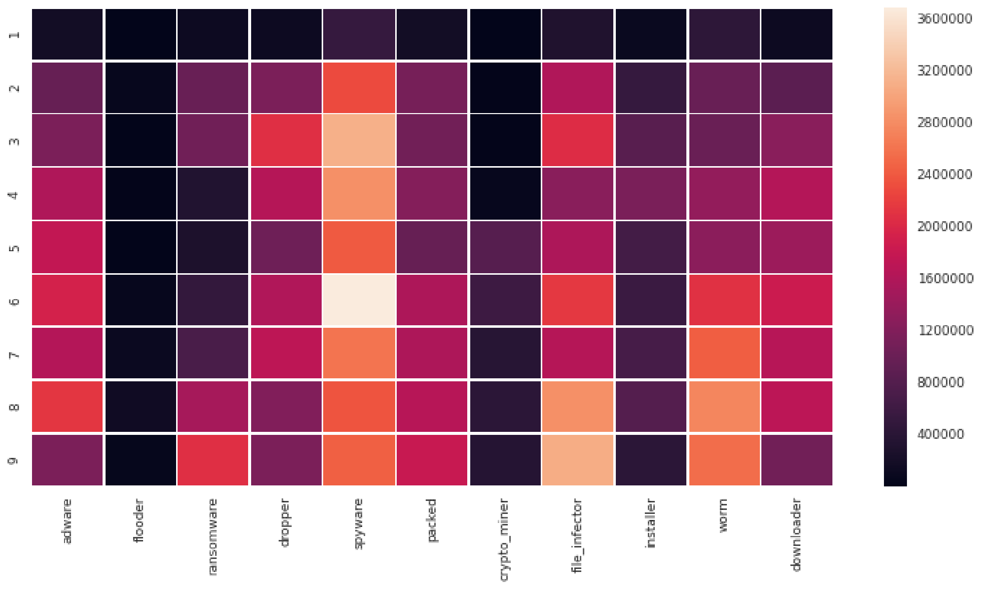


Figure 5.3: Size Based - Malware Groups Distribution

5.3.5 ML Model

The model that we will use in this experiment is based on the Aloha model (Auxiliary Loss Optimisation for Hypothesis Augmentation) [72]. The main contribution of the model is using multiple sources for the loss sources, also known as *'auxiliary loss'*, including multi-source malicious/benign, a count of multi-source detection, and malware tags. Figure 5.4 illustrates the Aloha model architecture. The authors of the paper show that using this way improves the performance of the machine learning model.

Since there are multiple output layers of the models, we will report the results of the malware groups/tags. The results of the group will give more insights into the malware behaviours of different groups.

We will use two models. Both models are based on Artificial Neural Networks (ANN). Moreover, both models are created by the same authors that provided the dataset. The base model is provided with the dataset repository, while the Aloha model is described in the original paper [72].

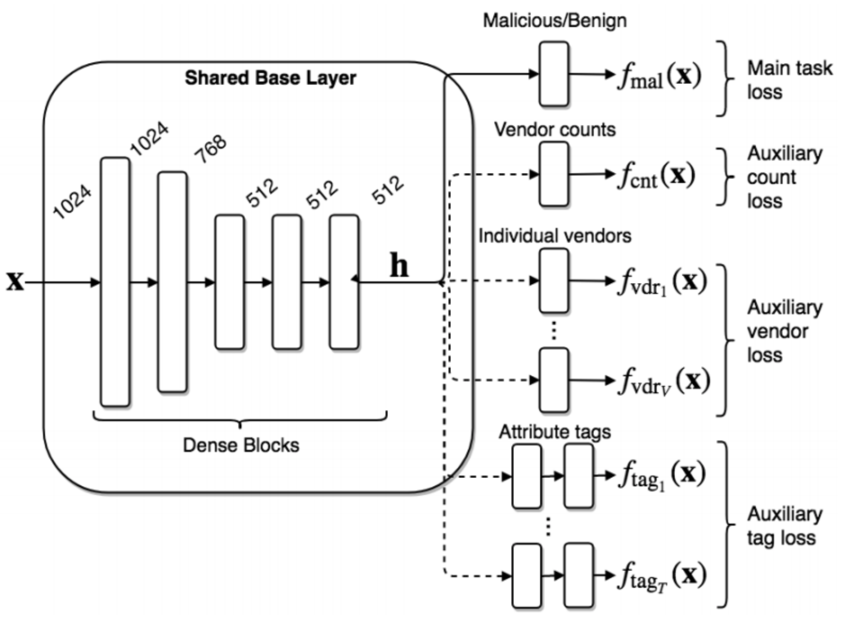


Figure 5.4: Aloha Model Architecture

Base Model

The author of the SOREL dataset provides this model as a benchmark for the dataset. It is just a stack of three artificial neural layers (size in order: 512,512,128) with dropout regularisation of 15% [82] and an exponential linear unit (ELU) activation function [14]. The model's code is available in the GitHub repository of the SOREL dataset [1]. Similar to the Aloha model, the base model also uses the same auxiliary Loss and output layer.

Aloha Model

The full details of the Aloha model (Auxiliary Loss Optimisation for Hypothesis Augmentation) can be found in the paper [72] and illustrated in Figure 5.4. The model contains five hidden layers of sizes (1024, 768, 512, 512, 512). Each layer contains exponential linear unit (ELU) activation [14] and dropout of 15% [82]. The model uses Adam optimiser as a learning method [41].

[1] `https://github.com/sophos-ai/SOREL-20M"`

5.3.6 The Flow of the Training and Testing

To train and test with the splits, we have five different flows as follows:

1- Normal Flow

In the normal flow, each split will be used for training individually and testing with each other. For example, in the first step, we will train the model with split one and then report the performance with all other splits. In the next phase, we will train with split two and with the rest and so on. Figure 5.5 illustrates the normal flow. The validation part during the training is the one contained in each split.

The normal flow might look like the normal k-fold cross-validation method but there are differences as follows:

1. Normal flow uses the small split as training, while k-fold CV uses the small fold/split as a validation set.

2. Normal flow test on the other splits individually. In k-fold CV, they combine the rest of the splits together to be used for the training.

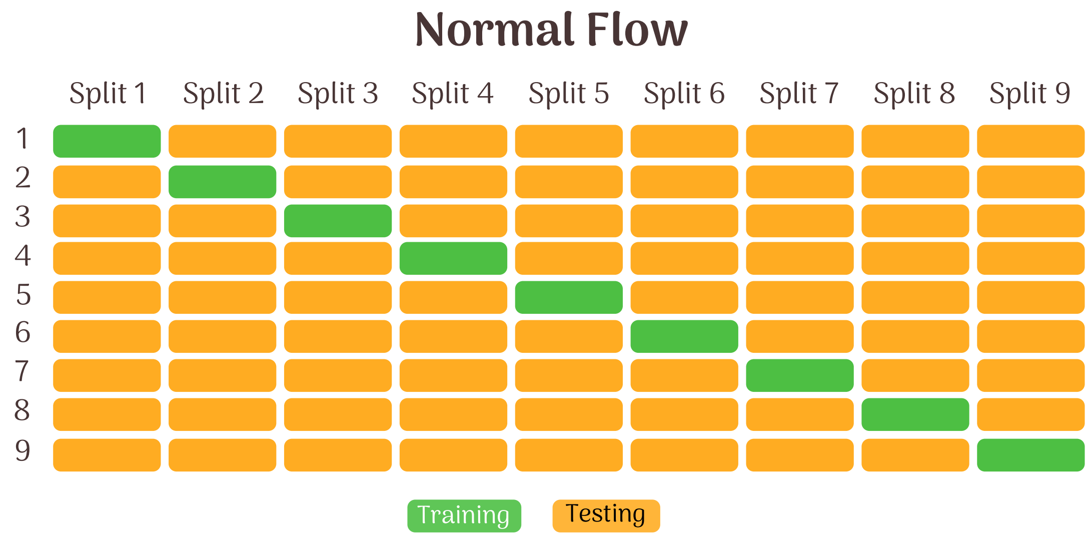


Figure 5.5: Normal Flow Illustration

2- Forward Accumulative with the Whole Testing

In the accumulative flow, the size of data in the training increases by combining more splits at each stage. The testing will include all the remaining data that have not been used in training. For example, in stage 1, we will use split 1 as training and the rest of the splits as testing at once. For the next stage, we will use split 1 and 2 for the training and combine the rest of the splits for the testing. In this way, we will keep increasing the size of training. Figure 5.6 illustrates the forward accumulative with the whole testing flow. The validation part that will be used in this flow is the validation of the end split. For example, in stage 6, the validation in split 6 will be used.

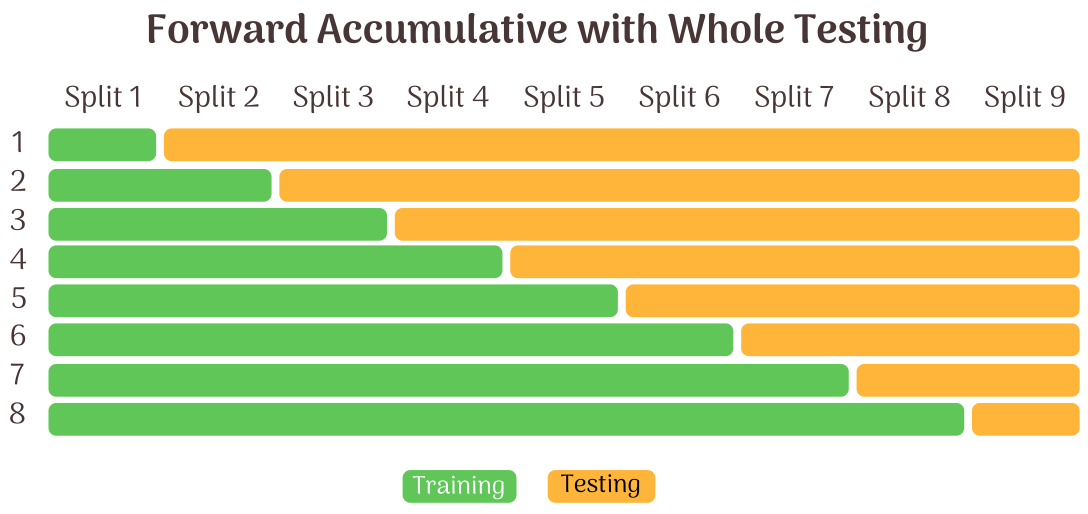


Figure 5.6: Forward Accumulative with Whole Testing Flow Illustration

3- Forward Accumulative with Splits Testing

This flow is similar to forward accumulative with the whole testing flow. The only difference is in the testing. Instead of combining all the reaming splits in the testing, we will test each split separately. Figure 5.7illustrates the forward accumulative with the whole testing flow. For the validation part, it will be similar to the Forward Accumulative with the Whole Testing. The last split validation will be used.

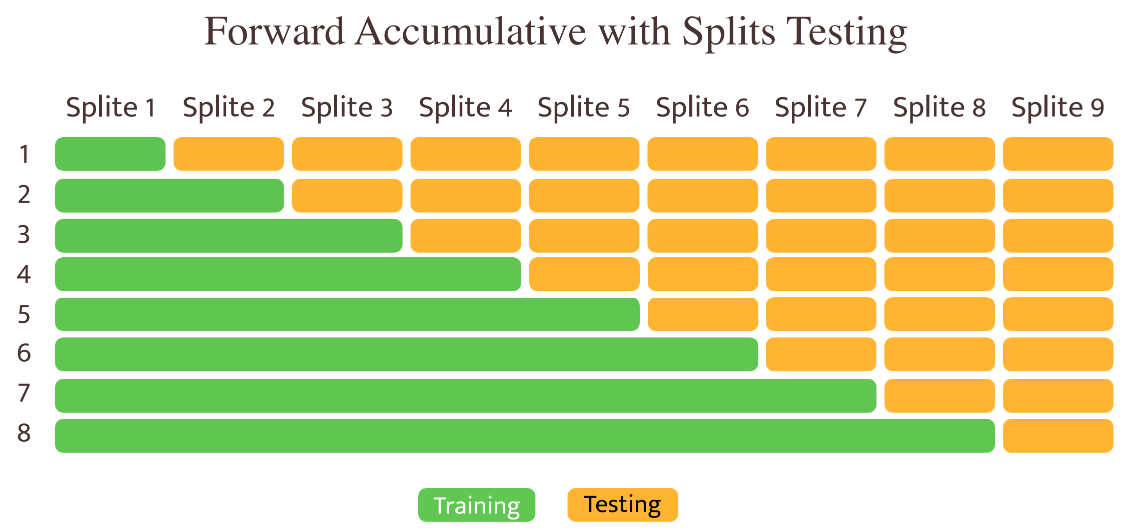


Figure 5.7: Forward Accumulative with Splits Testing Flow Illustration

4- Backward Accumulative with the Whole Testing

This process is similar to forward accumulative with the whole testing but to move backwards. We will not start from split 1; we will start from the last split moving backwards. In the first stage, we will use split 9 for the training and the rest of the splits in the testing combined. In stage 2, we will use split 9 and 8 for the training and test with the other splits. We keep increasing the training size while decreasing the testing size. Figure 5.8 illustrates the backward accumulative with the whole testing flow. In this flow, the validation of split nine will be used in all stages.

5- Backward Accumulative with Splits Testing

This flow is similar to backward accumulative with the whole testing. The difference is in the testing part. Instead of testing with all data, we test in each split separately. For example, in the first stage, split 9 will be used in training and tested in each split. Figure 5.9illustrates the backward accumulative with the whole testing flow. For the validation split, it will be similar to Backward Accumulative with the Whole Testing. We will use the validation of split 9 for all stages.

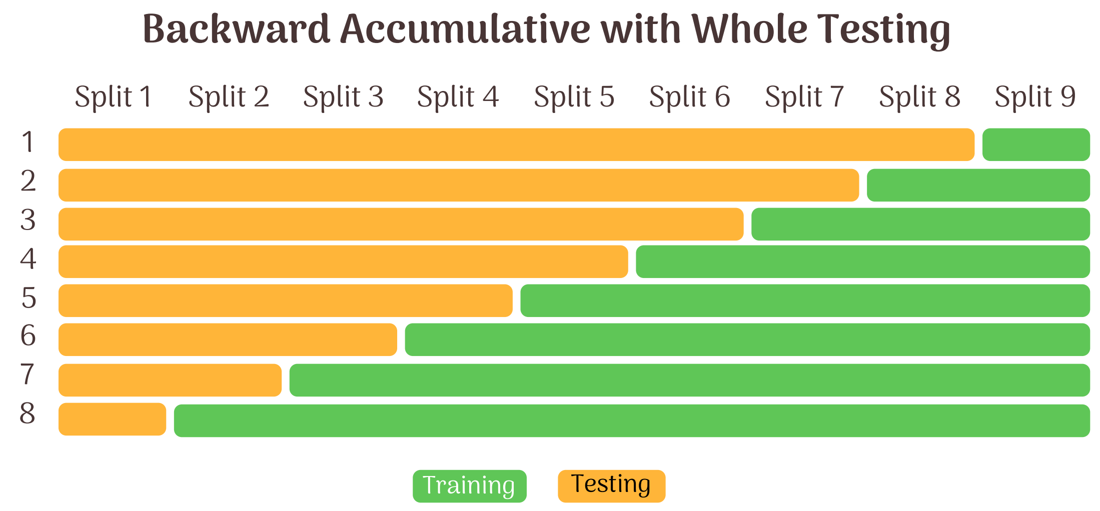


Figure 5.8: Backward Accumulative with Whole Testing Flow Illustration

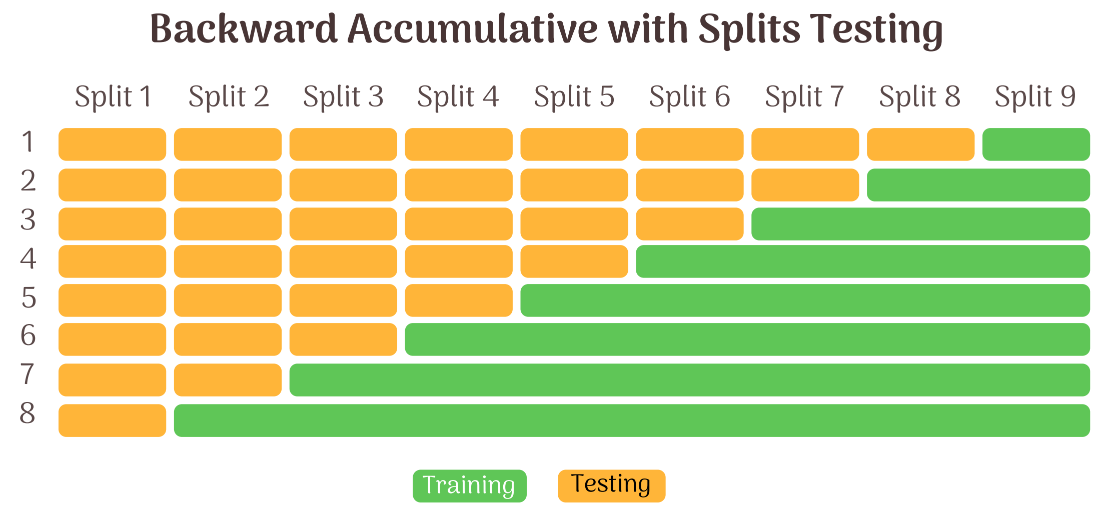


Figure 5.9: Backwards Accumulative with Splits Testing Flow Illustration

Forward/Backward Accumulative Flows vs. Rolling Cross-Validation

Referring back to Chapter 2, it disuses the rolling cross validation for time series. Forward/backward accumulative flows look similar to rolling cross-validation with an expanding window, but they are different. The difference is because of the design choices. Rolling cross validation is designed to fit the stock market needs. Forward/Backward Accumulative Flows design or the malware detection requirements. The main differences are as follows:

1. The testing in forward/backward accumulative flows is done with all other splits combined or individually. Rolling cross-validation with an expanding window is done only on one split (with or without a gap).

2. Rolling cross-validation with an expanding window has only a forward flow for the future. We have two different backward flows.

5.3.7 Scoring method

To check the model performance, we will consider two primary metrics: AUC score and drop score.

AUC

We reported the results in AUC (Area under the ROC Curve) as the performance metric. In all our malware experiments, the word *'performance'* will indicate AUC [9].

Drop Score

The drop score shows how much the AUC score drops from training to testing. In other words, it is the difference between the training and testing score. If there are many results in the testing, we will consider the lowest value.

5.4 Experiments

5.4.1 SOREL Dataset Files

The SOREL dataset contains several parts. We will not use the raw binaries of malware. We will use processed data. This part contains three different files:

- meta.db: It contains index and labels for the files. (size: 3.5GB)
- ember_features: LMDB directory with baseline features. (size: ≈72GB)
- pe_metadata: LMDB directory with full metadata dumps (size: ≈480GB)

5.4.2 Software and Hardware Setup

For creating the model, we used Pytorch [58] in Python. The training model is executed with different GPUs (Nvidia 2080ti and Nvidia Titan V). With CPU with 12 cores and 24 threats (AMD Ryzen Threadripper 1920x). And a 64GB of memory size. ALPlease rewrite the previous paragraph – too many short and incomplete sentences

5.5 Results and Analysis

This section will show the results of the Base model and Aloha model in different flow types. We will also analyse the results in different malware groups. The full results of each malware group with different flows can be found in the thesis repository 1.8.

5.5.1 Normal Flow

Table 5.3 shows the average result of all groups of malware of the base model. Figure 5.5.1visualises the average results in testing of each split in Base Model. Table 5.4 shows the average results of Aloha for normal flow and figure 5.5.1 visualises the average results of the performance. The black cell numbers in the tables indicate that the split train and tested with itself, including the validation set of the split.

Split 1 performs the worst in testing with itself (Base: 86%, Aloha: 84%), and the drop (Base: 24%, Aloha: 24%). This might be expected as split 1 only represents around 1.37% of the total dataset. Split 2 performs well when tested with itself (Base: 95%, Aloha: 86%) but shows a significant drop (Base: 16%, Aloha: 19%).

We can also notice that the splits in the middle (split 3, 4, 5, 6, 7) show the least drop (Base: 7-9%, Aloha: 11-16%). For Split 9 and 8, which consist of 48.66% of the dataset, the performance when they tested with themself are the highest, but the drop of both is also high (Split 8 - Base: 13%, Aloha 33%) (Split 9 - Base: 14%, Aloha: 18%). This might indicate that moving farther in time shows a more significant drop in the performance, even with having a bigger dataset (Split 9 and 8).

In most cases, the Base model, which is a smaller model, performs better than the Aloha model. The Aloha model has a higher drop compared with the Base model. The average drop in the Base model is around 12%. In comparison, the average drop of the Aloha model is 17%. It might indicate that there is overfitting of the bigger model that might be not obvious during the training. It also might show that a bigger model might need more retraining over time.

In general, we can find that the size of the split matters to a certain extends (split 1 and 2 compared with split 9 and 8). However, the time effect plays a big role as well (split 3 to 7). And if we have a bigger model, we need to pay attention to the overfitting problem.

Table 5.3: Time-Based Splits - Normal Flow - Base Model - Average Results.

Average	test										
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop	All data
Split 1	0.861	0.723	0.732	0.628	0.617	0.693	0.799	0.835	0.813	24.38%	0.778
Split 2	0.937	0.956	0.941	0.847	0.795	0.821	0.886	0.905	0.912	16.16%	0.889
Split 3	0.969	0.969	0.972	0.901	0.881	0.893	0.906	0.924	0.930	9.10%	0.904
Split 4	0.880	0.917	0.934	0.962	0.884	0.887	0.883	0.896	0.902	8.19%	0.905
Split 5	0.878	0.883	0.903	0.917	0.962	0.898	0.893	0.914	0.910	8.40%	0.905
Split 6	0.917	0.906	0.927	0.917	0.940	0.979	0.965	0.958	0.955	7.33%	0.959
Split 7	0.925	0.916	0.930	0.895	0.919	0.954	0.988	0.982	0.969	9.29%	0.970
Split 8	0.918	0.904	0.915	0.861	0.884	0.925	0.973	0.994	0.987	13.34%	0.967
Split 9	0.903	0.899	0.905	0.852	0.857	0.915	0.963	0.990	0.994	14.12%	0.961

Figure 5.10: Time-Based Splits - Normal Flow - Base Model

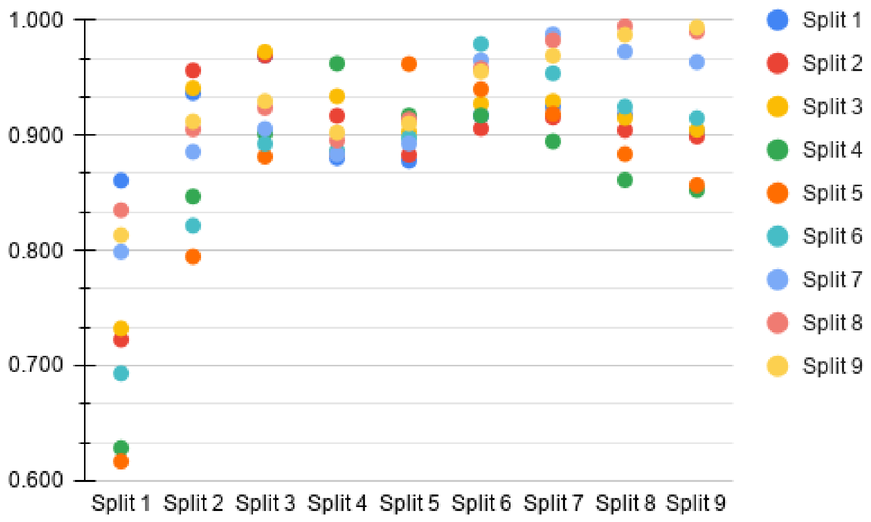
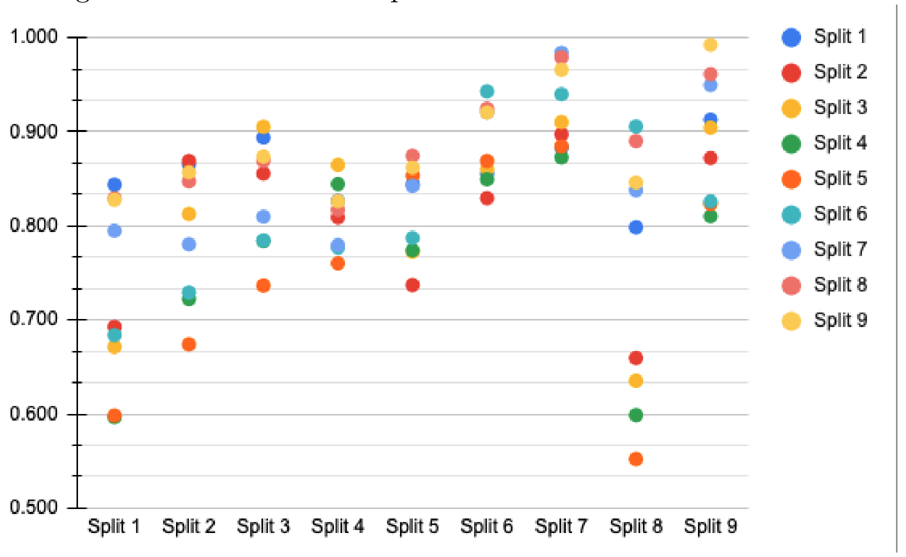


Table 5.4: Time-Based Splits - Normal Flow - Aloha Model - Average Results.

Average	test										
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop	All data
Split 1	0.844	0.693	0.671	0.597	0.598	0.684	0.795	0.829	0.828	24.71%	0.771
Split 2	0.865	0.869	0.813	0.722	0.674	0.729	0.780	0.847	0.857	19.50%	0.813
Split 3	0.894	0.856	0.905	0.784	0.737	0.785	0.810	0.869	0.874	16.87%	0.846
Split 4	0.827	0.809	0.865	0.844	0.760	0.777	0.780	0.817	0.826	8.42%	0.821
Split 5	0.844	0.737	0.772	0.774	0.853	0.787	0.842	0.874	0.862	11.63%	0.844
Split 6	0.855	0.829	0.860	0.849	0.869	0.943	0.921	0.925	0.920	11.35%	0.919
Split 7	0.883	0.897	0.910	0.873	0.884	0.940	0.984	0.979	0.966	11.12%	0.959
Split 8	0.798	0.660	0.635	0.599	0.552	0.906	0.838	0.890	0.846	33.79%	0.961
Split 9	0.913	0.872	0.904	0.810	0.823	0.826	0.950	0.961	0.992	18.21%	0.959

Figure 5.11: Time-Based Splits - Normal Flow - Aloha Model



5.5.2 Forward Accumulative with the Whole Testing

Tables 6.5 and 6.6 show the average results of the Base and Aloha models, respectively. The black cells indicate the training part, and the white cells indicate the splits included in the testing. The drop here indicates the difference between the training result and the testing result.

In this flow, both models perform similarly. The general observation is that the testing performance increases each time we include more splits in the training part. The drop also decreases when the training size increases. After training a model with one year of data (split 1 to 4), we can expect an average drop of around 3% in the next 15 months (Split 5 to 9). Similarly, if we train our model with one and half years' worth of data (split 1 to 6), we can expect an average drop of around 1% for the next nine months (split 7 to 9). We can use such calculation as a general guideline to decide the time of data needed to retrain the model.

Table 5.5: Time-Based Splits - Forward Accumulative with Whole Testing - Aloha Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	Drop
1	85.23%								77.36%	7.87%
2		95.92%							88.38%	7.54%
3			96.48%						91.45%	5.03%
4				97.51%					94.47%	3.04%
5					98.07%				95.94%	2.13%
6						98.16%			97.14%	1.02%
7							98.59%		98.41%	0.18%
8								98.99%	98.82%	0.17%

Table 5.6: Time-Based Splits - Forward Accumulative with Whole Testing - Base Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	Drop
1	83.14%								77.01%	6.13%
2		89.34%							80.60%	8.74%
3			95.97%						87.12%	8.86%
4				95.93%					92.26%	3.67%
5					97.27%				94.89%	2.38%
6						97.95%			97.05%	0.89%
7							98.50%		98.33%	0.17%
8								98.90%	98.80%	0.10%

5.5.3 Forward Accumulative with Splits Testing

The testing is done in the previous flow by combining all remaining splits after the testing. This flow is similar in the training but different in the testing. The testing will show the results of each split separately.

Table 5.7 shows the average results of the Base model. And Table 5.8 shows the results of the Aloha model.

Like the Forward Accumulative with Whole Testing flow, the performance increases over time by adding more splits in the training and the drop decreases. In the first stage of the training (only with split 1), there is a significant drop of 23-24% which has been not clear from the previous flow (was 6-7%). The difference in drop was carried out in stages 2 and 3. From stage 4 to stage 8, the result is close to the previous flow in the Base model. It might indicate that we should start with at least one year worth of data to have a model that can last for the future. However, more data is better, as the results show.

Another observation is that there is no direct link between the performance and closeness of the testing split. For example, in stage 3 (Training includes split 1 to 3), the highest performing result is for split 8 (split started one year after a trained model) with 93% and a drop of around 3%. The base model shows slightly better results compared with the Aloha model in terms of performance and drop.

Table 5.7: Time-Base Splits - Forward Accumulative with Splits Testing - Base Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	Drop
1	85.64%	72.16%	72.01%	64.60%	62.50%	69.23%	79.29%	81.70%	82.50%	23.14%
2		95.44%	92.81%	84.93%	81.49%	84.19%	86.61%	89.23%	90.18%	13.95%
3			96.49%	87.94%	85.60%	87.73%	91.22%	93.25%	92.86%	10.90%
4				97.40%	93.39%	92.68%	93.29%	95.08%	95.01%	4.72%
5					98.01%	94.50%	95.63%	96.76%	96.05%	3.50%
6						98.15%	97.07%	97.29%	96.94%	1.22%
7							98.58%	98.69%	98.02%	0.56%
8								98.98%	98.82%	0.16%

Table 5.8: Time-Base Splits - Forward Accumulative with Splits Testing - Aloha Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	Drop
1	83.14%	67.53%	64.98%	59.65%	58.50%	68.05%	79.65%	83.26%	83.54%	24.64%
2		89.34%	84.14%	75.41%	70.77%	76.01%	76.74%	83.32%	83.55%	18.57%
3			95.97%	87.18%	84.07%	86.44%	88.05%	91.79%	92.22%	11.91%
4				95.93%	90.14%	89.83%	89.65%	93.44%	93.62%	6.28%
5					97.27%	92.20%	94.01%	95.75%	94.93%	5.07%
6						97.95%	96.92%	97.24%	96.83%	1.11%
7							98.50%	98.60%	97.97%	0.53%
8								98.90%	98.80%	0.10%

5.5.4 Backward Accumulative with the Whole Testing

In this flow, we start from the future and go back to the past with combined splits in the testing. Table 5.9 shows the average results of the Base model. And Table 5.10 shows the average results of the Aloha model.

Comparing both Forward Accumulative with the Whole Testing, the Backward Accumulative with the Whole Testing shows a lesser drop. It might be because of the size of the data. The last splits have a significantly large size difference compare with the starting splits. The difference in the drop is around 2-3%. However, the results of the drop did not go to 1% or lower in this flow. But for the Forward Accumulative with Whole Testing flow, the last three stages got 1% or lower drop (refer to Table 5.5 and 5.6). It might indicate that the type and quality of the data to train with also matter not only the size.

We can also notice that increasing the amount of the data in training led to better testing accuracy, but not by a considerable margin. For stage 1 (train with split 9) the Base model and Aloha model give around 94%. And for the last stage (training from split 2 to 9) the performance increased to around 96%. In general, both models perform similarly in this flow.

Table 5.9: Time-Base Splits - Backward Accumulative with Whole Testing - Base Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
1								94.75%	99.41%	4.66%
2							94.42%	■	99.51%	5.09%
3						95.50%	■	■	99.43%	3.93%
4					96.02%	■	■	■	99.36%	3.33%
5				96.11%	■	■	■	■	99.29%	3.18%
6			96.63%	■	■	■	■	■	99.24%	2.61%
7		96.33%	■	■	■	■	■	■	99.20%	2.86%
8	96.77%	■	■	■	■	■	■	■	99.17%	2.40%

Table 5.10: Time-Base Splits - Backward Accumulative with Whole Testing - Aloha Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
1								94.33%	99.21%	4.89%
2							93.90%	■	99.47%	5.57%
3						94.43%	■	■	99.26%	4.83%
4					96.18%	■	■	■	99.35%	3.17%
5				95.70%	■	■	■	■	99.25%	3.54%
6			96.58%	■	■	■	■	■	99.22%	2.64%
7		96.22%	■	■	■	■	■	■	99.20%	2.98%
8	96.75%	■	■	■	■	■	■	■	99.17%	2.42%

5.5.5 Backward Accumulative with Splits Testing

This flow is similar to the previous flow; the difference is that now we test in each split instead of testing with combined splits. The result here is more detailed. Table 5.11 shows the average results of the Base model. And Table 5.12 shows the average results of the Aloha model.

The results here show a more significant drop than Backward Accumulative with Whole Testing flow (around 10% difference at the beginning). Compared with the Forward Accumulative with Splits Testing, this flow performing better in the drop, but it cannot go less than the 1%. The base model and Aloha model perform similarly.

Table 5.11: Time-Base Splits - Backward Accumulative with Splits Testing - Base Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
1	90.57%	89.89%	90.05%	85.38%	85.19%	91.59%	96.38%	98.99%	99.41%	14.22%
2	92.76%	91.46%	92.13%	88.30%	90.54%	93.38%	97.52%		99.51%	11.21%
3	94.13%	93.95%	95.07%	92.06%	93.73%	96.57%			99.43%	7.38%
4	94.69%	94.77%	96.11%	93.74%	95.89%				99.36%	5.62%
5	94.26%	95.34%	96.65%	95.28%					99.29%	5.03%
6	95.78%	95.55%	97.15%						99.24%	3.69%
7	96.10%	96.18%							99.20%	3.09%
8	96.84%								99.17%	2.33%

Table 5.12: Time-Base Splits - Backward Accumulative with Splits Testing - Aloha Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
1	89.59%	87.86%	87.22%	83.50%	85.04%	90.42%	95.85%	98.78%	99.21%	15.71%
2	90.27%	90.91%	91.88%	87.30%	89.03%	92.38%	97.50%		99.47%	12.18%
3	92.93%	92.50%	93.67%	90.83%	92.49%	95.89%			99.26%	8.42%
4	94.30%	94.89%	96.23%	94.09%	96.29%				99.35%	5.27%
5	94.51%	95.14%	96.39%	94.84%					99.25%	4.74%
6	95.08%	95.61%	97.08%						99.22%	4.15%
7	95.86%	96.08%							99.20%	3.34%
8	96.68%								99.17%	2.50%

5.5.6 Malware Groups

In this section, we will talk about the behaviour that we notice for each malware group. The full results can be found in the thesis repository 1.8. Table 5.13 shows the average, minimum, and maximum for the drop and performance of each group. It only includes the results of testing. Figures 5.5.6 and 5.5.6 visualise the same results. The results are for both models (Base and Aloha). The negative number in the minimum means that the performance in the testing is more than the training. In other words, there is gaining in the performance.

The overview of the results is that the average drop of all types is 8.7%. The minimum average drop is from the installer label with 5.7%. And the maximum average drop is from a crypto miner with 13.9%. The average performance result of all labels is 90.5%. The minimum average performance is from a crypto miner with 84.24%. And the maximum average performance is from a file infector with 93.31%. In general, this shows that installer and file infector groups are easier to detect, and crypto miner seems to be the hardest. The file infector group get the highest average performance. Moreover, the installer group get the lowest average drop.

Table 5.13: Malware Groups (Time-Based) - Drop & AUC - Average, Min, Max.

	Drop			AUC		
Malware Group	Minimum	Maximum	Average	Minimum	Maximum	Average
adware	0.23%	53.00%	9.13%	38.86%	98.71%	90.27%
flooder	-6.66%	56.68%	11.25%	34.31%	99.22%	89.17%
ransomware	-0.26%	54.01%	9.37%	45.78%	99.70%	91.81%
dropper	-0.33%	27.73%	7.49%	58.64%	98.73%	90.58%
spyware	0.31%	34.31%	7.86%	62.30%	98.80%	91.44%
packed	-0.66%	33.33%	8.54%	51.60%	99.20%	90.36%
crypto miner	-0.61%	43.80%	13.90%	26.33%	99.21%	84.24%
file infector	-0.19%	30.17%	6.36%	61.25%	99.62%	93.67%
installer	0.11%	50.45%	5.70%	39.42%	99.03%	93.31%
worm	-0.45%	30.53%	7.62%	60.74%	99.54%	92.24%
downloader	0.38%	42.19%	8.75%	49.74%	98.82%	89.18%

Figure 5.12: Illustration of Malware Groups (Time-Based) AUC Results (Average, Min, Max)

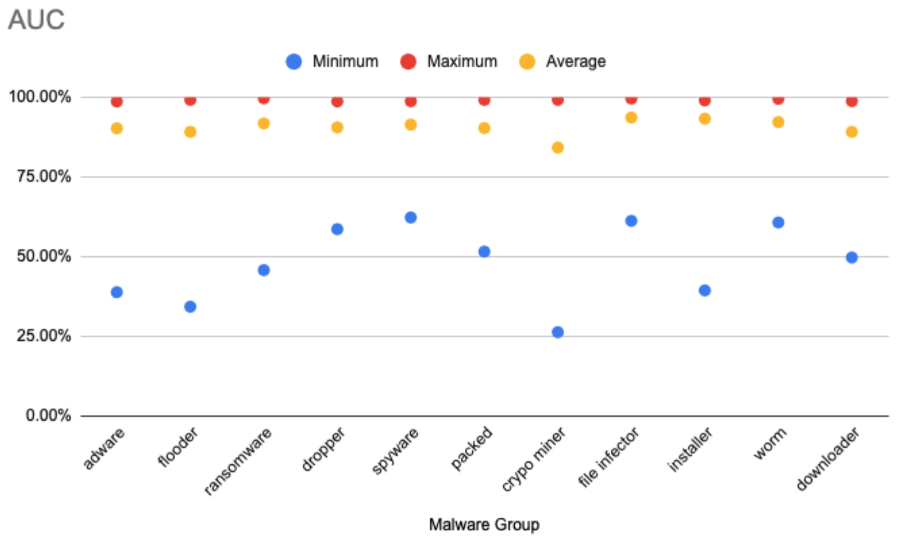
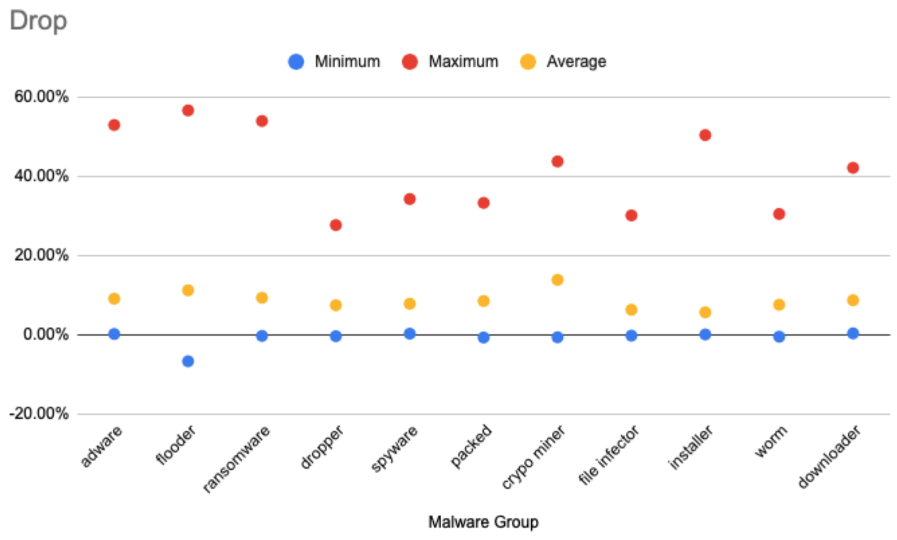


Figure 5.13: Illustration of Malware Groups (Time-Based) Drop Results (Average, Min, Max)



Adware

Adware shows an average drop of 9.13% from all training flows. The maximum drop is 53% when the Aloha model is trained with split 8 and tested with split 5 in normal flow. In the same flow with the same split, the Base model gives a drop of 17%. The drop is not because of the sample size of this label. Split 8 contains 2,148,282 samples for adware and split 5 contains 1,769,849 samples. The drop seems a result of difficulty by the model to generalise the detection for this type of malware.

The Forward Accumulative with Whole Testing flow shows that we can expect less than 1% of the drop for the next nine months combined after one and a half years of training. Furthermore, we can expect around 2% of the drop for the Forward Accumulative with Splits Testing with the same period of training time.

The average performance of all forward accumulative flows with both models is 89.5%, and the backward accumulative flows is 93.6%. It indicates that testing for the past adware performs better than testing for the future one in average performance. The average drop of all forward flows is 4.5% and for the backward ones is 7.6%. That means testing in past flooder samples shows a more considerable drop.

Flooder

With all flows, flooder malware shows an average drop of 11.25% and an average performance of 89.17%. Flooder gets the maximum (56.6%) and minimum (-6.6%) drop out of all groups. It is due to the small sample size compared with others. The least minimum performance for a flooder is 34%.

Based on the results of Accumulative with Whole Testing flow, we can expect a drop of less than 1% in the model after we train a model with one year and three months' worth of data (Both models - Base and Aloha). Moreover, we can expect a drop of 2% on the same period with the Accumulative with Splits Testing flow.

The average performance of all forward accumulative flows with both models is 87%, and the backward accumulative flows is 94%. It indicates that testing in past flooder malware performs better than testing in a future one in terms of average performance. The average drop of all forward flows is 8.2% and for the backward flows is 5.9%. That means testing in past flooder samples shows a more significant drop.

Ransomware

The average performance result of all models for ransomware from all flows is 91.8%. And the average drop from all flows is 9.3%. Ransomware gets the second-highest average maximum drop after flooder with a drop of 54%.

Based on the results of the Accumulative with Whole Testing flow, we can expect a drop of 1% or less in the model performance after we train a model with one year and a half's worth of data (Both models - Base and Aloha). Moreover, the results are consistent with the Accumulative with Splits Testing flow in the same period.

The average performance of all the forward accumulative flows with both models is 91.1%, and the backward accumulative flows is 95%. It indicates that testing in past ransomware malware performs better than testing in future ones in terms of average performance. The average drop of all forward flows is 6.3% and for the backward flows is 6.6%. That means testing in past ransomware samples shows a similar average drop as testing for future samples.

Dropper

The average performance result of models in the dropper is 90.5%. And the average drop is 7.5%. The dropper malware models show the lowest maximum drop compared with all other groups with a drop of 27.7%.

Based on the results of the Accumulative with Whole Testing flow, we can expect a drop of less than 1% in the model performance after we train a model with one year and a half worth of data (Both models - Base and Aloha). Furthermore, the results are consistent with the Accumulative with Splits Testing flow in the same period.

The average performance of all forward accumulative flows with both models is 89.2%, and the backward accumulative flows are 94.1%. It indicates that testing in past dropper malware performs better than testing in future ones in terms of average performance. The average drop of all forward flows is 5.4% and for the backward flows is 5.5%. That means testing in past dropper samples shows a similar average drop as testing for future samples.

Spyware

The average performance result of all models for spyware from all flows is 91.4%. And the average drop from all flows is 7.8%. Spyware shows the highest minimum AUC performance of 62.3%.

Based on the results of Accumulative with Whole Testing flow, we can expect a drop of less than 1% in the model performance after we train a model with one year and a half worth of data (Both models - Base and Aloha). Moreover, the same results are shown in the Accumulative with Splits Testing flow.

The average performance of all forward accumulative flows with both models is 88.9%, and the backward accumulative flows is 95.5%. It indicates that testing in past spyware malware performs better than testing in future one in term of average performance. The average drop of all forward flows is 8.2%, and for the backward flows is 4%. That means testing in past spyware samples shows a more significant drop than testing for future samples.

Packed

The average performance result of all models for packed from all flows is 90.3%. And the average drop from all flows is 8.5%. The maximum drop of all the models is 33.3%.

Based on the results of Accumulative with Whole Testing flow, we can expect a drop of 1% or less in the model performance after we train a model with just one year worth of data (Both models - Base and Aloha). Based on Accumulative with Splits Testing flow, we need one and half years worth of training to reach the same performance for the next nine months

The average performance of all forward accumulative flows with both models is 89.3%, and the backward accumulative flows is 93.7%. It indicates that testing in the past with packed malware performed better than testing in future one in terms of average performance. The average drop of all forward flows is 6.1% and for the backward flows is 6.2%. That means testing in the past packed samples show a similar average drop as testing for future samples.

Crypto Miner

The average performance result of all models for crypto miner from all flows is 84.2%. And the average drop from all flows is 13.9%. Crypto miner gets the worst average performance out of all groups with an average of 84.2%. Moreover, it also gets the highest average drop result of 13.9%. This group of malware seems to be the most challenging one for the models.

Based on the results of Accumulative with Whole Testing flow, we can expect a drop of less than 1% in the model performance after we train a model with one year and nine months' worth of data (Both models - Base and Aloha). Furthermore, the results are consistent with the Accumulative with Splits Testing flow in the same period.

The average performance of all forward accumulative flows with both models is 81.8%, and the backward accumulative flows is 90.6%. It indicates that testing in past crypto-miner malware performs better than testing in the future one in terms of average performance. The average drop of all forward flows is 9% and for the backward flows is 10.8%. That means testing for a future crypto miner gets better results than testing for the past one.

File Infector

The average performance results of all models for the file infector from all flows is 93.6%. It is the highest average performance out of all groups. And the average drop from all flows is 6.3%, which is the second-best result after installer. The maximum drop of all the models is 30.1%.

Based on the results of Accumulative with Whole Testing flow, we can expect a drop of less than 1% in the model performance after we train a model with one year and a half worth of data (Both models - Base and Aloha). Moreover, the results are consistent with the Accumulative with Splits Testing flow in the same period.

The average performance of all forward accumulative flows with both models is 92.2%, and the backward accumulative flows is 97.1%. It indicates that testing in past file infector malware performs better than testing in the future one in terms of average performance. The average drop of all forward flows is 5.6%, and for the backward flows is 3%. That means testing in past file infector samples shows a better average drop than testing for future samples.

Installer

The average performance result of all models for an installer from all flows is 93.3%. It is the second-highest average performance after the file infector group. And the average drop from all flows is 5.7%, which is the best average result out of all groups. The maximum drop of all the models is 50.4%.

Based on the results of the Accumulative with Whole Testing flow, we can expect a drop of 1% or less in the model performance after we train a model with one year and a half worth of data (Both models - Base and Aloha). Furthermore, the results are consistent with the Accumulative with Splits Testing flow in the same period.

The average performance of all forward accumulative flows with both models is 92.5%, and the backward accumulative flows is 96%. It indicates that testing in past installer malware performs better than testing in a future one in terms of average performance. The average drop of all forward flows is 4.2% and for the backward flows is 3.6%. That means testing in past installer samples shows a similar average drop as testing for future samples.

Worm

The average performance result of all models for the worm group is 92.2%. And the average drop from all flows is 7.6%. The maximum drop of all the models is 30.5%.

Based on the results of Accumulative with Whole Testing flow, we can expect a drop of less than 1% in the model performance after we train a model with one year and a half worth of data (Both models - Base and Aloha). Moreover, the results are consistent with the Accumulative with Splits Testing flow in the same period.

The average performance of all forward accumulative flows with both models is 90.9%, and the backward accumulative flows is 95.3%. It indicates that testing in past worm malware performs better than testing in future ones in terms of average performance. The average drop of all forward flows is 5.8% and for the backward flows is 5.1%. That means testing in past worm samples shows a similar average drop as testing for future samples.

Downloader

The average performance result of all models for the downloader group from all flows is 89.1%. And the average drop from all flows is 8.7%. The maximum drop of all the models is 42.1%.

Based on the results of Accumulative with Whole Testing flow, we can expect a drop of less than 1% in the model performance after we train a model with one year and three months of data (Both models - Base and Aloha). Moreover, one year and half of training data to get the same performance with Accumulative with Splits Testing flow in the same period.

The average performance of all forward accumulative flows with both models is 87.6%, and the backward accumulative flows is 93.6%. It indicates that testing in past downloader malware performs better than testing in the future one in terms of average performance. The average drop of all forward flows is 6.5% and for the backward flows is 5.7%. That means testing in past downloader samples shows a similar average drop as testing for future samples.

5.5.7 Training time needed for each malware group

Table 5.14 shows how much training time (in months) is needed for each malware group before reducing the drop to be less than 5% or 1%. The results are based on the average results of both models (Base and Aloha).

Table 5.14: Training time needed to drop less than 5% and 1% (m=Months)

	Forward				Backward			
	Whole		Splits		Whole		Splits	
Malware Group	5%	1%	5%	1%	5%	1%	5%	1%
adware	9m	18m	15m	21m	9m	-	24m	-
flooder	9m	15m	15m	18m	12m	-	18m	-
ransomware	12m	18m	18m	21m	9m	24m	18m	24m
dropper	12m	18m	18m	21m	12m	-	18m	-
spyware	18m	21m	21m	21m	9m	24m	12m	24m
packed	12m	18m	15m	18m	12m	-	15m	-
crypo miner	12m	21m	18m	21m	-	-	-	-
file infector	6m	18m	18m	18m	3m	24m	12m	24m
installer	6m	21m	18m	21m	3m	-	12m	-
worm	15m	18m	15m	18m	9m	24m	15m	24m
downloader	12m	-	15m	-	12m	-	18m	-

5.6 Limitation

There are several limitations of the experiments:

- **Dataset Duration:** The dataset started from January 2017 to mid-April 2019 (around two years and three months). We consider the dataset to be the best in terms of the size and duration at the time of writing. The actual duration to understand the problem might be longer.

- **Machine Learning Model:** We did not test all the machine learning models in such split and flow of learning. While we attempt to design a better malware detector in a later chapter, the result reported is based on the tested models. Different classifiers might show different results.
- **Distribution of tags in splits:** Based on the time window constraint, there will be an imbalance of each split's distribution. It will make some groups more dominant in the split, making it not an ideal setup from a data science perspective. Nevertheless, this might also represent the real-life scenario, where some malware groups are more spread than others.

5.7 Summary

We can summarise this chapter in the following points:

- We propose a different way of evaluating the performance of a malware classifier that tries to understand the performance of the classifier model and the behaviour of the malware over time. Instead of splitting the dataset into a training and testing split, we propose dividing the dataset into several splits based on time. For the experiments done here, we divide the dataset into nine splits based on the time (three months each).
- To train and test, we have two machine learning models with five different flows. The flows are Normal Flow, Forward Accumulative with the Whole Testing, Forward Accumulative with Split Testing, Backward Accumulative with the Whole Testing, Backward Accumulative with Split Testing.
- The results from Normal Flow indicates the large size of the dataset might make the model prone to overfitting, and small size tends to perform worse. The splits in the middle (split 3, 4, 5, 6 and 7) perform better and drop less compared with the splits in the edge of the time (split 1, 2, 8 and 9). The result also shows that a bigger model (Aloha model) drops more than a smaller model (Base model) and performs less. A bigger model tends to have a problem of overfitting.

- The general observation in the Forward Accumulative with Whole Testing flow, that the increase of training size over time increases the model's performance and decreases the drop. We can use the general guideline to retrain the mode based on the results given by this flow. For example, training a model with one year's worth of data should give around a 3% drop for the next 15 months.
- Based on the Forward Accumulative with Split Testing flow, it shows there is a more significant drop happen in small data size. The previous flow has smoothed out this drop. The results from this flow indicate that we should have one year's worth of data to train with to get a good malware classifier.
- The Backward Accumulative with Whole Testing flow shows a better result compared with the Forward Accumulative with Whole Testing flow with a small margin (3-4% in drop). It might be because of the amount of data that started with training. However, the drop of this flow did not go 1% or below, compared with the other flow. It shows that the type of the data matter as well in the trained model. And it might not generalise well.
- Backward Accumulative with Split Testing flow shows more drop compared with Backward Accumulative with Whole Testing (in stage one around 10% difference). It also shows better results than the Forward Accumulative with Split Testing flow, but the drop did not go to 1% or less.
- The average results of each group of malware analysis shows in general that the installer and file infector groups seem to be the easiest to detect, while the crypto miner is the hardiest.

In the next chapter, we will follow the same methodology of this chapter. However, the change will be in the way of splitting. Instead of splitting the data based on a fixed time, the split is based on the size. All splits will have the same size regardless of the duration of the time.

Chapter 6

Malware Detection Performance Over Time - Size-Based Splits

6.1 Chapter Overview

In the last chapter, we investigated the machine learning model's performance in malware detection in the time-based split. In this chapter, we will follow the same process, but we will change how we divide the dataset. The split in this chapter will maintain the equal size of each split regardless of the duration of the time.

6.2 Introduction

In the last chapter, we discuss the results when the splits are based on a fixed time window (3 months) regardless of the size of samples in each split. The argument for that is that we want to preserve the natural occurrence of malware collection in a security lab. It also helps in scheduling a time to retrain our model with a new sample that happens later. It will also allow the understanding of which malware group needs more updates and retraining.

The SOREL dataset that we used is not distributed evenly across time. Figure 6.1 shows the sample distribution of the dataset over time. Even thorough that is what might happen in a real security lab, where malware coming has different spikes and lows across time, we want to divide the splits based on the equal size of samples. The reasons to do so are as following:

- We want to retrain the model based on the number of samples that we have. For example, if we reach a certain number of samples, we will retrain the model.
- We want to check the result of the last chapter. How many observations will hold in this different way of splitting?

For that, the splits will preserve the timing. It means that split 1 is older than split 2 in time. We will follow the same training and testing flows as in the last chapter. Then, we will analyse the results of each flow and each malware group.

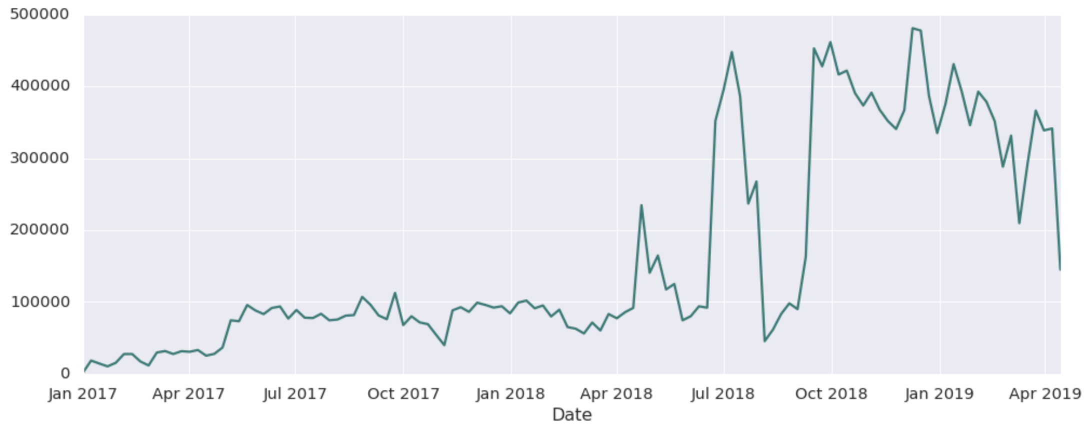


Figure 6.1: Data distribution of the SOREL dataset over time

6.3 Methodology

The methodology that we will use will be the same as the last chapter. The change is in the way we split the dataset. The rest will be the same. It includes the machine learning models (Base and Aloha), the flows of training and testing (1- Normal Flow, 2-Forward Accumulative with the Whole Testing, 3- Forward Accumulative with Splits Testing, 4- Backward Accumulative with the Whole Testing, and 5- Backward Accumulative with Splits Testing) and scoring metrics (AUC and Drop). We will also divide the dataset into nine different splits.

Table 6.1 shows the distribution of each malware group in each split. Figure 6.2 shows the same information with heat map. The last 5% of each split is used as a validation set of that split.

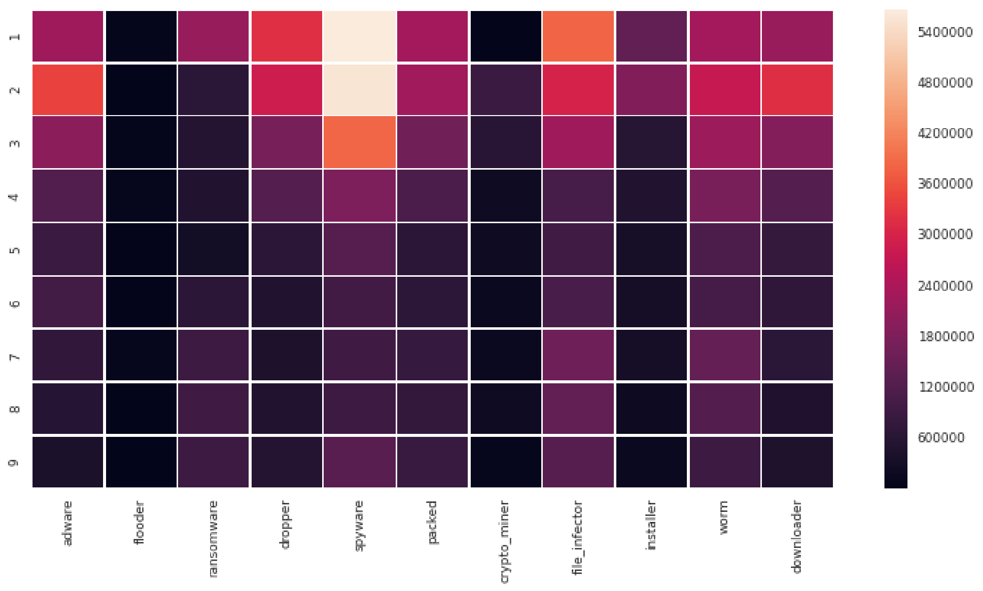


Figure 6.2: Size-Based Splits - Malware Groups Distribution In Each Split

Table 6.1: Malware Groups Distribution of each split in Size-Based Splits.

Split	adware	flooder	ransomware	dropper	spyware	packed	crypto_miner	file_infector	installer	worm	downloader
1	2247030	78230	2141941	3217705	5673328	2325317	46629	3824932	1433034	2304966	2164119
2	3449806	29112	647849	2879733	5582786	2291935	897386	3019662	1854412	2798329	3193785
3	1996404	73779	549373	1724224	3840161	1656339	619196	2256314	601774	2217659	1916140
4	1212661	98890	523152	1268834	1818882	1143226	237801	1062910	503370	1756050	1273102
5	893833	40213	283515	665515	1284452	678947	223439	985250	359442	1162287	798972
6	1019322	54401	675106	524044	994985	694099	177848	1117453	335376	1055233	717136
7	749769	85395	918014	452583	950744	813897	188260	1623768	328586	1488488	639603
8	589704	29527	950300	503554	942500	784122	220019	1437959	195809	1253901	492733
9	430289	17119	921506	565925	1318366	866557	101498	1280773	179087	938152	471449

6.4 Results and Analysis

This section will show the results of the Base model and Aloha model in different flow types. We will also analyse the results in different malware groups. The full results of each malware group with different flows are in the thesis repository 1.8.

6.4.1 Normal Flow

Table 6.2 shows the average result of the base model for all groups of malware. Figure 6.4.1 visualises the average results in testing of each split in Base Model. Table 6.3 shows the average results of Aloha for normal flow and Figure 6.4.1 visualises the average results. The black cell numbers in the tables indicate that the split was trained and tested with itself, including the validation set of the split.

The Base model performs better than the Aloha model and drops less. The average drop of the Base model is 12%, while the average drop of the Aloha model is 17.5%.

The best performance result of the normal flow is when the models are trained with split 3. In this split, the Base model drop is 4.8%, and the Aloha model gives a drop of 8.5%. The last five splits perform the worst (Split 5 to 9). Like a normal flow in time-based splits, there is no direct link between how close split to training with the performance.

Table 6.2: Size-Based Splits - Normal Flow - Base Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
Split 1	0.967	0.883	0.883	0.901	0.926	0.923	0.929	0.929	0.940	8.48%
Split 2	0.950	0.973	0.918	0.905	0.923	0.934	0.922	0.924	0.931	6.88%
Split 3	0.934	0.940	0.982	0.970	0.966	0.962	0.950	0.954	0.955	4.81%
Split 4	0.916	0.903	0.949	0.983	0.977	0.975	0.970	0.965	0.966	7.95%
Split 5	0.895	0.859	0.908	0.963	0.990	0.982	0.977	0.973	0.968	13.10%
Split 6	0.877	0.835	0.891	0.948	0.981	0.990	0.988	0.982	0.974	15.47%
Split 7	0.857	0.810	0.883	0.939	0.974	0.984	0.992	0.986	0.979	18.21%
Split 8	0.851	0.816	0.890	0.939	0.972	0.982	0.989	0.991	0.981	17.46%
Split 9	0.868	0.828	0.891	0.940	0.969	0.978	0.984	0.986	0.990	16.20%

Table 6.3: Size-Based Splits - Normal Flow - Aloha Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
Split 1	0.946	0.822	0.841	0.876	0.910	0.914	0.904	0.908	0.927	12.45%
Split 2	0.911	0.942	0.868	0.851	0.878	0.904	0.890	0.893	0.905	9.17%
Split 3	0.868	0.874	0.954	0.943	0.933	0.944	0.930	0.934	0.944	8.58%
Split 4	0.837	0.824	0.894	0.965	0.954	0.954	0.941	0.942	0.943	14.05%
Split 5	0.864	0.812	0.870	0.950	0.982	0.974	0.971	0.963	0.961	16.98%
Split 6	0.787	0.734	0.821	0.913	0.959	0.977	0.972	0.961	0.954	24.30%
Split 7	0.824	0.756	0.831	0.919	0.956	0.973	0.981	0.970	0.965	22.48%
Split 8	0.810	0.716	0.808	0.897	0.946	0.964	0.974	0.976	0.969	26.01%
Split 9	0.827	0.742	0.835	0.918	0.950	0.963	0.973	0.976	0.984	24.24%

Figure 6.3: Size-Based Splits - Normal Flow - Base Model

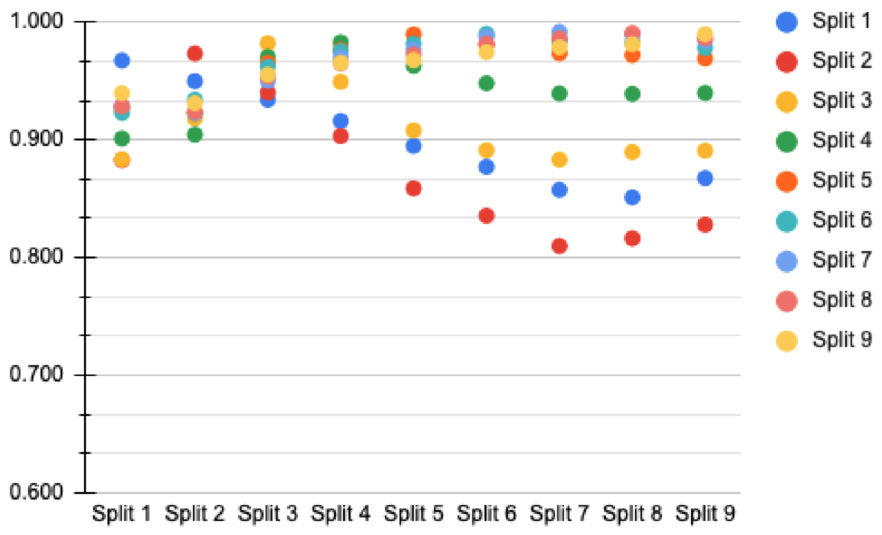
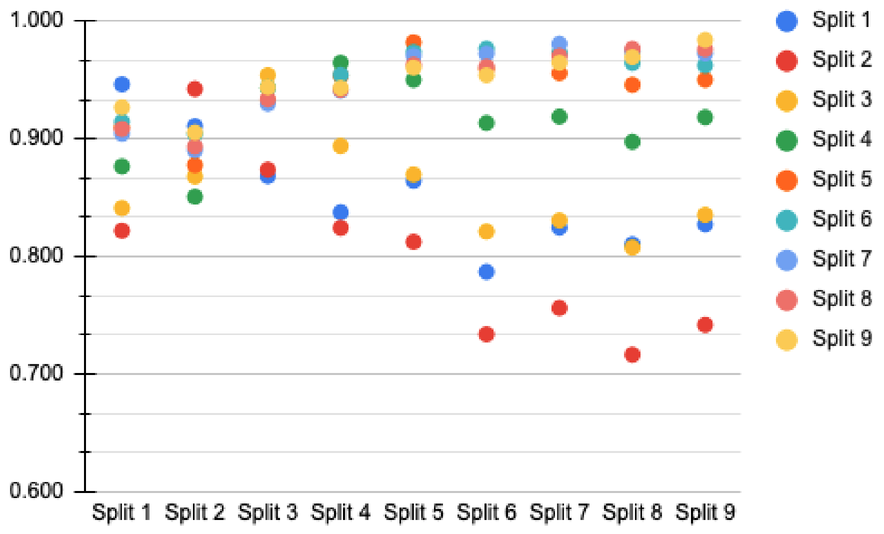


Figure 6.4: Size-Based Splits - Normal Flow - Aloha Model



6.4.2 Forward Accumulative with the Whole Testing

Table 6.4 shows the average results of Base model. Table 6.5 shows the average result of the Aloha model. The black cells indicate the training part, and the white cells indicate the splits included in the testing. The drop here indicates the difference between the training result and the testing result.

In the first and second stages, the Base model performs slightly better than the Aloha model and drops less. However, in the rest of the stages, both models perform similarly. The average drop of the Base model is 1.2%, while the average drop of the Aloha model is 1.34%.

From stage 3 (models trained with split 1 to 3) to the end (stage 8), the drop of both models is less than 1%. It is an indication that split 1 to 3 contains sufficient information about the malware that can perform well with many splits in the future. In other words, the size of the training samples from split 1 to 3 is around 33% of the total dataset and the size of the testing sample from split 4 to 9 is around 67% of the dataset. It means that the model that learns with quality samples can perform well in future data.

Table 6.4: Size-Based Splits - Forward Accumulative with Whole Testing - Base Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	Drop
1	97.00%								92.06%	4.94%
2		97.83%							95.10%	2.73%
3			98.20%						97.61%	0.60%
4				98.48%					97.81%	0.67%
5					98.69%				98.37%	0.32%
6						98.87%			98.81%	0.07%
7							99.03%		98.82%	0.21%
8								99.10%	98.99%	0.11%

Table 6.5: Size-Based Splits - Forward Accumulative with Whole Testing - Aloha Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	Drop
1	93.34%								87.57%	5.76%
2		96.98%							93.59%	3.39%
3			97.82%						97.05%	0.77%
4				98.38%					98.16%	0.22%
5					98.64%				98.38%	0.26%
6						98.85%			98.77%	0.07%
7							98.85%		98.78%	0.06%
8								99.05%	98.88%	0.17%

6.4.3 Forward Accumulative with Splits Testing

The previous flow shows the results of all remaining splits combined after the training. This flow is similar in training but different in testing. The testing will show the results of each split separately. Table 6.6 shows the average results of the Base model. Table 6.7 shows the results of the Aloha model.

In this flow, the Base model shows slightly better results compared with the Aloha model. The average drop of the Base model is 1.9%, while the average drop of the Aloha model is 3.05%.

Like the results of the previous flow, training from stage 3 to the end gives a drop of 1% or less. This result gives more evidence that the first three splits contain quality samples that makes it perform better.

Table 6.6: Size-Based Splits - Forward Accumulative with Splits Testing - Base Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	Drop
1	97.00%	0.895	0.896	0.914	0.925	0.923	0.919	0.921	0.935	7.54%
2		97.83%	94.14%	94.02%	95.16%	95.37%	95.50%	94.92%	95.22%	3.81%
3			98.20%	97.78%	97.66%	97.68%	97.65%	97.15%	97.19%	1.05%
4				98.48%	98.39%	98.08%	98.11%	97.48%	97.36%	1.12%
5					98.69%	98.48%	98.81%	98.29%	98.04%	0.65%
6						98.87%	99.23%	98.76%	98.36%	0.52%
7							99.03%	98.99%	98.62%	0.40%
8								99.10%	99.00%	0.10%

Table 6.7: Size-Based Splits - Forward Accumulative with Splits Testing - Aloha Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	Drop
1	93.34%	0.798	0.825	0.850	0.888	0.909	0.902	0.907	0.916	13.57%
2		96.98%	90.93%	90.59%	93.43%	94.96%	94.84%	94.42%	94.96%	6.39%
3			97.82%	97.28%	97.10%	96.99%	96.99%	96.70%	96.15%	1.67%
4				98.38%	98.45%	98.09%	98.17%	97.74%	97.57%	0.82%
5					98.64%	98.62%	98.70%	98.17%	97.77%	0.87%
6						98.85%	99.17%	98.70%	98.21%	0.64%
7							98.85%	98.97%	98.54%	0.30%
8								99.05%	98.89%	0.16%

6.4.4 Backward Accumulative with the Whole Testing

In this flow, we start from the future to go back to the past with combined splits in the testing. Table 6.8 shows the average results of the Base model. Table 6.9 shows the average results of the Aloha model.

The average drop of the Base model is 4.31%, while the average drop of the Aloha model is 4.74%. The performance of both models is comparable.

In all stages of this flow, both models did not get less than 2% of the drop or more than 97% in the performance. Even with the last stage (largest training size - trained with split 2 to 9 and tested with split 1), the models did not perform like that of the forward flows. It indicates that the earlier splits contain more valuable information about the malware than the later splits, regardless of the size.

The aim of these backward flows is to understand if the mode can perform well in past malware that have not been trained with it. The results from this flow show that the model struggles to learn from the past malware compared to future malware (from both forward flows).

Table 6.8: Size-Based Splits - Backward Accumulative with Whole Testing - Base Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
1								93.60%	99.05%	5.45%
2							95.09%		99.40%	4.30%
3						94.68%			99.47%	4.79%
4					94.41%				99.49%	5.08%
5				94.43%					99.49%	5.06%
6			95.57%						99.44%	3.87%
7		96.11%							99.37%	3.25%
8	96.54%								99.25%	2.71%

Table 6.9: Size-Based Splits - Backward Accumulative with Whole Testing - Aloha Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
1								90.55%	98.39%	7.84%
2							94.00%		99.24%	5.24%
3						94.73%			99.41%	4.67%
4					94.85%				99.48%	4.63%
5				94.07%					99.48%	5.41%
6			95.28%						99.41%	4.12%
7		96.01%							99.34%	3.33%
8	96.55%								99.23%	2.69%

6.4.5 Backward Accumulative with Splits Testing

This flow is similar to the previous flow; the difference is that now we test in each split instead of testing with combined splits. Table 6.10 shows the average results of the Base model. Table 6.11 shows the average results of the Aloha model.

The average drop of the Base model is 9.2%, while the average drop of the Aloha model is 10.11%. The base model performs slightly better than the Aloha model.

The results show a more considerable drop compared with the Backward Accumulative with Whole Testing flow (more than 10% difference at the earlier stages). Moreover, the highest drop happens when the model is tested with split 2.

Table 6.10: Size-Based Splits - Backward Accumulative with Splits Testing - Base Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
1	86.97%	81.59%	88.98%	94.27%	96.75%	97.66%	98.42%	98.57%	99.05%	17.46%
2	90.30%	86.62%	91.71%	95.73%	98.12%	98.79%	99.31%		99.40%	12.77%
3	91.96%	86.53%	91.66%	96.32%	98.46%	99.05%			99.47%	12.94%
4	92.10%	88.99%	92.65%	96.28%	98.74%				99.49%	10.50%
5	92.95%	90.81%	94.22%	97.73%					99.49%	8.68%
6	94.95%	93.94%	96.69%						99.44%	5.50%
7	95.85%	95.56%							99.37%	3.81%
8	96.56%								99.25%	2.70%

Table 6.11: Size-Based Splits - Backward Accumulative with Splits Testing - Aloha Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
1	83.05%	74.54%	83.31%	91.37%	94.60%	96.26%	97.43%	97.65%	98.39%	23.86%
2	88.40%	84.21%	89.72%	95.04%	97.78%	98.55%	99.14%		99.24%	15.03%
3	91.03%	88.07%	92.43%	95.90%	98.16%	98.85%			99.41%	11.34%
4	91.99%	90.15%	93.32%	96.87%	98.81%				99.48%	9.33%
5	91.67%	90.63%	93.80%	97.40%					99.48%	8.84%
6	95.70%	93.65%	96.21%						99.41%	5.76%
7	95.50%	95.29%							99.34%	4.05%
8	96.54%								99.23%	2.69%

6.4.6 Malware Groups

In this section, we will talk about the behaviour that we notice for each malware drop. The full results can be found in the thesis repository 1.8. Table 6.12 shows the average, minimum, and maximum for drop and performance of each group. It only includes the results of testing. And Figures 6.4.6 and 6.4.6 visualise the same results. The results are for both models (Base and Aloha). The negative number in the minimum means that the performance in the testing is higher than in the training. In other words, there is improvement in the performance.

The overview of the results is that the average drop of all types is 7.4%. The minimum average drop is for the installer label with 4.5%. Moreover, the maximum average drop is from crypto miner with 10.9%. The average performance result of all labels is 94.4%. The minimum average performance is for a crypto miner with 92.9%.

Furthermore, the maximum average performance is from the file infector with 96.6%. In general, this shows that installer and file infector groups are easier to detect, and crypto-miner seems to be the hardest. The file infector group get the highest average performance. Moreover, the installer group get the lowest average drop. These results here are consistent with the time-based splits. In general, the performance results here is slightly higher compared with the time-based splits.

Note that we will refer to the results of time-based splits with the word before for comparison reasons.

Table 6.12: Malware Groups (Size-Based) - Drop & AUC - Average, Min, Max.

	Drop			AUC		
Malware Group	Minimum	Maximum	Average	Minimum	Maximum	Average
adware	0.31%	21.86%	6.29%	75.79%	98.92%	94.25%
flooder	-1.08%	34.27%	7.99%	57.60%	99.62%	94.54%
ransomware	-0.26%	46.15%	9.96%	53.57%	99.83%	94.49%
dropper	-0.41%	25.83%	7.51%	70.60%	99.02%	93.82%
spyware	0.00%	24.50%	6.75%	72.26%	99.08%	94.09%
packed	-0.57%	32.84%	8.31%	64.35%	99.44%	93.99%
crypto miner	-0.12%	40.17%	10.98%	52.05%	99.52%	92.95%
file infector	-0.18%	36.22%	5.14%	62.55%	99.76%	96.60%
installer	0.34%	22.26%	4.56%	74.33%	98.99%	95.72%
worm	-0.47%	25.50%	5.89%	73.63%	99.79%	95.39%
downloader	0.16%	33.31%	8.11%	63.85%	99.22%	93.16%

Figure 6.5: Illustration of Malware Groups (Size-Based) AUC Results (Average, Min, Max)

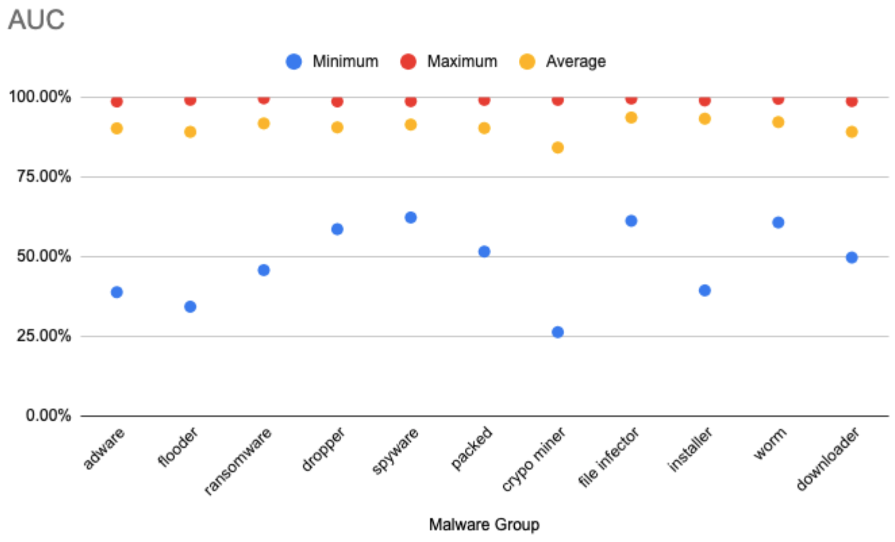
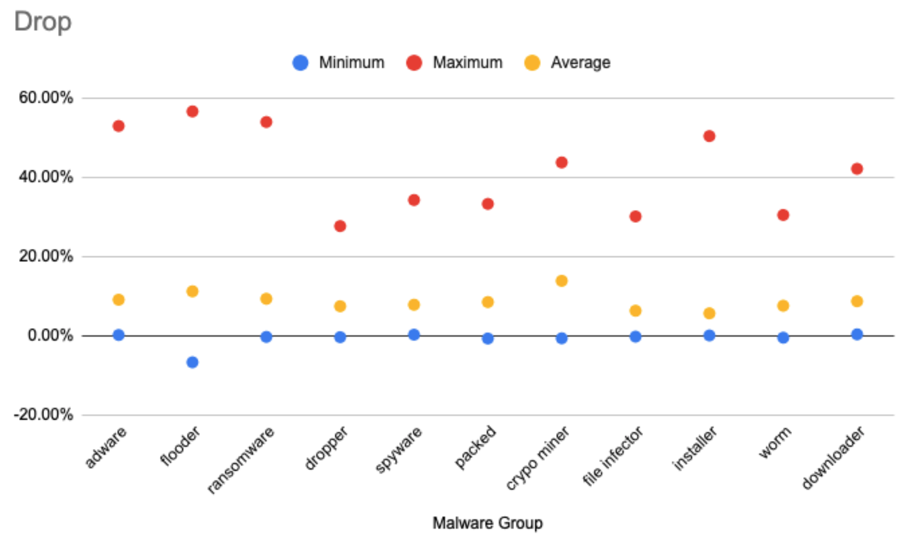


Figure 6.6: Illustration of Malware Groups (Size-Based) Drop Results (Average, Min, Max)



Adware

Adware shows an average drop of 6.29% (before 9.13%) for all training flows. The average performance result of all models for adware for all flows is 94.2% (before 90.2%). The maximum drop is 21.86% (before 53%) when the Aloha model was trained with split 9 and tested with split 1 in the Backward Accumulative with Splits Testing flow. In the same flow with the same split, the Base model gives a drop of 13.5%. The drop seems a result of difficulty by the model to generalise the detection for this type in split 1.

The Forward Accumulative with the Whole Testing flow shows that with just three splits of training (33.3% of the dataset), the drop is less than 1%. However, we need to train from split 1 to 8 to reach less than 1% of drop for the Forward Accumulative with Splits Testing. The drop of all backward flows (whole and splits) did not get less than 3%.

The average performance of all forward accumulative flows with both models is 95.9% (before 89.5%), and the backward accumulative flows is 93.8% (before 93.6%). The average drop of all forward flows is 2.2% (before 4.5%) and for the backward flows is 6.7% (before 7.6%). It indicates that testing in past adware malware performs less than testing in future one in terms of average performance and drop.

Flooder

Flooder shows an average drop of 7% (before 11.25%) for all training flows. The average performance result of all models for flooder for all flows is 94.5% (before 89.1%). The maximum drop is 34.27% (before 56.6%) when the Aloha model trained with split 4 and tested with split 2 in the normal flow. In the same flow with the same split, the base model gives a drop of 10.66%. The drop seems to be a result of outfitting from the Aloha model. In time-based splits, flooder malware was getting the highest and lowest drop due to the size. That effect did not repeat in size-based splits.

The Forward Accumulative with Whole Testing flow shows that with just one split of training from the base model (or two splits results of Aloha model), the drop is less than 1%. However, these results are not consistent with the Forward Accumulative with Splits Testing flow. It required to train from split 1 to 8 to reach a drop of less than 1% but only with the Base model (the Aloha model did not reach that). The drop of all backward flows (whole and splits) did not get less than 1%.

The average performance of all forward accumulative flows with both models is 96.13% (before 87% for time-based splits), and the backward accumulative flows are 95% (before 94%). The average drop of all forward flows is 2.8% (before 8.2%) and for the backward flows is 6.8% (before 5.9%). It means that forward and backwards drop are similar in performance average in flooder malware, but backwards flows drop twice as forward flows.

Ransomware

Ransomware shows an average drop of 9.9% (before 9.3%) for all training flows. The average performance result of all models for ransomware for all flows is 94.4% (before 91.8%). The maximum drop is 46.1% (before 54%) when the Base model is trained with split 7 and tested with split 3 in the normal flow. In the same flow with the same split, the Aloha model gives a drop of 22%. In time-based splits, ransomware shows the second-highest average maximum drop. Moreover, in size-based splits, it is the highest average drop out of all groups.

Forward Accumulative with Whole Testing flow shows that with just three splits of training (33.3% of the dataset), the drop is less than 1% with both models. The same happens with the Forward Accumulative with Splits Testing flow. Not only that, but it also shows some gain (negative drop value) in several stages of training with both models. The drop of all backward flows (whole and splits) did not get less than 1%.

The average performance of all forward accumulative flows with both models is 97.56% (before 91.1%), and the backward accumulative flows is 92.92% (before 95%). The average drop of all forward flows is 1.61% (before 6.3%), and for the backward flows is 11.06% (before 6.6%). That means testing in past ransomware samples shows a similar average drop as testing for future samples. It means testing with past ransomware data performs significantly less than testing for future ones with this way of splitting.

Dropper

Dropper shows an average drop of 7.5% (before 7.5%) for all training flows. The average performance result of all models for the dropper for all flows is 93.8% (before 90.5%). The maximum drop is 25.8% (before 27.7%) when the Aloha model is trained with split 6 and tested with split 2 in the normal flow. In the same flow with the same split, the base model gives a drop of 15.4%.

The Forward Accumulative with Whole Testing flow shows that with just three splits of training (33.3% of the dataset), the drop is 1% or less with both models. The same happens with the Forward Accumulative with Splits Testing flow (except with the Aloha model - need to train with four splits). The drop of all backward flows (whole and splits) did not get less than 1%.

The average performance of all forward accumulative flows with both models is 95.6% (before 89.2%), and the backward accumulative flows is 93.7% (before 94.1%). The average drop of all forward flows is 1.9% (before 5.4%) and for the backward flows is 7.8% (before 5.5%). It means that forward and backwards drop are similar in an average performance in flooder malware, but backwards flows drop far more than forward flows.

Spyware

The average performance result of all models for spyware for all flows is 94.09% (before 91.4%). And the average drop for all flows is 6.75% (before 7.8%). The maximum drop is 24.5% (before 34.3%) when the Aloha model trained with split 8 and tested with split 2 in the Normal flow. In the same flow with the same split, the Base model gives a drop of 13.3%.

The Forward Accumulative with Whole Testing flow shows that with the five splits of training (55.5% of the dataset), the drop is less than 1% with both models. The same happens with the Forward Accumulative with Splits Testing flow. The drop of all backward flows (whole and splits) did not get less than 1%. However, it shows around 1% at the last stage (training with split 1 to 8 and test with split 9).

The average performance of all forward accumulative flows with both models is 94.9% (before 88.9%), and the backward accumulative flows is 94.4% (before 95.5%). It indicates that testing in past spyware malware performs similarly as testing in future one in term of average performance. The average drop of all forward flows is 3.2% (before 8.2%) and for the backward flows is 6.28% (before 4%). That means testing for past spyware samples shows a more considerable drop compared with testing for future samples.

Packed

The average performance result of all models for packed group for all flows is 94% (before 90.3%). And the average drop for all flows is 8.3% (before 8.5%). The maximum drop is 32.8% (before 33.3%) when the Aloha model trained with split 6 and tested with split 2 in the Normal flow. In the same flow with the same split, the Base model gives a drop of 19.6%.

The Forward Accumulative with Whole Testing flow shows that with just two splits of training (22.22% of the dataset), the drop is less than 1% in both models. The same happens with the Forward Accumulative with Splits Testing flow. Not only that, both models perform slightly better in the testing part comparing with training on several occasions. The drop of all backward flows (whole and splits) did not get less than 1%.

The average performance of all forward accumulative flows with both models is 95.9% (before 89.3%), and the backward accumulative flows is 93.3% (before 93.7%). It indicates that testing in past packed malware performs better than testing in future one in term of average performance. The average drop of all forward flows is 1.5% (before 6.1%) and for the backward flows is 9.2% (before 6.2%). That means testing for past packed samples shows samples show a more significant drop compared with testing for future samples.

Crypto Miner

The average performance result of all models for crypto miner for all flows is 92.9% (before 84.2%). And the average drop for all flows is 10.9% (before 13.9%). It is the highest average drop compared with all groups (same as in time-based splits). The maximum drop is 40.1% (before 43.8%) when the Aloha model trained with split 6 and tested with split 1 in the normal flow. In the same flow with the same split, the base model gives a drop of 29%.

The Forward Accumulative with Whole Testing flow shows that with five splits of training (55.5% of the dataset), the drop is 1% or less with the Base model (Aloha model required only four splits). The same happens with the Forward Accumulative with Splits Testing flow. The drop of all backward flows (whole and splits) did not get less than 3%.

The average performance of all forward accumulative flows with both models is 94.7% (before 81.8%), and the backward accumulative flows is 93.1% (before 90.6%). It indicates that testing in past crypto-miner malware performs similar to testing in the future one in terms of average performance. The average drop of all forward flows is 2.7% (before 9%) and for the backward flows is 11.2% (before 10.8%). That means testing for past crypto miner samples shows a more significant drop compared with testing for future samples.

File Infector

The average performance results of all models for file infector for all flows is 96.6% (before 93.6%). It is the highest average performance out of all groups (same as in time-based splits results). And the average drop for all flows is 5.1%, which is the second-best result after installer. The maximum drop of all the models is 36.2% (before 30.1%) when the Aloha model trained with split 8 and tested with split 2 in the normal flow. In the same flow with the same split, the base model gives a drop of 17.5%.

The Forward Accumulative with Whole Testing flow shows that with just three splits of training (33.3% of the dataset), the drop is less than 1% with both models. The same happens with the Forward Accumulative with Splits Testing flow (expect in Aloha required five splits). Moreover, there are several occasions where the testing performs better than training. The drop of all backward flows (whole and splits) gets less than 1% after training with eight splits in both models.

The average performance of all forward accumulative flows with both models is 97.3% (before 92.2%), and the backward accumulative flows is 96.9% (before 97.1%). It indicates that testing in past file infector malware performs is similar to testing in a future one in terms of average performance. The average drop of all forward flows is 1.91% (before 5.6%), and for the backward flows is 4% (before 3%). That means testing in past file infector samples shows a better average drop than testing for future samples.

Installer

The average performance result of all models for the installer for all flows is 95.7% (before 93.3%). It is the second-highest average performance after the file infector group (same as in time-based splits). And the average drop for all flows is 4.56% (before 5.7%), which is the best (lowest) average result out of all groups. The maximum drop of all the models is 22.2% (before 50.4%) when the Aloha model trained with split 8 and tested with split 2 in the Normal flow. In the same flow with the same split, the Base model gives a drop of 13.3%.

The Forward Accumulative with Whole Testing flow shows that with just three splits of training (33.3% of the dataset), the drop is less than 1% with both models. However, it required training with four splits with the Forward Accumulative with Splits Testing flow to reach to 1% drop or less. The drop of all backward flows (whole and splits) gets around 1% after training with eight splits in both models.

The average performance of all forward accumulative flows with both models is 97% (before 92.5%), and the backward accumulative flows is 95% (before 96%). This indicates that testing in past installer malware performs similarly to testing in future one in terms of average performance. The average drop of all forward flows is 1.6% (before 4.2%) and for the backward flows is 4.6% (before 3.6%). That means testing in past installer samples shows a more considerable average drop compared to testing for future samples.

Worm

The average performance result of all models for the worm group for all flows is 95.4% (before 92.2%). And the average drop for all flows is 5.9% (before 7.6%). The maximum drop of all the models is 25.5% (before 30.5%) when the Aloha model trained with split 8 and tested with split 2 in the normal flow. In the same flow with the same split, the Base model gives a drop of 13.3%.

The Forward Accumulative with Whole Testing flow shows that with just three splits of training (33.3% of the dataset), the drop is less than 1% in both models. The same happens with the Forward Accumulative with Splits Testing flow. The drop of all backward flows (whole and splits) did not get less than 2%.

The average performance of all forward accumulative flows with both models is 96.7% (before 90.9%), and the backward accumulative flows is 94.9% (before 95.3%). It indicates that testing in past worm malware performs similar to testing in future one in terms of average performance. The average drop of all forward flows is 1.49% (before 5.8%) and for the backward flows is 6.82% (before 5.1%). That means testing in past worm samples shows a more significant average drop than testing for future samples.

Downloader

The average performance result of all models for the downloader group for all flows is 93.1% (before 89.1%). And the average drop for all flows is 8.1% (before 8.7%). The maximum drop of all the models is 33.3% (before 42.1%) when the Aloha model is trained with split 6 and tested with split 2 in the Normal flow. In the same flow with the same split, the Base model gives a drop of 16.5%.

The Forward Accumulative with Whole Testing flow shows that with just three splits of training (33.3% of the dataset), the drop is 1% or less in both models. Nevertheless, it required training with six splits with the Forward Accumulative with Splits Testing flow to reach to 1% drop or less. The drop of all backward flows (whole and splits) did not get less than 3%.

The average performance of all forward accumulative flows with both models is 94.9% (before 87.6%), and the backward accumulative flows is 92.6% (before 93.6%). It indicates that testing in past downloader malware performs less than testing in future one in terms of average performance. The average drop of all forward flows is 2.8% (before 6.5%) and for the backward flows is 8.6% (before 5.7%). That means testing in past downloader samples shows a more considerable average drop than testing for future samples.

6.5 Summary

We can summarise this chapter in the following points:

- We split the dataset based on the size of samples instead of splitting based on time duration like the last chapter. It might be useful when we want to retrain our model based on the number of samples instead of how much time has passed. We can also check the observations of the last chapter if they are consistent with this different way of splitting.
- We follow the same methodology of the last chapter with the same training and testing flows.

- In the normal flow, the Base model shows better results than the Aloha model and drops less. Both models drop less when trained with the first half of the splits compared with the last half.
- In the Forward Accumulative with Whole Testing flow, just training with the split from 1 to 3 (33% of the dataset) gives very good results with a drop less than 1%, when tested with the reset of the splits combined (67% of the dataset).
- Forward Accumulative with Splits Testing flow gives more evidence about the quality of the data in split 1 to 3 combined.
- In the Backward Accumulative with Whole Testing flow, the models in all stages fail to reach a drop of less than 2%. There is valuable information in split 1 that causes this result. Moreover, the models cannot generalise well to learn it from other splits combined.
- The Backward Accumulative with Whole Testing flow shows a bigger drop from Backward Accumulative with Whole Testing flow. And the highest drop happens when testing with split 2.
- The average results of each group of malware analysis shows in general that the installer and file infector groups seem to be the easiest to detect, while the crypto miner is the hardiest. It is consistent with the size-based splits results.

In the next chapter, we will try to create another model to minimise the drop while increasing the performance. We will test with both time-based splits and size-based splits and with all training and testing flows. Then, we will compare the results we got Base and Aloha models.

Chapter 7

Propose a Better ML Model for Malware Detection (Model-33)

7.1 Chapter Overview

This chapter attempts to develop a better model that can perform well based on different ways of splitting the data proposed in the last two chapters. The goals of the model are to increase the performance (AUC) and minimise the drop.

7.2 Methodology to Find A Model

Based on the results of the last two chapters, the drop in model performance is high. We want to have a better model with better performance and less drop than the Base and Aloha models. The results should be based on the proposed splitting from the last two chapters.

We have two different ways to split the data, and each way to split the data has five different flows of training and testing. But to look for a better model, it will be time-consuming to test each model with ten different flows (including both ways of splitting). For that, we will first focus on the normal flow in a time-based split. Next, we will calculate the average performance (AUC), minimum performance number (the lowest point shows the maximum drop), and maximum performance. We will then calculate the average of those three numbers (average, minimum, and maximum of AUC). The target is to find a model that will get a better result than the Base model and Aloha model. We will iterate with the same process with different deep learning architecture designs. If the model gets higher results than the Base and Aloha Models, then we will test the model with the normal flow in a size-based split and do the same calculation. The model that we will select is the one that has a better average in both averages of time-based and size-based splits. Once we selected the model, we will run the model with the rest of the training flows in both time and size-based splits. And we will compare the results with the Base model and Aloha model.

To summarise, the steps are as follows:

1. Calculate the average, minimum, maximum performance values (AUC) of the Base model and the Aloha model for the normal flows of both time-based split and size-based splits. And calculate the average of those three values.
2. Design deep learning model and test it with the normal flow of time-based Split.
3. Find average, minimum, maximum performance values (AUC) of the result. Then calculate the average of those three values.
4. Compare the results with the base model and Aloha model or a model that we designed. If the results are far better than those results, repeat the experiment in the normal flow of size-based split. We do not need to calculate the results of the Normal flow in size-based split often. The result of the normal flow in a time-based split is sufficient to understand the model's performance.
5. Repeat the step 2 to 4 until we are satisfied with the results.
6. Select the model with a better average of the three values (average, minimum, and maximum performance) in both ways of splitting (time-based and size-based).
7. Train and test the selected model in all flows of training and testing in both time-based and size-based.
8. Compare and analyse the result with the results of the Base model and Aloha model.

The main deep learning architecture that we will use most is the convolutional neural network (ConvNet). In particular, we will use a one-dimensional convolutional neural network. ConvNet has shown promising results in the first part of the research (in network intrusion detection system, NIDS). For this reason, we will iterate over the different designs of the convolutional neural network. There will be a change in the number of layers and the hyperparameter of each design. We will also use different activation functions and dropout regulator.

7.3 Experiment and the results of finding a model

Table 7.1 shows the results of the experiments with different model designs. The first two row shows the results of the Base model and the Aloha model. The Base model averages 89.49% in the normal flow of time-based split and 93.96% in the size-based split. The Aloha model performs less than the Base model. The Aloha model gets an average result of 84.5% in the normal flow of the time-based and an average of 90.76% in the size-based one. Both models have a big drop. The based model lowest performance value is 30.01% in the time-based split and 53.57% in the size-based split. The Aloha model lowest performance result in the time-based is 24.63% and 56.06%. The ways of splitting the data make a challenging task for the model to perform. Such a big drop in both models makes it not the ideal model to deploy in a real-world scenario. The base model gets an overall average of 77.77% (includes average, minimum, maximum performance values of both ways of splitting), and the Aloha model gets 75.88%. We argue that a model with a higher overall average is a more suitable model to work in a real-world scenario (based on the performance metric, not on the model's size).

We experiment with 37 different models. The iterations are based on different designs and hyperparameter tuning. The changes include the number of layers, the size of the input and output layer, the percentage of the dropout, the activation functions, the number of filters in the convolutional layer, and the type of layers. Each model is named based on the iteration number. The best overall models (based on the average of both ways of splitting) are Model-31, Model-33, and Model-35. The overall averages are (in order) 86.10%, 86.49% and 86.19%. Model-31 get the highest average performance results in time-based split out of all models (92.07%). Model-35 get the lowest drop out of all models with a minimum performance value of 69.27%. However, the overall best average of the six values is Model-33. For that, we will choose Model-33 as our select model for further analysis.

Model-33 is a far better model than the Base model and the Aloha models in most metrics. The only metric that is better than Model-33 is the average results of the size-based performance. The Base model gets 93.96%, and the Model-33 get 91.96%. Nevertheless, all other metric is higher in the Model-33. The average time-based split performance for Model-33 is 92.00% (Base: 89.49%, Aloha: 84.5%). The lowest performance value in the time-based split of Model-33 is 68.76% (Base: 30.01%, Aloha: 24.63%). The lowest performance value in the size-based split of Model-33 is 66.83% (Base: 53.57%, Aloha: 56.06%). The average of the three values (average, minimum, maximum performance values) in the time-based of Model-33 is 86.86% (Base: 73.11%, Aloha: 69.64%). And for the size-based, the Model-33 get 86.11% (Base: 82.43%, Aloha: 82.13%). The average result of the six values (average, minimum, and maximum performance values of both ways of splitting) of the Model-33 is 86.49% (Base: 77.77%, Aloha: 75.88%). We can notice that the biggest improvement is in the drop (the lowest performance value). The results here show that Model-33 is a better model than the Base model and the Aloha model.

Table 7.1: Results of different iteration of a better model design for malware detection.

	Time-based Split (Normal Flow)			Size-based Split (Normal Flow)			Averge		
ML Models	Avergae	Max	Min	Avergae	Max	Min	time-based	size-based	Both
Base model	89.49%	99.83%	30.01%	93.96%	99.75%	53.57%	73.11%	82.43%	77.77%
Aloha Model	84.50%	99.77%	24.63%	90.76%	99.57%	56.06%	69.64%	82.13%	75.88%
Model-1	88.89%	99.66%	58.23%	88.92%	99.67%	61.24%	82.26%	83.27%	82.77%
Model-2	83.70%	99.15%	52.77%	83.14%	99.14%	49.66%	78.54%	77.31%	77.92%
Model-3	61.90%	93.64%	41.51%	60.86%	95.21%	37.42%	65.68%	64.50%	65.09%
Model-4	89.90%	99.75%	50.49%				80.05%		80.05%
Model-5	88.03%	99.60%	57.21%				81.61%		81.61%
Model-6	89.43%	99.71%	57.91%				82.35%		82.35%
Model-7	87.41%	99.54%	57.43%				81.46%		81.46%
Model-8	89.37%	99.75%	54.27%				81.13%		81.13%
Model-9	87.52%	99.62%	51.66%				79.60%		79.60%
Model-10	89.69%	99.77%	53.18%				80.88%		80.88%
Model-11	88.19%	99.70%	56.88%				81.59%		81.59%
Model-12	87.61%	99.62%	55.45%				80.89%		80.89%
Model-13	89.81%	99.74%	60.98%				83.51%		83.51%
Model-14	90.32%	99.77%	64.11%				84.73%		84.73%
Model-15	91.03%	99.83%	63.31%				84.72%		84.72%
Model-16	91.61%	99.86%	51.46%				80.98%		80.98%
Model-17	91.63%	99.81%	67.86%	91.51%	99.83%	64.15%	86.43%	85.16%	85.80%
Model-18	91.78%	99.84%	64.60%				85.41%		85.41%
Model-19	89.70%	99.73%	62.14%				83.86%		83.86%
Model-20	91.85%	99.85%	61.58%				84.43%		84.43%
Model-21	91.70%	99.84%	63.02%				84.86%		84.86%
Model-22	92.00%	99.84%	61.66%				84.50%		84.50%
Model-23	90.81%	99.79%	63.30%				84.63%		84.63%
Model-24	91.89%	99.83%	63.63%				85.12%		85.12%
Model-25	88.29%	99.83%	52.83%				80.31%		80.31%
Model-26	80.41%	99.82%	42.70%				74.31%		74.31%
Model-27	91.62%	99.84%	59.27%				83.57%		83.57%
Model-28	91.89%	99.85%	66.89%				86.21%		86.21%
Model-29	91.86%	99.84%	65.40%				85.70%		85.70%
Model-30	89.17%	99.78%	55.28%				81.41%		81.41%
Model-31	92.07%	99.84%	67.97%	91.88%	99.81%	65.04%	86.62%	85.58%	86.10%
Model-32	66.63%	98.50%	32.39%				65.84%		65.84%
Model-33	92.00%	99.82%	68.76%	91.69%	99.81%	66.83%	86.86%	86.11%	86.49%
Model-34	90.39%	99.69%	66.92%				85.67%		85.67%
Model-35	91.48%	99.77%	69.27%	91.55%	99.76%	65.30%	86.84%	85.54%	86.19%
Model-36	81.86%	99.77%	49.65%				77.09%		77.09%
Model-37	88.01%	99.66%	55.90%	87.96%	99.65%	52.13%	81.19%	79.91%	80.55%

7.4 Model-33 Design

Model-33 is based on the one-dimensional convolutional neural network. Therefore, it has several stacks of layers. Similar to the Aloha model, we will use the same auxiliary Loss and multiple output layers. The model difference is on the size of the malware tags output layer. The design details are as following (in order):

1. 1D ConvNet: Input = 1, Output = 256, Kernel Size = 3, Stride = 2, Padding = 0.
2. Activation Function: Rectified Linear Unit (ReLU).
3. Max Pool 1D: Kernel Size = 3, Stride = 2.
4. Dropout: 50%.
5. 1D ConvNet: Input = 256, Output = 16, Kernel Size = 3, Stride = 2, Padding = 0.
6. Activation Function: Rectified Linear Unit (ReLU).
7. Max Pool 1D: Kernel Size = 3, Stride = 2.
8. Dropout: 50%.
9. Fully Connected Layer: Input = 2352, Output= 512.
10. Activation Function: Rectified Linear Unit (ReLU).
11. Dropout: 50%.
12. Linear layer: Input: 512, Output: 512.
13. Activation Function: Exponential Linear Unit (ELU).
14. Linear layer: Input: 512, Output: 256.
15. Activation Function: Exponential Linear Unit (ELU).
16. Output layers:
    (a) Binary Classification (Malware/Benign):
        i. Linear layer: Input: 256, Output: 1.

ii. Activation Function: Sigmoid functions.

(b) Vendor Count:

i. Linear layer: Input: 256, Output: 1.

(c) Multi Malware Tags:

i. Linear layer: Input: 256, Output: 512.

ii. Activation Function: Exponential Linear Unit (ELU).

iii. Linear layer: Input: 512, Output: 256.

iv. Activation Function: Exponential Linear Unit (ELU).

v. Linear layer: Input: 256, Output: 11.

vi. Activation Function: Sigmoid functions.

The model uses different losses functions for each output. Moreover, the combination of those losses functions is used to update the weights of the network. The full details of the process can be found in the Aloha model paper [72]. We use Adam optimiser for the training [41].

7.5 Results and analysis

In this section, we will discuss the results of Model-33 of time-based splits and size-based splits. We will also compare it with the Base and Aloha models. The full results can be found in the project repository 1.8.

7.5.1 Normal Flow

Table 7.2: Time-Based Splits - Normal Flow - Model-33 Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
Split 1	0.966	0.923	0.902	0.835	0.814	0.839	0.867	0.882	0.890	15.20%
Split 2	0.948	0.972	0.949	0.878	0.849	0.873	0.888	0.909	0.912	12.35%
Split 3	0.942	0.948	0.975	0.903	0.882	0.890	0.906	0.908	0.910	9.34%
Split 4	0.915	0.927	0.960	0.973	0.920	0.899	0.893	0.894	0.898	7.99%
Split 5	0.908	0.919	0.946	0.944	0.978	0.924	0.912	0.920	0.917	6.98%
Split 6	0.914	0.920	0.936	0.923	0.941	0.979	0.961	0.943	0.942	6.59%
Split 7	0.924	0.924	0.940	0.907	0.932	0.962	0.988	0.978	0.968	8.12%
Split 8	0.911	0.893	0.905	0.862	0.889	0.919	0.966	0.991	0.984	12.97%
Split 9	0.895	0.877	0.886	0.841	0.854	0.911	0.957	0.986	0.992	15.10%

Table 7.3: size-based Splits - Normal Flow - Model-33 Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
Split 1	0.966	0.921	0.901	0.837	0.810	0.830	0.862	0.882	0.891	15.68%
Split 2	0.945	0.970	0.940	0.875	0.849	0.866	0.877	0.905	0.905	12.08%
Split 3	0.943	0.950	0.975	0.905	0.892	0.892	0.913	0.912	0.910	8.37%
Split 4	0.913	0.925	0.959	0.973	0.920	0.901	0.888	0.895	0.899	8.44%
Split 5	0.915	0.921	0.949	0.947	0.978	0.924	0.917	0.927	0.923	6.26%
Split 6	0.919	0.921	0.945	0.925	0.951	0.982	0.961	0.943	0.942	6.33%
Split 7	0.922	0.911	0.931	0.901	0.921	0.956	0.987	0.975	0.963	8.60%
Split 8	0.900	0.865	0.882	0.830	0.861	0.901	0.960	0.988	0.980	15.74%
Split 9	0.893	0.867	0.875	0.833	0.847	0.906	0.956	0.985	0.991	15.74%

Table 7.2 shows the average results in all groups of malware of Model-33 with the normal flow in time-based splits. Figure 7.5.1 visualises the average results of each split. Table 7.3 shows the average results in all groups of malware of Model-33 with the normal flow in size-based splits and Figure 7.3 for the visualisation. The black cell numbers in the tables indicate that the split train and tested with itself.

For the time-based splits, Model-33 get better average performance and drop results compared with the Base model and Aloha model. The model-33 average performance result is 91.99%, and the average drop is 10.52%. For comparison, the base model gets average performance results of 90.03% and an average drop results of 12.26%. Aloha model gets average performance results of 82.62% and an average drop results of 14.76%.

For the size-based splits, the base model gets better results in the average performance result, but Model-33 gets better results in an average drop. Model-33 gets average performance results of 91.69% and an average drop results of 10.8%. The base model gets an average performance result of 93.96% and an average drop result of 12.06%. Aloha model gets average performance results of 90.74% and an average drop results of 17.58%.

Figure 7.1: Time-Based Splits - Normal Flow - Model-33

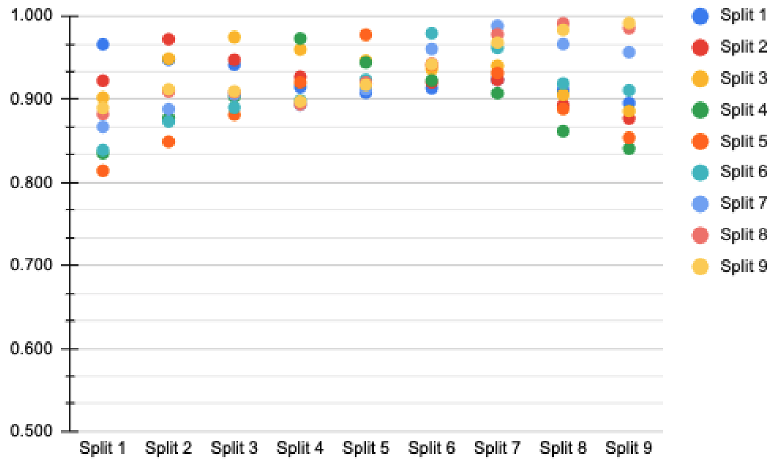
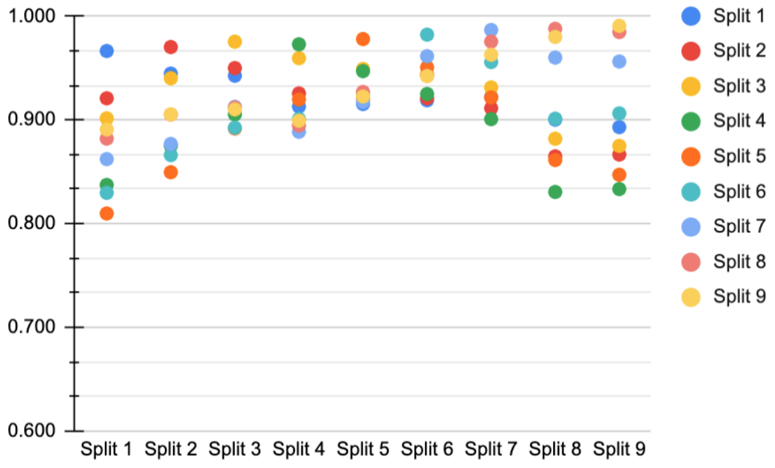


Figure 7.2: Size-Based Splits - Normal Flow - Model-33



7.5.2 Forward Accumulative with Whole Testing

Table 7.4 shows the average results in all groups of malware of Model-33 with forward accumulative with the whole testing in time-based splits. Table 7.5 shows the average results in size-based splits. For the time-based splits, all models get similar results. For example, the model-33 average performance result is 93.5%, and the average drop is 3.98%. For comparison, the base model gets average performance results of 92.75% and an average drop results of 3.37%. Aloha model gets average performance results of 90.76% and an average drop result of 3.87%.

For the size-based splits, all models also get similar results. For example, the model-33 average performance result is 95.91%, and the average drop is 1.8%. For comparison, the base model gets average performance results of 97.19% and an average drop of 1.2%. Aloha model gets average performance results of 96.4% and an average drop result of 1.34%.

Table 7.4: Time-Based Splits - Forward Accumulative with Whole Testing Flow - Model-33 Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	Drop
1	96.84%								88.29%	8.55%
2		97.14%							90.47%	6.67%
3			97.49%						91.88%	5.61%
4				97.17%					92.70%	4.47%
5					97.50%				93.64%	3.86%
6						97.73%			95.82%	1.91%
7							97.78%		97.06%	0.73%
8								98.13%	98.12%	0.00%

Table 7.5: Size-Based Splits - Forward Accumulative with Whole Testing Flow - Model-33 Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	Drop
1	97.07%								90.96%	6.11%
2		97.24%							93.04%	4.21%
3			97.55%						95.66%	1.89%
4				97.81%					96.69%	1.12%
5					97.84%				97.25%	0.59%
6						98.08%			97.85%	0.23%
7							97.73%		97.81%	-0.08%
8								98.31%	98.02%	0.29%

7.5.3 Forward Accumulative with Splits Testing

Table 7.6 shows the average results in all groups of malware of Model-33 with forward accumulative with splits testing in time-based splits. Table 7.7 shows the average results in size-based splits.

For the time-based splits, Model-33 get better performance results. However, the drop results of all models are close. The model-33 average performance result is 91.09%, and the average drop is 6.46%. For comparison, the base model gets average performance results of 88.09% and an average drop results of 7.27%. Aloha model gets average performance results of 85.13% and an average drop result of 8.53%.

For the size-based splits, all models get similar results. For example, the model-33 average performance result is 94.48%, and the average drop is 2.63%. For comparison, the base model gets average performance results of 96.06% and an average drop of 1.9%. Aloha model gets average performance results of 94.73% and an average drop result of 3.05%.

Table 7.6: Time-Based Splits - Forward Accumulative with Splits Testing Flow - Model-33 Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	Drop
1	96.84%	92.08%	91.20%	83.77%	81.94%	84.06%	87.11%	88.73%	88.64%	14.90%
2		97.14%	94.58%	87.92%	84.59%	87.26%	88.76%	90.97%	91.10%	12.55%
3			97.49%	90.49%	87.75%	89.23%	90.74%	92.57%	92.88%	9.75%
4				97.17%	91.54%	90.73%	91.63%	93.10%	93.26%	6.43%
5					97.50%	93.09%	92.53%	93.92%	93.62%	4.97%
6						97.73%	95.80%	95.70%	95.74%	2.03%
7							97.78%	97.28%	96.70%	1.09%
8								98.13%	98.13%	0.00%

Table 7.7: Size-Based Splits - Forward Accumulative with Splits Testing Flow - Model-33 Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	Drop
1	97.07%	87.38%	87.31%	88.94%	91.11%	91.10%	89.83%	90.21%	92.25%	9.77%
2		97.24%	92.12%	91.77%	92.84%	93.12%	92.70%	92.80%	94.15%	5.47%
3			97.55%	96.00%	95.62%	95.66%	95.68%	95.50%	96.01%	2.05%
4				97.81%	97.36%	96.93%	96.69%	96.28%	96.46%	1.53%
5					97.84%	97.56%	97.23%	96.87%	96.82%	1.02%
6						98.08%	98.31%	97.74%	97.31%	0.77%
7							97.73%	97.99%	97.58%	0.15%
8								98.31%	97.99%	0.32%

7.5.4 Backward Accumulative with Whole Testing

Table 7.9 shows the average results in all groups of malware of Model-33 with backward accumulative with the whole testing in time-based splits. Table 7.9 shows the average results in size-based splits.

For the time-based splits, all models get similar results. For example, the model-33 average performance result is 94.12%, and the average drop is 4.61%. For comparison, the base model gets average performance results of 95.82% and an average drop result of 3.51%. Aloha model gets average performance result of 95.51% and an average drop result of 3.76%.

For the size-based splits, all models also get similar results. For example, the model-33 average performance result is 93.53%, and the average drop is 4.88%. For comparison, the base model gets average performance results of 95.05% and an average drop results of 4.31%. Aloha model gets average performance results of 94.5% and an average drop result of 4.74%.

Table 7.8: Time-Based Splits - Backward Accumulative with Whole Testing Flow - Model-33 Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
1								94.22%	99.18%	4.96%
2							93.16%	■	99.15%	5.99%
3						93.68%	■	■	98.97%	5.30%
4					94.12%	■	■	■	98.73%	4.61%
5				93.90%	■	■	■	■	98.65%	4.75%
6			94.98%	■	■	■	■	■	98.57%	3.59%
7		93.94%	■	■	■	■	■	■	98.26%	4.32%
8	94.94%	■	■	■	■	■	■	■	98.31%	3.37%

Table 7.9: Size-Based Splits - Backward Accumulative with Whole Testing Flow - Model-33 Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
1								93.95%	98.76%	4.82%
2							94.15%		99.16%	5.00%
3						93.54%			99.19%	5.65%
4					92.65%				99.04%	6.40%
5				92.59%					99.02%	6.43%
6			93.62%						98.89%	5.27%
7		94.13%							95.09%	0.96%
8	93.60%								98.07%	4.48%

7.5.5 Backward Accumulative with Splits Testing

Table 7.11 shows the average results in all groups of malware of Model-33 with backward accumulative with splits testing in time-based splits. Table 7.11 shows the average results in size-based splits.

For the time-based splits, all models get similar results. For example, the model-33 average performance result is 92.12%, and the average drop is 7.81%. For comparison, the base model gets average performance results of 93.72% and an average drop result of 6.57%. Aloha model gets average performance result of 93.03% and an average drop result of 7.04%.

For the size-based splits, all models also get similar results. For example, the model-33 average performance result is 92.68%, and the average drop is 10.33%. For comparison, the base model gets average performance results of 94.05% and an average drop result of 9.29%. Aloha model gets average performance results of 93.14% and an average drop result of 10.11%.

Table 7.10: Time-Based Splits - Backward Accumulative with Splits Testing Flow - Model-33 Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
1	90.24%	88.00%	89.35%	85.18%	86.05%	91.27%	95.89%	98.22%	99.18%	14.00%
2	91.13%	89.38%	90.62%	86.28%	89.46%	92.05%	96.64%		99.15%	12.87%
3	92.18%	91.26%	92.72%	89.69%	92.08%	95.26%			98.97%	9.28%
4	92.88%	91.99%	93.91%	91.29%	94.43%				98.73%	7.44%
5	93.30%	92.49%	94.54%	93.23%					98.65%	6.16%
6	93.78%	93.63%	95.53%						98.57%	4.94%
7	93.79%	93.67%							98.26%	4.59%
8	95.09%								98.31%	3.22%

Table 7.11: Size-Based Splits - Backward Accumulative with Splits Testing Flow - Model-33 Model - Average Results.

Average	test									
	Split 1	Split 2	Split 3	Split 4	Split 5	Split 6	Split 7	Split 8	Split 9	drop
1	87.22%	84.18%	90.28%	94.37%	96.62%	97.44%	98.21%	98.57%	98.76%	14.59%
2	88.41%	85.52%	91.03%	94.98%	97.45%	98.26%	98.96%		99.16%	13.63%
3	89.05%	85.85%	91.21%	95.39%	97.74%	98.51%			99.19%	13.34%
4	88.85%	86.38%	91.24%	95.48%	98.02%				99.04%	12.66%
5	87.39%	83.79%	89.67%	96.75%					99.02%	15.23%
6	92.05%	91.81%	95.26%						98.89%	7.08%
7	93.49%	93.56%							95.09%	1.60%
8	93.58%								98.07%	4.50%

7.6 Malware Groups

In this section, we will compare the average performance and drop results between Model-33, the base model, and Aloha model in each malware group. If the results difference is within 3%, we will consider it as similar results. In general, most of the results will be similar between the three models.

7.6.1 Adware

For time-based splits, all models getting similar average results in testing for the adware group. For example, model-33 get an average performance result of 91.01% and an average drop of 9.62%. The base model got an average performance result of 91.46% and an average drop of 8.06%. Aloha model average performance result is 89.08% and a drop of 10.20%.

For size-based splits, Model-33 performed worse and drops more than the other models. Model-33 get an average performance result of 91.39% and an average drop of 8.96%. The base model got an average performance result of 94.84% and an average drop of 5.84%. Aloha model average performance result is 93.66% and a drop of 6.74%.

7.6.2 Flooder

For time-based splits, Model-33 getting better results in the average performance and the average drop. Model-33 get an average performance result of 94.73% and an average drop of 9.33%. The base model got an average performance result of 90.87% and an average drop of 8.06%. Aloha model average performance result is 87.47% and a drop of 13.17%.

The average performance results are similar for all models for size-based splits, but the drop results are different. For the drop, Model-33 and the base models have similar average drops and are better than the result of the Aloha model. Model-33 get an average performance result of 95.71% and an average drop of 5.53%. The base model got an average performance result of 95.72% and an average drop of 6.51%. Aloha model average performance result is 93.35% and a drop of 9.47%.

7.6.3 Ransomware

For time-based splits, Model-33 and the base model average performance and drop results are close to each other and better than the Aloha model results for time-based splits. Model-33 get an average performance result of 95.27% and an average drop of 6.3%. The base model got an average performance result of 93.54% and an average drop of 7.91%. The Aloha model average performance result is 90.08% and a drop of 10.83%.

For size-based splits, all models got similar average performance, but Model-33 have a better average drop. Model-33 get an average performance result of 95.55% and an average drop of 6.82%. The base model got an average performance result of 94.96% and an average drop of 9.46%. Aloha model average performance result is 94.01% and a drop of 10.46%.

7.6.4 Dropper

For time-based splits, all models get similar average results in testing for the dropper group. For example, model-33 get an average performance result of 91.12% and an average drop of 9.23%. The base model got an average performance result of 91.76% and an average drop of 7.06%. Aloha model average performance result is 89.40% and a drop of 7.93%.

For size-based splits, all models got similar average performance. Model-33 get an average performance result of 91.88% and an average drop of 9.00%. The base model got an average performance result of 94.54% and an average drop of 7.10%. Aloha model average performance result is 93.09% and a drop of 7.92%.

7.6.5 Spyware

For time-based splits, all models get similar average results in testing for the spyware group. Model-33 got an average performance result of 92.16% and an average drop of 7.94%. The base model got an average performance result of 92.39% and an average drop of 7.48%. Aloha model average performance result is 90.50% and a drop of 8.24%.

For size-based splits, all models got similar average performance. Model-33 get an average performance result of 92.65% and an average drop of 7.51%. The base model got an average performance result of 94.57% and an average drop of 6.46%. Aloha model average performance result is 93.60% and a drop of 7.05%.

7.6.6 Packed

For time-based splits, all models get similar average results in testing for the packed group. Model-33 get an average performance result of 91.40% and an average drop of 7.80%. The base model got an average performance result of 91.51% and an average drop of 7.88%. Aloha model average performance result is 89.21% and a drop of 9.21%.

For size-based splits, all models got similar average performance. Model-33 get an average performance result of 92.84% and an average drop of 7.33%. The base model got an average performance result of 94.63% and an average drop of 7.40%. Aloha model average performance result is 93.34% and a drop of 9.22%.

7.6.7 Crypto Miner

For time-based splits, all models get similar average results in testing for the crypto miner group. Model-33 get an average performance result of 87.42% and an average drop of 11.86%. The base model got an average performance result of 85.63% and an average drop of 13.09%. Aloha model average performance result is 82.84% and a drop of 14.72%.

For size-based splits, all models got a similar average performance. Model-33 get an average performance result of 91.59% and an average drop of 10.25%. The base model got an average performance result of 93.97% and an average drop of 9.81%. Aloha model average performance result is 91.93% and a drop of 12.16%.

7.6.8 File Infector

For time-based splits, all models get similar average results in testing for the file infector group, but the Aloha model drops more. Model-33 get an average performance result of 95.68% and an average drop of 4.71%. The base model got an average performance result of 95.00% and an average drop of 5.47%. Aloha model average performance result is 92.34% and a drop of 7.25%.

For size-based splits, all models got similar average performance. Model-33 get an average performance result of 96.45% and an average drop of 3.91%. The base model got an average performance result of 97.43% and an average drop of 3.74%. Aloha model average performance result is 95.77% and a drop of 6.53%.

7.6.9 Installer

For time-based splits, all models get similar average results in testing for the installer group. Model-33 get an average performance result of 94.19% and an average drop of 5.22%. The base model got an average performance result of 94.49% and an average drop of 4.65%. Aloha model average performance result is 92.13% and a drop of 6.75%.

For size-based splits, all models got similar average performance. Model-33 get an average performance result of 94.5% and an average drop of 5.24%. The base model got an average performance result of 96.15% and an average drop of 4.17%. Aloha model average performance result is 95.29% and a drop of 4.96%.

7.6.10 Worm

All models get similar average results in testing for the worm group for time-based splits, but the Aloha model average performance result is less. Model-33 get an average performance result of 93.90% and an average drop of 6.52%. The base model got an average performance result of 93.55% and an average drop of 6.66%. Aloha model average performance result is 90.94% and a drop of 8.57%.

For size-based splits, all models got similar average performance. Model-33 get an average performance result of 94.44% and an average drop of 6.03%. The base model got an average performance result of 95.89% and an average drop of 4.98%. Aloha model average performance result is 94.89% and a drop of 6.79%.

7.6.11 Downloader

For time-based splits, all models get similar average results in testing for the downloader group. Model-33 get an average performance result of 90.41% and an average drop of 8.54%. The base model got an average performance result of 90.98% and an average drop of 7.63%. Aloha model average performance result is 87.38% and a drop of 9.87%.

For size-based splits, all models got similar average performance. Model-33 get an average performance result of 91.54% and an average drop of 8.27%. The base model got an average performance result of 93.97% and an average drop of 7.03%. Aloha model average performance result is 92.36% and a drop of 9.19%.

7.7 Time-Based vs Size-Based

After we have the results of all three models in time-based splits and size-based splits, we can report the following observations:

- The average performance of all the results from all models in time-based splits is 92.65%, and the average drop is 6.89%. On the other hand, the average performance for size-based splits is 95.27%, and the average drop is 6.34%. Thus, in general, the size-based results are slightly better in performance, but both drop at the same average result.

- For the normal flow, the average performance result of all models for time-based splits is 88.21%, and the average drop is 12.51%. On the other hand, the average performance result is 92.13% for the size-based splits, and the average drop is 13.48%. Thus, size-based splits perform better than time-based, but it drops slightly more. The drop increases because the last splits (split 7 to 9) contain less information about the earlier splits attacks.

- For the forward accumulative with the whole testing flow, the average performance result of all models for time-based splits is 94.20%, and the average drop is 3.74%. On the other hand, the average performance result is 97.22% for the size-based splits, and the average drop is 1.45%. Thus, the size-based results are better in average performance and drop.
- For the forward accumulative with splits testing flow, the average performance result of all models for time-based splits is 89.64%, and the average drop is 7.09%. The average performance result is 95.98% for the size-based splits, and the average drop is 2.21%. The size-based results again are better in average performance and drop.
- For the backward accumulative with the whole testing flow, the average performance result of all models for time-based splits is 97.13%, and the average drop is 3.96%. For the size-based splits, the average performance result is 96.68%, and the average drop is 4.64%—both methods of splitting show similar results.
- For the backward accumulative with splits testing flow, the average performance result of all models for time-based splits is 94.08%, and the average drop is 7.14%. On the other hand, the average performance result is 94.33% for the size-based splits, and the average drop is 9.91%. Thus, both splitting methods show similar performance results, but size-based splits drop more than time-based splits.

7.8 Summary

We can summarise this chapter by the following points:

- This chapter aims to find a new machine learning with better performance and less drop than the base and Aloha models.
- We describe the process to find a model. Moreover, after testing 37 different machine learning models, we selected a model and called it Model-33. The model is based one-dimensional convolutional neural network.

- We tested the model in both ways of splitting the data (time-based and size-based) and with the five different flows of training and testing that we discussed in the last two chapters.
- The main big advantage of the model is having far less drop than the base and Aloha models.
- The average results from all models shows that the size-based split shows slightly better result, but they drop at the same average result.

In the next chapter, we will look at a different way to evaluate if the data can be generated. We will take the RSA algorithm as an example.

Chapter 8

Generator Evaluation Method (RSA Security as a Case Study)

8.1 Chapter Overview

In this chapter, we will look at how to evaluate a problem that has unbounded data to test. Such data can be found in different mathematical problems, such as prime number factorisation. We propose the use of a programmatic generator with any deep learning model. The data generation will be executed in the CPU and the training in the GPU. We will take RSA (a public key algorithm) as a case study for the proposed workflow.

8.2 Introduction

The source of the data used in training deep learning models can vary from application to application. In some cases, the data can be generated on-demand algorithmically by a generation process. For example, we can examine mathematical data or data obtained from the simulation of a particular scenario. We can approach the training phase with such data with different strategies. In one approach, the data is generated and stored before it is used in subsequent training. The downside of such an approach is the memory and storage consumption, leading to a limited amount of data that can be efficiently used.

Two principle questions on labelled datasets are how to get the data and how to label it. Sometimes the nature of the problem allows us to automate the generation and labelling stages. Conventional workflow for dealing with such datasets adopted in machine learning is generating and storing the labelled data first and then using it for the training. However, for huge datasets, such an approach can be restrictive for huge datasets due to the limits for available memory and storage.

To alleviate these limitations, we propose an alternative approach, in which the data are generated on the fly by the CPU and the training performed by the GPU in parallel. This approach has several advantages:

1. It consumes less memory, and we do not need to store the data.
2. It can go for an unlimited amount of time and generate as much data as we want.
3. Each training epoch has unique data that may help to produce a better model.

In this work, we present our implementation of the above approach as a workflow using Keras/TensorFlow framework and Python's generator and report on the experiments with the RSA algorithm as a case study. In such a setup, the available resources can be efficiently utilised. The CPU is used to generate the data, and the GPU is used to train the model in parallel.

8.3 Algorithmically Generated Data

The source of the datasets used in supervised machine learning can vary depending on the problem at hand. We consider the following general categories of data based on how they have been generated and labelled:

1. **Non-algorithmically Generated Data**: This is the most common type of data where the data are collected from the environment and labelled with human involvement. For example, images that have been captured and labelled by a human.
2. **Semi-algorithmically Generated Data**: This data where one stage of the process handled by the algorithm. For example, a human might generate the data, then labelled by an algorithm, or vice versa—for example, using a signature-based intrusion detection system to label the computer network traffic.
3. **Algorithmically Generated Data**: any data can be generated and labelled by using an algorithm. Such data can be found in simulation or generated by a mathematical experiment. For example, one can use machine learning to try to factorise prime numbers.

Our work is focused on Algorithmically Generated Data, and we will take the security of the RSA algorithm as a case study.

8.4 Handling The Data

By the convention, we can approach the problem by just generating the dataset and the labels. Then, use those data to train the model. This approach might be a suitable approach for the first and second types of data, but it has its limitation for the third type:

- There is a limit of how much data can be generated because of the amount of memory required.
- It may be unknown in advance how much data will be sufficient for solving the problem at hand. Furthermore, if the amount of data generated is less than optimal, the learned model might be susceptible to overfitting or underfitting.

To deal with these issues, we propose an alternative approach. Instead of separating the two phases of generating the model (Getting the data and train the model), we can run both phases in parallel. A part of the application will keep generating the data for as much as needed and another part for training the model. In a typical implementation, the CPU will be used for the data generation and GPU for training the model. This approach has several advantages:

- It requires far less memory than another approach;
- The data can be generated on-demand, and it can go forever;
- The learning process may result in a better-generated model. For each training epoch, the data might be unique, subject to the properties of the data generation algorithm. This may result in a model trained with more data and not prone to overfitting.

We can call the first approach of handling the data a *Finite list workflow*. The first step here is to generate the data and store it on the desk. Then, we used the stored data to train the model. We can call the second approach an *infinite list workflow*. In this approach, we do not generate the data and store it in the disk to use in training. Instead, we generate the data at the time of training the model. Thus, we keep generating new data as much as we want the training process to last. There is no limit on the memory in this process as the data is generated on the fly by using Python's generator. Figure 8.1 illustrates both workflows.

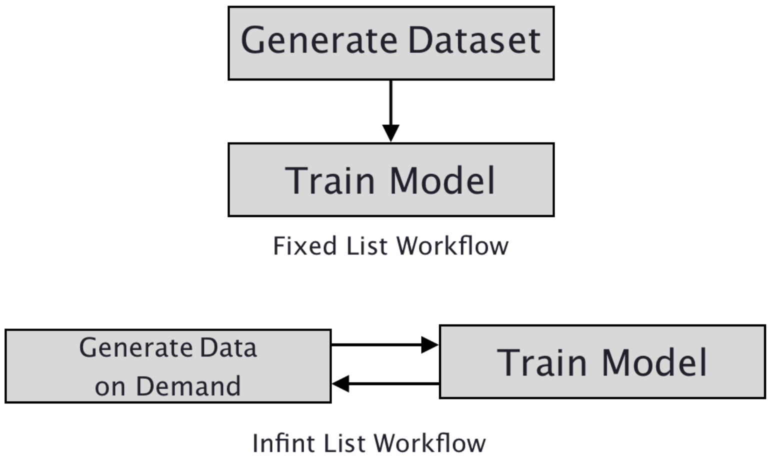


Figure 8.1: Finite List and Infinite List Workflows

8.5 Implementation

To implement the part of generating the data, we will take advantage of Python Generator functions [20]. Python Generator is a function that behaves like an iterator and returns an item only when it is called. The generator function uses the keyword 'yield' instead of 'return'. Here is an example of a generator to return even numbers:

```
def even_gen():
    n = 0
    while True:
        yield n
        n += 2
```

Code 8.1: Python Generator Function Example

For our deep learning framework, there are several options. We use Keras Framework for its simplicity [85]. After defining the architecture in Keras, you have three options to fit the model (fit(), train_on_batch(), fit_generator()). The common function for training is fit(), which requires the whole data to be available in the memory. For our purpose, we will use fit_generator() [23]. It allows us to provide each batch of data from a generator in its turn. The common use of this function is for data augmentation on images. This might include horizontal and vertical flips, rotation, or adding random noise [60]. It allows for an increase in the amount of training data without increasing the actual dataset. We can utilise the same function for our problem differently. Here is a sample code of the function [1] :

```
model.fit_generator(
        train_generator,
        steps_per_epoch=2000,
        epochs=50,
        validation_data=val_generator,
        validation_steps=800)
```

Code 8.2: Keras fit_generator() Example

8.6 Case Study: RSA Algorithm

One of the most famous computational problems in mathematics is *prime factorisation* [37]. Its complexity underpins the security of a well-known public key RSA encryption algorithm [68]. When we have two large prime numbers, it is computationally easy to multiply them but difficult (e.g. unknown to be possible in PTIME) to factorise the result back. This is a good problem to demonstrate the use of Algorithmically Generated Data.[2]

In this case study, we report on attempts to attack the RSA cryptosystem using deep learning methods and the generator. There are several ways to approach the problem. We reformulate the problem as a supervised machine learning problem. We examine two settings. In the first one, we consider a closed pre-computed list of primes. In the second, the generation of the primes goes as long as the learning process goes. Finally, we reported the results and future direction of the work—the goal of this case study is to show how the generator evaluation can be implemented.

[1]Note that the newer version of TensorFlow/Keras v.2 deprecated the use of the fit_generator() method. We recommend using Tensorflow v.1.15.* to run the code.

[2]After we started this research, we have been positively encouraged to continue by the prediction made for the year 2019 by R. Lipton and K W Regan in the well-known blog in Computer Science: "Deep learning methods will be found able to solve integer factoring. This will place current cryptography is trouble. " Jan 6, 2019, `https://rjlipton.wordpress.com"`

8.6.1 RSA Algorithm

RSA (Rivest-Shamir-Adleman) is one of the most popular public key cryptosystems. It is based on the difficulty of prime number factorisation. The key generation of RSA starts by selecting two prime numbers p and q at random. The numbers should be similar in magnitude but differ in length by a few digits to make factoring harder [64]. Then, the values $n = p \cdot q$ and $\phi(n) = (p-1)(q-1)$ are computed. The n value will be used as modulus for the encryption/decryption process and it is a public value. Choose an integer e such that $1 < e < \phi(n)$ and gcd(e, $\phi(n)) = 1$. Calculate d as $d = e^{-1}(mod\ \phi(n))$. The encryption proceeds as follows:

$$c \ \equiv \ m^e \ (n). \tag{8.1}$$

where is a number encoding a data block. Similarly, the decryption proceeds as follows:

$$m \ \equiv \ c^d \ (n). \tag{8.2}$$

There are two main categories to attack: brute-force the key (attack the key), attack the algorithm itself. Selecting a sizeable random key means the longer it takes to brute-force the key. This might not be available based on the computational power that exists. The NIST recommends 2048-bit keys for RSA [5]. The largest factorised number n as we write is 829-bit [25]. Choosing the attack on the cryptosystem might be more visible.

There are three different ways to attack RSA:

1. Factorise n to p and q. This is the famous way to benchmark the security of the RSA.
2. Finding $\phi(n)$ from n. This will allow us to calculate d (private key), or p and q.
3. Attacking d out of public values. This one is the most difficult.

The rise of deep learning makes us question if we can use it to examine the security of RSA with the three ways available to attack. The problem can be presented as supervised learning. We think such problems are a good fit for the generator evaluation method.

8.6.2 Methodology

This problem has many dimensions: prime size to choose, the attack type, the way to format the data to machine learning, what data to give the model, the way to feed the data to learning (Finite, Infinite) and deep learning architectures to select.

Increasing the key size makes the problem way harder. So we will examine prime size as small as 16-bit (n size of 32-bit) and as large as 512-bit (n size of 1024-bit). We considered the value of e to be 65537 (standard).

We consider the attacks as factorising n, finding $\phi(n)$ from n, and attacking d. For finding $\phi(n)$ attack, around half of $\phi(n)$ bits is same as n. So, we are interested in finding the lower band bits of $\phi(n)$. When we report the accuracy, we reported all the bits predictions.

There are several ways we can format the data to machine learning. We can format it as binary, where each feature of the input represents the location of the bit. We can also feed it as the location of the decimal value or as a whole number. The same thing applies to the output. In our experiment, we consider the bit location method—each bit is considered a feature in the input to the model.

We considered this problem as supervised learning. We want to learn information about the output based on the input we have. In our case, the input is the public value, and the output is the value we want to learn/attack. The input that we will feed to the machine learning model is the value of n (public value). The output will be based on the attack we want. If we want to factorise it to p and q, then the output will be the value of p and q. The input and the output of the three scenarios are as follows:

- *Factorise n to p and q*: the input is the value n, the output are p and q. The smallest value between p and q will be first in order for the output.
- *Finding $\phi(n)$ from n*: the input is the value n, the output is the value $\phi(n)$.
- *Attacking d out of public values*: the input is the value n, the output is the value d.

There are different deep learning architectures that can be tested. The one that gives us a good result is the use of Convolutional Neural Network (CNN, or ConvNet) [45].

The problem can also be divided into two parts. The first is the finite list when we have a fixed number of primes and want to learn from them. The second is an infinite list when we considered an infinite number of primes that can go as learning goes. The visualisation of both workflows is shown in Figure 8.1.

Finite List

In this scenario, we first create the dataset and then train the model to learn. We fixed the number of primes (e.g. 300). We then generate a dataset that contains all n values with the output that we need. We divide the data into training and testing sets. The testing is 10% or less of the data. Then report the accuracy of the prediction in the testing set. We experiment with a finite list to show the challenges of the conventional way. We used Python's generators here as well, but only for the first step, to generate the data and store it in the hard drive.

Infinite List

In this scenario, we generate data as long as learning running. The details of this approach are explained in detail [73].

We used a Python generator. It requires far less memory than the finite list workflow. The data can be generated on-demand, and it can go forever. The learning process may result in a better model. Result in a model trained with more data and not prone to overfitting.

In this workflow, we do not have a testing set. Since every time the data is new, and after a certain time, the accuracy stays the same.

8.6.3 Implementation

We used AMD Ryzen Threadripper 1920X CPU (12 cores and 24 threads) with NVIDIA TITAN V GPU and 32 GB of RAM.

For the prime number generator, we used PyCryptodome (library based on Python)[24].

The first workflow to examine is a finite list when we separate the generation and training. The dataset size is 5 million instances. For such a dataset, we could not generate it in our workstation since it requires larger memory. For that, we used Amazon Cloud Services (AWS) instance that has 192 GB RAM and takes around 2 hours to create. The process here includes the conversion from the NumPy array [56] to Panda dataframe [19].

To store the dataframe as a CSV file, it takes 1 hour and 57 minutes and uses 56.3 GB. Unfortunately, we could not read this file back on our workstation. So, we used a better format called Feather Format [91]. Feather format stores the same data with 1.19 GB, and the reading/writing takes less than a second.

When we use this data for the training, it takes 11.3 GB of RAM (The size of the model is 134,284,800 of total parameters). The model architecture is based on a convolutional neural network[45].

By comparison, we need only 1.8 GB of RAM when using our method with the same model size (134,284,800 of total parameters). Therefore, the usage of the memory is the same for 5 million input or 50 million.

We can assume that 5 million inputs require ≈10 GB of RAM using the finite list method. For 100 million instances, we will need ≈200 GB of RAM, which is not available for the most typical computer. In comparison, the use of the proposed method allows train the model with such big data.

Here is a sample code for our data generator:

```
def primes_generator(batch_size=256):
 prime_size = 512
 while True:
   X = []
   y = []
   for i in range(batch_size):
    primes = [number.getPrime(prime_size)
               for x in range(2)]
    p = min(primes)
    q = max(primes)
    n = format(p*q, 'b').zfill(prime_size*2)
    p_bin = format(p, 'b').zfill(prime_size)
    q_bin = format(q, 'b').zfill(prime_size)
    X.append(list(map(bool,
                   [int(d) for d in str(n)]   )))
    y.append(list(map(bool,
                   [int(d) for d in str(p_bin)])) +
     list(map(bool,  [int(d) for d in str(q_bin)])))
 yield ( np.array(X) , np.array(y)  )
```

Code 8.3: Data Generator

And here is a sample for fitting the model:

```
model.fit_generator(
  primes_generator(batch_size=256),
  steps_per_epoch=10000,
  validation_steps=20,
  validation_data=primes_generator(batch_size=256),
  epochs=1000, use_multiprocessing=True, workers=22 )
```

Code 8.4: Keras fit_generator() Method

When we fit the model, multiple threads in the CPU will run to generate the data. This can be specified by *use_multiprocessing=True* and *workers=number of threads*.

8.6.4 Result

The output of this experiment is not the focus of the thesis, but it confirms the difficulty of the problem. So far, no one could factorise a 1024-bit number (each factor 512-bit) with any method [71]. The accuracy that we got with different deep learning models is 50.35%, far from factoring a 1024-bit number. Nevertheless, the workflow proposed here makes experimenting with such problems easier.

The data format for the reported result is a binary representation of the input and the output.

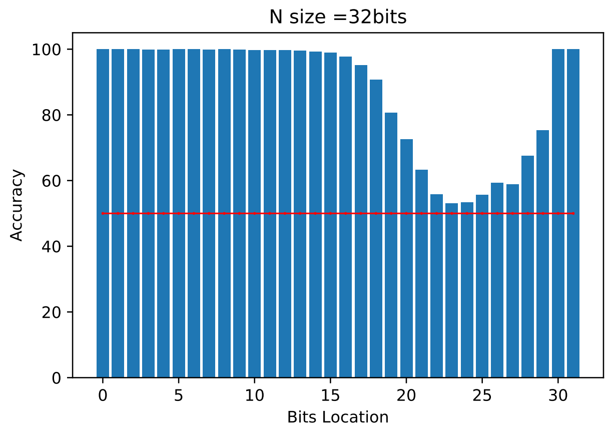


Figure 8.2: Example of Predicting $\phi(n)$ of 32-bit

Finite List

The best result we can report in this workflow with the following setup:

- Try to find $\phi(n)$ from n.
- Prime number Size is 512-bit.
- Having a big ConvNet layer (kernel = 262,144).
- The data size is 1.5 million.

- This setup gives us a training accuracy of 99% and a testing accuracy of 68%.

Figure 8.3 shows the results of only the lower bits of the (n) when the size of the n is 1024-bit. The accuracy result of the lower bits is around 60%.

Even though we think that the result is due to overfitting of that list, but the setting with the finite list setting might allow for *common factor* attacks [8] on RSA. Thus, in a way, ML can replicate the attacks based on such datasets to some extent. However, this phenomenon requires further investigation.

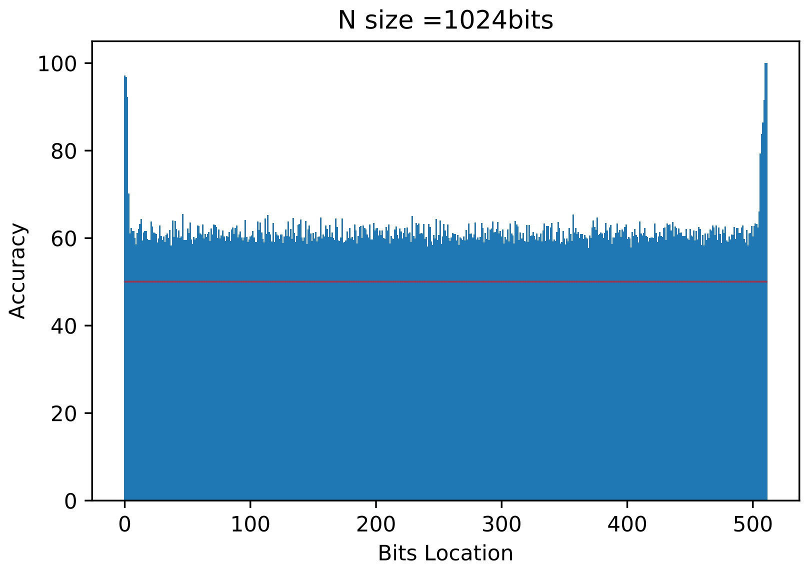


Figure 8.3: Lower Bits Prediction of (n), n Size of 1024-bit in Finite List

Infinite List - the use of the generator

Table 8.1 shows the results of generator evaluation with three different attacks and three different n sizes (64-bit, 256-bit, and 1024-bit). Here are some highlights about the results for Infinite List workflow:

- The accuracy that we got with different deep learning models is 50.35%, far from factoring a 1024-bit number. It confirms the difficulty of the problem. So far, no one could factorise a 1024-bit number (each factor 512-bit) with any method.

Table 8.1: The accuracy results for Infinite List with different n size

Attack Type	64-bit	256-bit	1024-bit
Factorisation	0.5573	0.5143	0.5035
Finding ϕ(n)	0.8095	0.8095	0.8095
Finding D	0.5401	0.5102	0.5026

- In factoring of 128-bit number n, the accuracy that we reached is 55.7%.
- For ϕ(n) problem of 32-bit numbers, we have reached 92.5% accuracy.

Figure 8.4 visualises the accuracy of each bit predicated by the model for the value (n) when the size of the value n is 1024-bits. Thus, almost half of the predicated bits are 100% as half of the bits is shared with the value n.

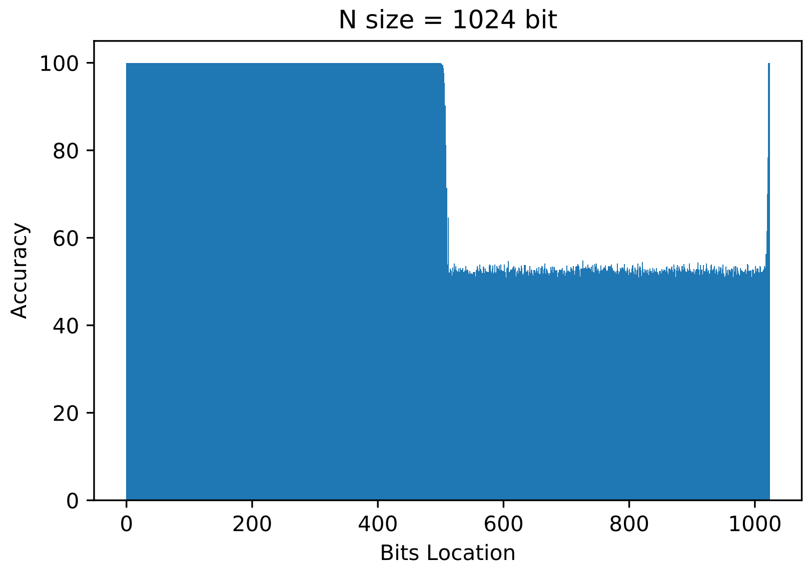


Figure 8.4: Prediction of (n) in Infinite List with 1024-bit of n

8.6.5 Limitations

There might be two limitations to our workflow approach:

- The number of CPU threads. If the processor has a small number of threads supported, this will affect the performance of the training. For our experiment, the processor used supports up to 24 threads. This makes the training goes smoother.

- The time needed to execute the generator function. In our case study, when we increase the prime number's size, the execution's time increases.

8.7 Discussion

Known algorithms can easily attack some small problem sizes. However, thinking about a different way to examine the security of a cryptosystem might lead to more insight into the system. The use of machine learning is an excellent approach to add to our toolkit. In this work, we examine the security of RSA by using deep learning. We reported the highlights of the multi-dimensional problem using deep learning.

The reported results are far from a practical attack. However, it opens the door to try different problems and different machine learning models. Unfortunately, we still did not explore other deep learning methods that might be valuable to examine the problem.

While algorithmically generated data problems are less common than other types, they should be dealt with differently. In this work, we compare the conventional way of dealing with the problem and comparing it with our method. We demonstrate both methods with a case study. We showed how we could train a model with massive data that does not fit in memory.

8.8 Summary

We can summarise this chapter in the following points:

- We highlighted three different types of data obtaining in machine learning. One of them is Algorithmically Generated Data, when the data and the label of the data can be generated algorithmically without human involvement.

- We show that for such a problem, we can deal with it in two different methods: finite list and infinite list.

- We propose the infinite list by using Python generators. The generator will run on the CPU, and the deep learning model will train on the GPU as long as we want. In this method, we do not need a separation between training and testing. As the data is new at every learning epoch. And the learning here will consume far less memory and computer space.

- We demonstrate the generator evaluation with an RSA algorithm security case study as there are three possible ways to attack the algorithm.
- The case study highlighted both methods: finite list and infinite list. And we showed that a finite list has many challenges, and it cannot run in a normal workstation work large size of n value of the RSA.
- The case study results are far from a practical attack for the RSA but show how we can integrate deep learning methods to deal with algorithmically generated data on RSA. This workflow will open the possibility of trying the method with different deep learning architectures and in other problems.

The next chapter will be a conclusion chapter. We will summarise the work and what are possible future works based on this thesis.

Chapter 9

Conclusion and Future Work

9.1 Chapter Overview

This chapter is a summary of the presented work in this thesis. It shows the main finding based on the research questions presented in Chapter 1, followed by suggestions for future directions for the research.

9.2 Summary

The thesis proposed and explored different evaluation methods for the applications of machine learning in cybersecurity. This includes: (1) Cross-datasets Evaluation Method, (2) Time-Based Splitting, (3) Size-Based Splits, and (4) Generator Evaluation Method. In addition, the thesis covers three different areas of cybersecurity: (1) Network-based Intrusion Detection Systems (NIDS), (2) Malware Detection, and (3) Cryptography.

The Chapter 1 focuses on the big picture of the thesis. It started with the motivation for the research. Part of the motivation is the rapid increase in the number of threats. Machine learning is a valuable tool to be used for defence. Thus, the research explores different evaluations that allow for a deeper understanding about the ML models in the cybersecurity space. The chapter is followed by the research questions, methodology, main contributions, publications and the thesis structure.

A literature review of previous work was provided in Chapter 2. It started with defining the evaluation term in the context of this thesis. The evaluation process can be looked at from four angles: (1) Stages of Evaluation, (2) Data-Splitting Methods, (3) Scoring

Metrics, and (4) Overfitting and Underfitting. The main two stages of evaluation are training and testing. And data can be split into two known methods: Handout or k-fold cross-validation. There are several scoring metrics like accuracy, F1 Score, and Area Under the Curve (AUC). The chapter continued with an overview of machine learning and its application to cybersecurity. The chapter ended with an explanation of the datasets that used in the thesis. It also gives an overview of the hardware and software used.

Chapter 3 started with a discussion about the problems in the conventional way to deal with ML application in Network Intrusion Detection Systems. The same dataset is used for training and testing. Current research focuses more on increasing the accuracy than previously reported regardless of the quality of the model. The chapter followed a similar approach and reported the same or better results in three known NIDS datasets. Most of the results have an accuracy of 97% or more. However, there is a doubt about such good results for real-world scenarios. For that, the cross-datasets evaluation method is proposed as a different method for understanding the models deeper than the current method. The ideal of the method is to use two different datasets generated from two different computer networks. The goal is not to overfit the model for a particular computer network setting. The experiments used NSL-KDD and gureKDD datasets. Both datasets were generated from two different computer networks. The output of the experiments shows a big drop in performance. After the investigation about the drop, fixing the service labelling to match between the datasets was required. The fix required reprocessing the gureKDD from the raw data to match NSL-KDD. After the fix, the performance increased again. NSL-KDD is shown to be a better dataset to learn from compared with gureKDD.

The Chapter 4 uses cross-datasets evaluation for the multiclass classification problem in NIDS. It uses the same datasets as before. The challenge is that both datasets do not share the same labels for the attacks. The work was directed to map the attacks into the original KDD dataset categories. It consists of the normal label and four groups of attacks (DoS, R2L, U2R, Probe). NSL-KDD have those categories. The work presented to map the attacks of gureKDD to these five categories. After the mapping was done, several machine learning models were used. The results show the ML models learn about the normality and DoS attack and fail to perform with the other groups. The chapter ended by proposing such an evaluation method with the pipeline of the automation of signature generations and can also use normality as a signature.

The Chapter 5 proposes the use of splitting evaluation in the malware detection application. The splitting is based on the time duration, regardless of the size. The experiments use the SOREL-20M dataset, which contains 20 million samples (malware and benign). We set the size to be 3 months with a total of nine splits. The data contains 10 different groups of malware. The work presented five different flows of training and testing the data: (1) Normal Flow, (2) Forward Accumulative with the Whole Testing, (3) Forward Accumulative with Split Testing, (4) Backward Accumulative with the Whole Testing, (5) Backward Accumulative with Split Testing. Each flow is designed to give a different perspective of understanding the ML model. The Chapter uses two ML models: the Base and Aloha model. The chapter ended with the general results analysis as well as the details about each group of malware.

Chapter 6 follows the same methodology as the previous chapter. The change is in the criteria of splitting the data. Instead of using the time duration, this chapter uses the sample size. Thus, each split has an equal size to the other. The results, in general, show that the use of size base splitting gives a better performance than the time-based splitting.

Chapter 7 attempts to find a better model that has a better performance and less drop. The model was examined with time-based splitting and size-based splitting. All five different flows of training and testing were included. The chapter described the steps that were used to achieve the model. We refer to the model as Model-33. Model-33 shows far less drop in the performance compared with the Base and Aloha models.

Chapter 8 proposes a method to evaluate different types of problems when there is no limit in the data generated. This problem can be referred to as algorithmically generated data. Such problems can be found out of the mathematical problems. We propose the use of the generator evaluation. The training and testing phase are done at the same time. There are two parts to the evaluations: (1) data generator and (2) model training. The data generator runs in the CPU, and the model training runs in the GPU. It results in a model that can be trained as long as we want. The chapter supports the method with a case study in the RSA algorithm in cryptography. It shows three different ways to attack the RSA and how generator evaluation works for this case study.

9.3 Main Findings and Contributions

This section will discuss the main findings of the research. The main research question is *"How can we evaluate different machine learning cybersecurity applications that give more insights compared with the current methods?"*. To answer the main question, the sub-question from the Chapter 1 needs to be answered as follows:

1. *"How can we evaluate the change in the computer network environment effect in the trained model?"*.

   The answer to this question is to use the Cross-Datasets evaluation method. The method uses two datasets generated from two distinct computer networks but shares the same features. In this way, the computer network architecture design has less effect in the trained ML model once tested with a different computer network. More details about the Cross-Datasets evaluation method can be found in Chapter 3.

2. *"What is the effect on the performance of the model when the network environment changed?"*.

   Based on the experiments in Chapter 3, there is no big effect in binary classification. Extra attention on the labelling between datasets is required. However, the models in multiclass classification did not match the performance of the binary classification (Chapter 4). Once the network changes, three different groups of attacks did not perform well.

3. *"What groups of attacks are easier and harder to detect?"*.

   The easiest group of attacks were the Denial of Service (DoS) attacks. The hardest group of attacks, in order, were User to Root (U2R), Remote to Local (R2L), and Probing attacks.

4. *"How can we evaluate how often we need to retrain our model for malware detection?"*.

   We propose the use of the splitting evaluation method to answer this question. The method has five flows of training and testing: 1) Normal Flow, (2) Forward Accumulative with the Whole Testing, (3) Forward Accumulative with Split Testing, (4) Backward Accumulative with the Whole Testing, and (5) Backward Accumulative with Split Testing. Each flow will give a different perspective on understanding how the model will perform over time, thus understanding how often it requires retraining. The method also provides criteria of retraining: a time-based or size-based one. Time-based is when there is a fixed time between each retraining, for example, every three months. Size-based is when there is a number of samples to reach before retraining.

5. *"What is a suitable criteria for retraining the model? Time-based (wait for a fixed duration) or size-based (wait for a number of samples)?"*.

Based on the experiments with the Base, Aloha, and Model-33 models, size- based gives a better average performance. However, the average drop in performance between the time-based and the size-based methods are similar.

6. *"What are the groups of malware that require more frequent model retraining, and what groups require less?"*.

   The answer to this question is based on the average drop of each malware group in all models, both splitting criteria (Time, Size), and includes all five flows of training and splitting. The results are as follows (from most to the least frequent): (from most frequent to the least): Crypto Miner (12.44%), Ransomware (9.67%), Flooder (9.62%), Downloader (8.43%), Packed (8.43%), Adware (7.71%), Dropper (7.50%), Spyware (7.31%), Worm (6.76%), File Infector (5.75%), and Installer (5.13%). There are more details about each group of malware in Chapter 5, 6, and 7.

7. *"How can we evaluate the model in algorithmically generated data problems like the one in cryptanalysis?"*.

   Chapter 8 proposes the use of the generator evaluation method. The idea is the CPU will generate as much data as the training progresses. The training is done on the GPU. This method requires fewer system resources (Memory and Disk), and it can run for as long as we want.

Returning to the main question from Chapter 1, it has been shown in this thesis that various evaluation methods can serve different purposes. The thesis presented three machine learning evaluation methods that serve the application of cybersecurity. The first one is cross-datasets evaluation, which is designed to remove the bias that might occur from the computer network architecture. The second is splitting evaluation. The method aims to understand the behaviour of the malware over time and how the ML models will perform with the changes. The method also proposes using performance drop as a metric to evaluate the performance changes over time. The last evaluation method is the use of generator evaluation. This method design for the problem of algorithmically generated data. The training process can run for an infinite time with less usage of system resources (Memory and Disk).

9.4 Future work

We conclude the thesis with an outline of future work. The research done in this thesis can be extended in different future directions. The following are some suggestions:

- The cross-datasets evaluation method can be used with different datasets. It can also be tried using three or more datasets and mixing them in the training and testing. This might give further insights into the ML models.
- Model-33 was proposed as a better model with the splitting evaluation method compared with the Base and Aloha models. While the model got better results in terms of a performance drop, there is still work that needs to be done to find a model that performs far better than the current state. The model should have better performance and have less drop over time.
- Generator evaluation opens the door for many applications in cryptography or any problem that satisfies the algorithmically generated data. From one side, there are a lot of problems to be tried with the method. On the other, there are many deep learning models to be tested. There are many newly developed deep learning models in other ML applications that can be tested here. It is just a matter of how to map the feature and the labels with the generator. The problem might not be fully solved, but it will give a better understanding.
- The thesis proposes several evaluation methods. However, there is still space to develop other evaluation methods that serve other cybersecurity requirements.

Bibliography

[1] *2021 cyber Threat Report Mid-year update.* URL: `https://www.sonicwall.com/2021-cyber-threat-report/#form`.

[2] Abien Fred M Agarap. "A Neural Network Architecture Combining Gated Recurrent Unit (GRU) and Support Vector Machine (SVM) for Intrusion Detection in Network Traffic Data". In: *Proceedings of the 2018 10th International Conference on Machine Learning and Computing.* ACM. 2018, pp. 26–30.

[3] Zeeshan Ahmad et al. "Network intrusion detection system: A systematic study of machine learning and deep learning approaches". In: *Transactions on Emerging Telecommunications Technologies* 32.1 (2021), e4150.

[4] Ömer Aslan Aslan and Refik Samet. "A comprehensive review on malware detection approaches". In: *IEEE Access* 8 (2020), pp. 6249–6271.

[5] Elaine Barker et al. "Recommendation for key management part 1: General (revision 3)". In: *NIST special publication* 800.57 (2012), pp. 1–147.

[6] *Bias–variance tradeoff.* July 2021. URL: `https://en.wikipedia.org/wiki/Bias%5C%E2%5C%80%5C%93variance_tradeoff`.

[7] Christopher M Bishop et al. *Neural networks for pattern recognition.* Oxford university press, 1995.

[8] Dan Boneh et al. "Twenty years of attacks on the RSA cryptosystem". In: *Notices of the AMS* 46.2 (1999), pp. 203–213.

[9] Andrew P Bradley. "The use of the area under the ROC curve in the evaluation of machine learning algorithms". In: *Pattern recognition* 30.7 (1997), pp. 1145–1159.

[10] *Business models for the long-term storage of internet of things use case data.* URL: `https://www.idc.com/getdoc.jsp?containerId=AP45984120`.

[11] Vitor Cerqueira, Luis Torgo, and Igor Mozetič. “Evaluating time series forecasting models: An empirical study on performance estimation methods”. In: *Machine Learning* 109.11 (2020), pp. 1997–2028.

[12] Marc Chaumont. “Deep learning in steganography and steganalysis”. In: *Digital Media Steganography.* Elsevier, 2020, pp. 321–349.

[13] Kyunghyun Cho et al. “Learning phrase representations using RNN encoder-decoder for statistical machine translation”. In: *arXiv preprint arXiv:1406.1078* (2014).

[14] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. “Fast and accurate deep network learning by exponential linear units (elus)”. In: *arXiv preprint arXiv:1511.07289* (2015).

[15] Michelle Cotton et al. “Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry.” In: *RFC* 6335 (2011), pp. 1–33.

[16] Google - Machine Learning Crash Course. *Classification: Precision and Recall.* 2020. URL: `https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall` (visited on 02/28/2020).

[17] Michael Crawford et al. “Survey of review spam detection using machine learning techniques”. In: *Journal of Big Data* 2.1 (2015), pp. 1–24.

[18] *cross-validation: Evaluating estimator performance.* URL: `https://scikit-learn.org/stable/modules/cross_validation.html`.

[19] Pandas Datafram. *Pandas Datafram.* URL: `https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html` (visited on 11/11/2020).

[20] Python.org DavidFarago. *Python Generator functions documentation.* URL: `https://wiki.python.org/moin/Generators` (visited on 11/11/2020).

[21] Gert De Laet and Gert Schauwers. *Network security fundamentals.* Cisco press, 2005.

[22] Jia Deng et al. “Imagenet: A large-scale hierarchical image database”. In: *2009 IEEE conference on computer vision and pattern recognition.* Ieee. 2009, pp. 248–255.

[23] Keras Documentation. *tf.keras.Model fit_generato : TensorFlow Core v1.15.0.* URL: `https://www.tensorflow.org/versions/r1.15/api_docs/python/tf/keras/Model` (visited on 11/11/2020).

[24] Helder Eijs. *PyCryptodome's documentation.* 2019. URL: https://www.pycryptodome.org/en/latest/.

[25] *Factorization of RSA-250.* Feb. 2020. URL: https://lists.gforge.inria.fr/pipermail/cado-nfs-discuss/2020-February/001166.html.

[26] Evelyn Fix and Joseph Lawson Hodges. "Discriminatory analysis. Nonparametric discrimination: Consistency properties". In: *International Statistical Review/Revue Internationale de Statistique* 57.3 (1989), pp. 238–247.

[27] *Four fundamental shifts in media amp; advertising during 2020.* June 2021. URL: https://doubleverify.com/four-fundamental-shifts-in-media-and-advertising-during-2020/.

[28] Di Freeze. *Global ransomware damage costs predicted to exceed $265 billion by 2031.* June 2021. URL: https://cybersecurityventures.com/global-ransomware-damage-costs-predicted-to-reach-250-billion-usd-by-2031/.

[29] Yoav Freund and Robert E Schapire. "A desicion-theoretic generalization of on-line learning and an application to boosting". In: *European conference on computational learning theory.* Springer. 1995, pp. 23–37.

[30] Jerome H Friedman. "Greedy function approximation: a gradient boosting machine". In: *Annals of statistics* (2001), pp. 1189–1232.

[31] Seymour Geisser. "The predictive sample reuse method with applications". In: *Journal of the American statistical Association* 70.350 (1975), pp. 320–328.

[32] Christian A Hammerschmidt et al. "Reliable machine learning for networking: Key issues and approaches". In: *2017 IEEE 42nd Conference on Local Computer Networks (LCN).* IEEE. 2017, pp. 167–170.

[33] Richard Harang and Ethan M Rudd. "SOREL-20M: A Large Scale Benchmark Dataset for Malicious PE Detection". In: *arXiv preprint arXiv:2012.07634* (2020).

[34] NL Hjort. *Pattern recognition and neural networks.* Cambridge university press, 1996.

[35] Tin Kam Ho. "Random decision forests". In: *Proceedings of 3rd international conference on document analysis and recognition.* Vol. 1. IEEE. 1995, pp. 278–282.

[36] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[37] *Integer factorization*. Oct. 2020. URL: `https://en.wikipedia.org/wiki/Integer_factorization` (visited on 11/11/2020).

[38] Kaspersky. *The number of new malicious files detected every day increases by 5.2% TO 360,000 in 2020*. May 2021. URL: `https://www.kaspersky.com/about/press-releases/2020_the-number-of-new-malicious-files-detected-every-day-increases-by-52-to-360000-in-2020`.

[39] Guolin Ke et al. "Lightgbm: A highly efficient gradient boosting decision tree". In: *Advances in neural information processing systems* 30 (2017), pp. 3146–3154.

[40] Simon Kemp. *Digital 2021 JULY Global Statshot report - DATAREPORTAL – global Digital insights*. July 2021. URL: `https://datareportal.com/reports/digital-2021-july-global-statshot`.

[41] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[42] Mark A Kramer. "Nonlinear principal component analysis using autoassociative neural networks". In: *AIChE journal* 37.2 (1991), pp. 233–243.

[43] Alex Krizhevsky, Geoffrey Hinton, et al. "Learning multiple layers of features from tiny images". In: (2009).

[44] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).

[45] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[46] Yann LeCun et al. "Handwritten digit recognition with a back-propagation network". In: *Advances in neural information processing systems* 2 (1989).

[47] Ming Liu et al. "Host-based intrusion detection system with system calls: Review and future trends". In: *ACM Computing Surveys (CSUR)* 51.5 (2018), pp. 1–36.

[48] Matthew V Mahoney and Philip K Chan. "An analysis of the 1999 DARPA/Lincoln Laboratory evaluation data for network anomaly detection". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2003, pp. 220–237.

[49] *Malware statistics amp; Trends REPORT: AV-TEST*. URL: `https://www.av-test.org/en/statistics/malware/`.

[50] *McAfee Labs Threats Report: June 2021.* URL: `https://www.mcafee.com/enterprise/en-us/lp/threats-reports/jun-2021.html`.

[51] John McHugh. "Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory". In: *ACM Transactions on Information and System Security (TISSEC)* 3.4 (2000), pp. 262–294.

[52] Dean Richard McKinnel et al. "A systematic literature review and meta-analysis on artificial intelligence in penetration testing and vulnerability assessment". In: *Computers & Electrical Engineering* 75 (2019), pp. 175–188.

[53] Tom M Mitchell et al. "Machine learning". In: (1997).

[54] Avinash Navlani. *AdaBoost classifier algorithms using Python Sklearn tutorial.* Nov. 2018. URL: `https://www.datacamp.com/community/tutorials/adaboost-classifier-python`.

[55] Andrew Ng. "Machine learning yearning: Technical strategy for ai engineers in the era of deep learning". In: *Retrieved online at https://www. mlyearning. org* (2019).

[56] *Numpy Array Documentation.* URL: `https://numpy.org/doc/stable/reference/generated/numpy.array.html` (visited on 11/11/2020).

[57] YongJoon Park and JaeChul Park. "Web application intrusion detection system for input validation attack". In: *2008 Third International Conference on Convergence and Hybrid Information Technology.* Vol. 2. IEEE. 2008, pp. 498–504.

[58] Adam Paszke et al. "Pytorch: An imperative style, high-performance deep learning library". In: *arXiv preprint arXiv:1912.01703* (2019).

[59] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[60] Luis Perez and Jason Wang. "The effectiveness of data augmentation in image classification using deep learning". In: *arXiv preprint arXiv:1712.04621* (2017).

[61] Iñigo Perona et al. "Service-independent payload analysis to improve intrusion detection in network traffic". In: *Proceedings of the 7th Australasian Data Mining Conference-Volume 87.* Australian Computer Society, Inc. 2008, pp. 171–178.

[62] Muhammad Shakil Pervez and Dewan Md Farid. "Feature selection and intrusion classification in NSL-KDD cup 99 dataset employing SVMs". In: *Software, Knowledge, Information Management and Applications (SKIMA), 2014 8th International Conference on*. IEEE. 2014, pp. 1–6.

[63] Richard R Picard and R Dennis Cook. "Cross-validation of regression models". In: *Journal of the American Statistical Association* 79.387 (1984), pp. 575–583.

[64] Carl Pomerance. "Smooth numbers and the quadratic sieve". In: *Proc. of an MSRI workshop, J. Buhler and P. Stevenhagen, eds.(to appear)*. 2008.

[65] B Basaveswara Rao and K Swathi. "Fast kNN Classifiers for Network Intrusion Detection System". In: *Indian Journal of Science and Technology* 10.14 (2017).

[66] Sebastian Raschka and Vahid Mirjalili. *Python Machine Learning*. Packt Publishing, 2017.

[67] *Receiver operating characteristic (roc)*. URL: `http://scikit-learn.sourceforge.net/stable/auto_examples/model_selection/plot_roc.html`.

[68] Ronald L Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[69] Said Al-Riyami, Frans Coenen, and Alexei Lisitsa. "A Re-evaluation of Intrusion Detection Accuracy: Alternative Evaluation Strategy". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 2195–2197.

[70] Ishai Rosenberg et al. "Adversarial machine learning attacks and defense methods in the cyber security domain". In: *ACM Computing Surveys (CSUR)* 54.5 (2021), pp. 1–36.

[71] *RSA Factoring Challenge*. Sept. 2020. URL: `https://en.wikipedia.org/wiki/RSA_Factoring_Challenge` (visited on 11/11/2020).

[72] Ethan M Rudd et al. "{ALOHA}: Auxiliary Loss Optimization for Hypothesis Augmentation". In: *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 2019, pp. 303–320.

[73] Frans Coenen Said Al-Riyami Alexei Lisitsa. “An efficient way to deal with algorithmically generated data in deep learning”. In: *Proceedings of the EMerging Technology (EMiT) Conference 2019*. 2019.

[74] Jun Shao. “Linear model selection by cross-validation”. In: *Journal of the American statistical Association* 88.422 (1993), pp. 486–494.

[75] Ali Shiravi et al. “Toward developing a systematic approach to generate benchmark datasets for intrusion detection”. In: *computers & security* 31.3 (2012), pp. 357–374.

[76] Amrit Pal Singh and Manik Deep Singh. “Analysis of host-based and network-based intrusion detection system”. In: *IJ Computer Network and Information Security* 8 (2014), pp. 41–47.

[77] Raman Singh, Harish Kumar, and RK Singla. “An intrusion detection system using network traffic profiling and online sequential extreme learning machine”. In: *Expert Systems with Applications* 42.22 (2015), pp. 8609–8624.

[78] Richard Socher et al. “Recursive deep models for semantic compositionality over a sentiment treebank”. In: *Proceedings of the 2013 conference on empirical methods in natural language processing*. 2013, pp. 1631–1642.

[79] Robin Sommer and Vern Paxson. “Outside the closed world: On using machine learning for network intrusion detection”. In: *2010 IEEE symposium on security and privacy*. IEEE. 2010, pp. 305–316.

[80] Jungsuk Song, Hiroki Takakura, and Yasuo Okabe. “Description of kyoto university benchmark data”. In: *Available at link: http://www. takakura. com/Kyoto_data/BenchmarkData-Description-v5. pdf [Accessed on 15 March 2016]* (2006).

[81] Jungsuk Song et al. “Statistical analysis of honeypot data and building of Kyoto 2006+ dataset for NIDS evaluation”. In: *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*. ACM. 2011, pp. 29–36.

[82] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: `http://jmlr.org/papers/v15/srivastava14a.html`.

[83] Salvatore Stolfo, W Fan, W Lee, et al. *KDD-CUP-99 Task Description*. 1999.

[84] Mahbod Tavallaee et al. “A detailed analysis of the KDD CUP 99 data set”. In: *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*. IEEE. 2009, pp. 1–6.

[85] Keras Team. *Keras Framework*. URL: `https://keras.io/` (visited on 11/11/2020).

[86] *The internet of THINGS: Consumer, industrial amp; public Services 2020-2024*. URL: `https://www.juniperresearch.com/researchstore/devices-technology/internet-of-things-iot-data-research-report/subscription/consumer-industrial-public-services`.

[87] Lionel Sujay Vailshery. *Average number of connected devices in U.S. 2020*. Jan. 2021. URL: `https://www.statista.com/statistics/1107206/average-number-of-connected-devices-us-house/`.

[88] Vladimir N Vapnik. “The nature of statistical learning”. In: *Theory* (1995).

[89] Strother H Walker and David B Duncan. “Estimation of the probability of an event as a function of several independent variables”. In: *Biometrika* 54.1-2 (1967), pp. 167–179.

[90] Zhi Wang et al. “EvilModel 2.0: Bringing Neural Network Models into Malware Attacks”. In: *arXiv preprint arXiv:2109.04344* (2021).

[91] Wesm. *Feather Format*. URL: `https://github.com/wesm/feather` (visited on 11/11/2020).

[92] Wikipedia. *Accuracy paradox*. Accessed: 2018-04-17. 2018. URL: `https://en.wikipedia.org/wiki/Accuracy_paradox`.

[93] *World internet Users statistics and 2021 world population stats*. URL: `https://www.internetworldstats.com/stats.htm`.

[94] Xindong Wu et al. “Top 10 algorithms in data mining”. In: *Knowledge and information systems* 14.1 (2008), pp. 1–37.

[95] Chuanlong Yin et al. “A deep learning approach for intrusion detection using recurrent neural networks”. In: *IEEE Access* 5 (2017), pp. 21954–21961.

[96] Indrė Žliobaitė, Mykola Pechenizkiy, and Joao Gama. “An overview of concept drift applications”. In: *Big data analysis: new algorithms for a new society* (2016), pp. 91–114.